Jürgen **KOTZ**
Christian **WENZ**

C# und .NET 8

GRUNDLAGEN, PROFIWISSEN UND REZEPTE

Bonuskapitel und Beispiele unter plus.hanser-fachbuch.de

HANSER

Kotz/Wenz

C# und .NET 8
Grundlagen, Profiwissen und Rezepte

Ihr Plus – digitale Zusatzinhalte!

Auf unserem Download-Portal finden Sie zu diesem Titel kostenloses Zusatzmaterial.

Geben Sie auf **plus.hanser-fachbuch.de** einfach diesen Code ein:

plus-W36m2-T9er4

Bleiben Sie auf dem Laufenden!

Unser **Computerbuch-Newsletter** informiert Sie monatlich über neue Bücher und Termine. Profitieren Sie auch von Gewinnspielen und exklusiven Leseproben. Gleich anmelden unter:

www.hanser-fachbuch.de/newsletter

Jürgen Kotz
Christian Wenz

C# und .NET 8

Grundlagen, Profiwissen und Rezepte

HANSER

Die Autoren:

Jürgen Kotz, München

Christian Wenz, München

Alle in diesem Werk enthaltenen Informationen, Verfahren und Darstellungen wurden nach bestem Wissen zusammengestellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund sind die im vorliegenden Werk enthaltenen Informationen mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autoren und Verlag übernehmen infolgedessen keine juristische Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieser Informationen – oder Teilen davon – entsteht. Ebenso wenig übernehmen Autoren und Verlag die Gewähr dafür, dass die beschriebenen Verfahren usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt also auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Die endgültige Entscheidung über die Eignung der Informationen für die vorgesehene Verwendung in einer bestimmten Anwendung liegt in der alleinigen Verantwortung des Nutzers.

Aus Gründen der besseren Lesbarkeit wird auf die gleichzeitige Verwendung der Sprachformen männlich, weiblich und divers (m/w/d) verzichtet. Sämtliche Personenbezeichnungen gelten gleichermaßen für alle Geschlechter.

Bibliografische Information der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Dieses Werk ist urheberrechtlich geschützt.
Alle Rechte, auch die der Übersetzung, des Nachdruckes und der Vervielfältigung des Buches, oder Teilen daraus, vorbehalten. Wir behalten uns auch eine Nutzung des Werks für Zwecke des Text- und Data Mining nach § 44b UrhG ausdrücklich vor. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren) – auch nicht für Zwecke der Unterrichtsgestaltung – reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

© 2024 Carl Hanser Verlag München, www.hanser-fachbuch.de
Lektorat: Sylvia Hasselbach
Copy editing: Sandra Gottmann, Wasserburg
Umschlagdesign: Marc Müller-Bremer, München, www.rebranding.de
Umschlagrealisation: Max Kostopoulos
Titelmotiv: © thinkstockphotos.de/liuzishan
Satz: Eberl & Koesel Studio, Kempten
Druck und Bindung: CPI books GmbH, Leck
Printed in Germany

Print-ISBN: 978-3-446-47982-1
E-Book-ISBN: 978-3-446-48060-5
E-Pub-ISBN: 978-3-446-48178-7

Inhalt

Vorwort . . . . . **XXIII**

Zusatzmaterial online XXV

Teil I: Grundlagen **1**

1 .NET 8 . . . . . **3**

1.1 Microsofts .NET-Technologie 4
- 1.1.1 Zur Geschichte von .NET 4
- 1.1.2 .NET-Features und Begriffe 7

1.2 .NET Core 14
- 1.2.1 Geschichte von .NET Core 14
- 1.2.2 LTS – Long Term Support und zukünftige Versionen 16
- 1.2.3 .NET Standard 16

1.3 Features von .NET 6 17

1.4 Features von .NET 7 18

1.5 Features von .NET 8 19

2 Einstieg in Visual Studio 2022 . . . . . **21**

2.1 Die Installation von Visual Studio 2022 21
- 2.1.1 Überblick über die Produktpalette 22
- 2.1.2 Anforderungen an Hard- und Software 23

2.2 Unser allererstes C#-Programm 23
- 2.2.1 Vorbereitungen 23
- 2.2.2 Quellcode schreiben 27
- 2.2.3 Programm kompilieren und testen 27
- 2.2.4 Einige Erläuterungen zum Quellcode 28
- 2.2.5 Konsolenanwendungen sind out 29

2.3 Die Windows-Philosophie 29
- 2.3.1 Mensch-Rechner-Dialog 30
- 2.3.2 Objekt- und ereignisorientierte Programmierung 30
- 2.3.3 Programmieren mit Visual Studio 2022 31

2.4 Die Entwicklungsumgebung Visual Studio 2022 33
2.4.1 Neues Projekt 33
2.4.2 Die wichtigsten Fenster 36
2.4.3 Projektvorlagen in Visual Studio 2022 – Minimal APIs 39
2.5 Praxisbeispiele 41
2.5.1 Unsere erste Windows-Forms-Anwendung 41
2.5.2 Umrechnung Euro-Dollar 46
2.6 Neuerungen in Visual Studio 2022 Version 17.8 55

3 Grundlagen der Sprache C# 57

3.1 Grundbegriffe 57
3.1.1 Anweisungen 57
3.1.2 Bezeichner 58
3.1.3 Schlüsselwörter 59
3.1.4 Kommentare 60
3.2 Datentypen, Variablen und Konstanten 61
3.2.1 Fundamentale Typen 61
3.2.2 Wertetypen versus Verweistypen 62
3.2.3 Benennung von Variablen 63
3.2.4 Deklaration von Variablen 63
3.2.5 Typsuffixe 64
3.2.6 Zeichen und Zeichenketten 65
3.2.7 object-Datentyp 67
3.2.8 Konstanten deklarieren 68
3.2.9 Nullable Types 68
3.2.10 Typinferenz 70
3.2.11 Gültigkeitsbereiche und Sichtbarkeit 70
3.3 Konvertieren von Datentypen 71
3.3.1 Implizite und explizite Konvertierung 71
3.3.2 Welcher Datentyp passt zu welchem? 73
3.3.3 Konvertieren von string 74
3.3.4 Die Convert-Klasse 75
3.3.5 Die Parse-Methode 76
3.3.6 Boxing und Unboxing 77
3.4 Operatoren 78
3.4.1 Arithmetische Operatoren 79
3.4.2 Zuweisungsoperatoren 81
3.4.3 Logische Operatoren 82
3.4.4 Rangfolge der Operatoren 85
3.5 Kontrollstrukturen 86
3.5.1 Verzweigungsbefehle 86
3.5.2 Schleifenanweisungen 91
3.6 Benutzerdefinierte Datentypen 93
3.6.1 Enumerationen 94
3.6.2 Strukturen 95

3.7 Nutzerdefinierte Methoden . . . 97
3.7.1 Methoden mit Rückgabewert . . . 98
3.7.2 Methoden ohne Rückgabewert . . . 99
3.7.3 Parameterübergabe mit ref . . . 101
3.7.4 Parameterübergabe mit out . . . 102
3.7.5 Methodenüberladung . . . 103
3.7.6 Optionale Parameter . . . 104
3.7.7 Benannte Parameter . . . 105
3.8 Praxisbeispiele . . . 106
3.8.1 Vom PAP zur Konsolenanwendung . . . 106
3.8.2 Ein Konsolen- in ein Windows-Programm verwandeln . . . 109
3.8.3 Schleifenanweisungen verstehen . . . 111
3.8.4 Benutzerdefinierte Methoden überladen . . . 114
3.8.5 Anwendungen von Visual Basic nach C# portieren . . . 117

4 OOP-Konzepte . . . 125

4.1 Kleine Einführung in die OOP . . . 125
4.1.1 Historische Entwicklung . . . 126
4.1.2 Grundbegriffe der OOP . . . 127
4.1.3 Sichtbarkeit von Klassen und ihren Mitgliedern . . . 129
4.1.4 Allgemeiner Aufbau einer Klasse . . . 130
4.1.5 Das Erzeugen eines Objekts . . . 132
4.1.6 Einführungsbeispiel . . . 135
4.2 Eigenschaften . . . 141
4.2.1 Eigenschaften mit Zugriffsmethoden kapseln . . . 141
4.2.2 Berechnete Eigenschaften . . . 143
4.2.3 Lese-/Schreibschutz . . . 145
4.2.4 Property-Accessoren . . . 146
4.2.5 Statische Felder/Eigenschaften . . . 147
4.2.6 Einfache Eigenschaften automatisch implementieren . . . 149
4.3 Methoden . . . 151
4.3.1 Öffentliche und private Methoden . . . 151
4.3.2 Überladene Methoden . . . 152
4.3.3 Statische Methoden . . . 153
4.4 Ereignisse . . . 154
4.4.1 Ereignis hinzufügen . . . 155
4.4.2 Ereignis verwenden . . . 158
4.5 Arbeiten mit Konstruktor und Destruktor . . . 161
4.5.1 Konstruktor und Objektinitialisierer . . . 162
4.5.2 Destruktor und Garbage Collector . . . 165
4.5.3 Mit using den Lebenszyklus des Objekts kapseln . . . 167
4.6 Vererbung und Polymorphie . . . 168
4.6.1 Method-Overriding . . . 168
4.6.2 Klassen implementieren . . . 168

4.6.3 Implementieren der Objekte ... 171
4.6.4 Ausblenden von Mitgliedern durch Vererbung ... 173
4.6.5 Allgemeine Hinweise und Regeln zur Vererbung ... 174
4.6.6 Polymorphes Verhalten ... 176
4.6.7 Die Rolle von System.Object ... 179

4.7 Spezielle Klassen ... 180
4.7.1 Abstrakte Klassen ... 180
4.7.2 Versiegelte Klassen ... 181
4.7.3 Partielle Klassen ... 182
4.7.4 Statische Klassen ... 183

4.8 Schnittstellen (Interfaces) ... 184
4.8.1 Definition einer Schnittstelle ... 185
4.8.2 Implementieren einer Schnittstelle ... 185
4.8.3 Abfragen, ob Schnittstelle vorhanden ist ... 186
4.8.4 Mehrere Schnittstellen implementieren ... 187
4.8.5 Schnittstellenprogrammierung ist ein weites Feld ... 187

4.9 Datensatztypen – Records ... 187
4.9.1 Definition eines Record ... 188
4.9.2 Mutable Properties ... 190
4.9.3 Nicht-destruktive Änderung ... 192
4.9.4 Dekonstruktion ... 193

4.10 Praxisbeispiele ... 194
4.10.1 Eigenschaften sinnvoll kapseln ... 194
4.10.2 Eine statische Klasse anwenden ... 197
4.10.3 Vom fetten zum schlanken Client ... 198
4.10.4 Schnittstellenvererbung verstehen ... 209
4.10.5 Rechner für komplexe Zahlen ... 214
4.10.6 Sortieren mit IComparable/IComparer ... 222
4.10.7 Einen Objektbaum in generischen Listen abspeichern ... 227
4.10.8 OOP beim Kartenspiel erlernen ... 232
4.10.9 Eine Klasse zur Matrizenrechnung entwickeln ... 237
4.10.10 Vererbung von Records ... 243

5 Arrays, Strings, Funktionen ... 245

5.1 Datenfelder (Arrays) ... 245
5.1.1 Array deklarieren ... 246
5.1.2 Array instanziieren ... 246
5.1.3 Array initialisieren ... 247
5.1.4 Zugriff auf Array-Elemente ... 248
5.1.5 Zugriff mittels Schleife ... 249
5.1.6 Mehrdimensionale Arrays ... 250
5.1.7 Zuweisen von Arrays ... 252
5.1.8 Arrays aus Strukturvariablen ... 253
5.1.9 Löschen und Umdimensionieren von Arrays ... 254

5.1.10 Eigenschaften und Methoden von Arrays 256
5.1.11 Übergabe von Arrays 257
5.2 Verarbeiten von Zeichenketten 259
5.2.1 Zuweisen von Strings 259
5.2.2 Eigenschaften und Methoden von String-Variablen 260
5.2.3 Wichtige Methoden der String-Klasse 262
5.2.4 Die StringBuilder-Klasse 264
5.3 Datums- und Zeitberechnungen 267
5.3.1 Die DateTime-Struktur 267
5.3.2 Wichtige Eigenschaften von DateTime-Variablen 268
5.3.3 Wichtige Methoden von DateTime-Variablen 269
5.3.4 Wichtige Mitglieder der DateTime-Struktur 270
5.3.5 Konvertieren von Datumstrings in DateTime-Werte 271
5.3.6 Die TimeSpan-Struktur 271
5.3.7 DateOnly und TimeOnly 273
5.4 Mathematische Funktionen 274
5.4.1 Überblick 274
5.4.2 Zahlen runden 274
5.4.3 Winkel umrechnen 275
5.4.4 Potenz- und Wurzeloperationen 275
5.4.5 Logarithmus und Exponentialfunktionen 275
5.4.6 Zufallszahlen erzeugen 276
5.4.7 Kreisberechnung 277
5.5 Zahlen- und Datumsformatierungen 277
5.5.1 Anwenden der ToString-Methode 278
5.5.2 Anwenden der Format-Methode 279
5.5.3 Stringinterpolation 280
5.6 Praxisbeispiele 281
5.6.1 Zeichenketten verarbeiten 281
5.6.2 Zeichenketten mit StringBuilder addieren 284
5.6.3 Methodenaufrufe mit Array-Parametern 287

6 Weitere Sprachfeatures 293

6.1 Namespaces (Namensräume) 293
6.1.1 Ein kleiner Überblick 293
6.1.2 Einen eigenen Namespace einrichten 294
6.1.3 Die using-Anweisung 296
6.1.4 Namespace Alias 297
6.1.5 Globale using-Anweisungen 297
6.2 Operatorenüberladung 298
6.2.1 Syntaxregeln 298
6.2.2 Praktische Anwendung 299
6.3 Collections (Auflistungen) 300
6.3.1 Die Schnittstelle IEnumerable 300

6.3.2 ArrayList ... 303
6.3.3 Hashtable ... 304
6.3.4 Indexer ... 305
6.4 Generics ... 307
6.4.1 Generics bieten Typsicherheit ... 308
6.4.2 Generische Methoden ... 309
6.4.3 yield – Iteratoren ... 310
6.5 Generische Collections ... 310
6.5.1 List-Collection statt ArrayList ... 310
6.5.2 Vorteile generischer Collections ... 312
6.5.3 Constraints ... 312
6.6 Das Prinzip der Delegates ... 313
6.6.1 Delegates sind Methodenzeiger ... 313
6.6.2 Einen Delegate-Typ deklarieren ... 314
6.6.3 Ein Delegate-Objekt erzeugen ... 314
6.6.4 Anonyme Methoden ... 316
6.6.5 Lambda-Ausdrücke ... 317
6.6.6 Lambda-Ausdrücke in der Task Parallel Library ... 320
6.6.7 Action<> und Func<> ... 321
6.7 Dynamische Programmierung ... 323
6.7.1 Wozu dynamische Programmierung? ... 323
6.7.2 Das Prinzip der dynamischen Programmierung ... 324
6.7.3 Optionale Parameter sind hilfreich ... 326
6.7.4 Kovarianz und Kontravarianz ... 327
6.8 Weitere Datentypen ... 328
6.8.1 BigInteger ... 328
6.8.2 Complex ... 330
6.8.3 Tuple<> ... 331
6.8.4 SortedSet<> ... 332
6.9 Praxisbeispiele ... 333
6.9.1 ArrayList versus generische List ... 333
6.9.2 Generische IEnumerable-Interfaces implementieren ... 336
6.9.3 Delegates, Func, anonyme Methoden, Lambda Expressions ... 340

7 Einführung in LINQ ... 345

7.1 LINQ-Grundlagen ... 345
7.1.1 Die LINQ-Architektur ... 345
7.1.2 Anonyme Typen ... 347
7.1.3 Erweiterungsmethoden ... 348
7.2 Abfragen mit LINQ to Objects ... 349
7.2.1 Grundlegendes zur LINQ-Syntax ... 350
7.2.2 Zwei alternative Schreibweisen von LINQ-Abfragen ... 351
7.2.3 Übersicht der wichtigsten Abfrageoperatoren ... 352

7.3 LINQ-Abfragen im Detail ... 353
7.3.1 Die Projektionsoperatoren Select und SelectMany ... 354
7.3.2 Der Restriktionsoperator Where ... 355
7.3.3 Die Sortierungsoperatoren OrderBy und ThenBy ... 356
7.3.4 Der Gruppierungsoperator GroupBy ... 357
7.3.5 Verknüpfen mit Join ... 360
7.3.6 Aggregat-Operatoren ... 360
7.3.7 Verzögertes Ausführen von LINQ-Abfragen ... 362
7.3.8 Konvertierungsmethoden ... 363
7.3.9 Abfragen mit PLINQ ... 364
7.4 Praxisbeispiele ... 367
7.4.1 Die Syntax von LINQ-Abfragen verstehen ... 367
7.4.2 Aggregat-Abfragen mit LINQ ... 370
7.4.3 LINQ im Schnelldurchgang erlernen ... 373
7.4.4 Strings mit LINQ abfragen und filtern ... 375
7.4.5 Duplikate aus einer Liste entfernen ... 377
7.4.6 Arrays mit LINQ initialisieren ... 379
7.4.7 Arrays per LINQ mit Zufallszahlen füllen ... 381
7.4.8 Einen String mit Wiederholmuster erzeugen ... 383
7.4.9 Mit LINQ Zahlen und Strings sortieren ... 384
7.4.10 Mit LINQ Collections von Objekten sortieren ... 386
7.4.11 Where – Deep Dive ... 388

8 Neuerungen von C# im Überblick ... 397
8.1 C# 4.0 – Visual Studio 2010 ... 398
8.1.1 Datentyp dynamic ... 398
8.1.2 Benannte und optionale Parameter ... 399
8.1.3 Kovarianz und Kontravarianz ... 400
8.2 C# 5.0 – Visual Studio 2012 ... 400
8.2.1 Async und Await ... 401
8.2.2 CallerInfo ... 402
8.3 Visual Studio 2013 ... 403
8.4 C# 6.0 – Visual Studio 2015 ... 403
8.4.1 String Interpolation ... 403
8.4.2 Schreibgeschützte AutoProperties ... 403
8.4.3 Initialisierer für AutoProperties ... 404
8.4.4 Expression Body Member ... 404
8.4.5 using static ... 404
8.4.6 Bedingter Nulloperator ... 405
8.4.7 Ausnahmenfilter ... 405
8.4.8 nameof-Ausdrücke ... 406
8.4.9 await in catch und finally ... 406
8.4.10 Indexinitialisierer ... 406
8.5 C# 7.0 – Visual Studio 2017 ... 407

8.5.1 out-Variablen 407
8.5.2 Tupel 407
8.5.3 Mustervergleich 408
8.5.4 Discards 410
8.5.5 Lokale ref-Variablen und Rückgabetypen 411
8.5.6 Lokale Funktionen 411
8.5.7 Mehr Expression Body Member 411
8.5.8 throw-Ausdrücke 412
8.5.9 Verbesserung der numerischen literalen Syntax 412
8.6 C# 7.1 bis 7.3 – Visual Studio 2019 412
8.6.1 C# 7.1 412
8.6.2 C# 7.2 414
8.6.3 C# 7.3 415
8.6.4 Visual Studio 2019 – Live Share 415
8.7 C# 8.0 418
8.7.1 Standardschnittstellenmethoden 418
8.7.2 Vereinfachung von switch-Ausdrücken 420
8.7.3 Eigenschaftenmuster 421
8.7.4 Vereinfachte using-Ressourcenschutzblöcke 421
8.7.5 Null-Coalescing-Zuweisungen 422
8.7.6 Nullable Referenztypen 423
8.7.7 Indizes und Bereiche 423
8.7.8 Weitere Sprachneuerungen 425
8.8 C# 9.0 425
8.8.1 Records 425
8.8.2 Init-only Setter 426
8.8.3 Weitere Verbesserungen in Musterausdrücken 426
8.8.4 Weitere Sprachneuerungen 426
8.9 C# 10 427
8.9.1 Datensatzstrukturen 427
8.9.2 Globale using-Anweisungen 427
8.9.3 Weitere Sprachneuerungen 427
8.10 C# 11 428
8.10.1 Raw String Literals 428
8.10.2 Generische Mathematik 430
8.10.3 Generische Attribute 431
8.10.4 Listenmuster 431
8.10.5 Lokale Dateitypen 432
8.10.6 Required Member 433
8.10.7 Automatische Standardstrukturen 434
8.11 C# 12 434
8.11.1 Primäre Konstruktoren auch für Klassen und Strukturen 434
8.11.2 Sammlungsausdrücke 435
8.11.3 Weitere Sprachneuerungen 435

Teil II: Desktop-Anwendungen ... **437**

9 Einführung in WPF ... **439**

9.1 Einführung ... 439
9.1.1 Was kann eine WPF-Anwendung? ... 440
9.1.2 Die eXtensible Application Markup Language ... 442
9.1.3 Unsere erste XAML-Anwendung ... 443
9.1.4 Zielplattformen ... 449
9.1.5 Applikationstypen ... 449
9.1.6 Vor- und Nachteile von WPF-Anwendungen ... 450
9.1.7 Weitere Dateien im Überblick ... 451

9.2 Alles beginnt mit dem Layout ... 453
9.2.1 Allgemeines zum Layout ... 453
9.2.2 Positionieren von Steuerelementen ... 455
9.2.3 Canvas ... 458
9.2.4 StackPanel ... 459
9.2.5 DockPanel ... 461
9.2.6 WrapPanel ... 463
9.2.7 UniformGrid ... 464
9.2.8 Grid ... 465
9.2.9 ViewBox ... 470
9.2.10 TextBlock ... 471

9.3 Das WPF-Programm ... 475
9.3.1 Die App-Klasse ... 475
9.3.2 Das Startobjekt festlegen ... 475
9.3.3 Kommandozeilenparameter verarbeiten ... 477
9.3.4 Die Anwendung beenden ... 478
9.3.5 Auswerten von Anwendungsereignissen ... 478

9.4 Die Window-Klasse ... 479
9.4.1 Position und Größe festlegen ... 480
9.4.2 Rahmen und Beschriftung ... 480
9.4.3 Das Fenster-Icon ändern ... 480
9.4.4 Anzeige weiterer Fenster ... 481
9.4.5 Transparenz ... 481
9.4.6 Abstand zum Inhalt festlegen ... 482
9.4.7 Fenster ohne Fokus anzeigen ... 482
9.4.8 Ereignisfolge bei Fenstern ... 483
9.4.9 Ein paar Worte zur Schriftdarstellung ... 483
9.4.10 Ein paar Worte zur Darstellung von Controls ... 486
9.4.11 Wird mein Fenster komplett mit WPF gerendert? ... 487

10 Übersicht WPF-Controls ... 489

10.1 Allgemeingültige Eigenschaften ... 489
10.2 Label ... 491
10.3 Button, RepeatButton, ToggleButton ... 492
10.3.1 Schaltflächen für modale Dialoge ... 493
10.3.2 Schaltflächen mit Grafik ... 494
10.4 TextBox, PasswordBox ... 495
10.4.1 TextBox ... 495
10.4.2 PasswordBox ... 497
10.5 CheckBox ... 498
10.6 RadioButton ... 500
10.7 ListBox, ComboBox ... 501
10.7.1 ListBox ... 502
10.7.2 ComboBox ... 505
10.7.3 Den Content formatieren ... 506
10.8 Image ... 508
10.8.1 Grafik per XAML zuweisen ... 508
10.8.2 Grafik zur Laufzeit zuweisen ... 508
10.8.3 Bild aus Datei laden ... 510
10.8.4 Die Grafikskalierung beeinflussen ... 511
10.9 Slider, ScrollBar ... 512
10.9.1 Slider ... 512
10.9.2 ScrollBar ... 514
10.10 ScrollViewer ... 514
10.11 Menu, ContextMenu ... 515
10.11.1 Menu ... 516
10.11.2 Tastenkürzel ... 517
10.11.3 Grafiken ... 518
10.11.4 Weitere Möglichkeiten ... 519
10.11.5 ContextMenu ... 520
10.12 ToolBar ... 521
10.13 StatusBar, ProgressBar ... 524
10.13.1 StatusBar ... 524
10.13.2 ProgressBar ... 526
10.14 Border, GroupBox, BulletDecorator ... 526
10.14.1 Border ... 527
10.14.2 GroupBox ... 528
10.14.3 BulletDecorator ... 529
10.15 Expander, TabControl ... 531
10.15.1 Expander ... 531
10.15.2 TabControl ... 533
10.16 Popup ... 534

10.17 TreeView ... 537
10.18 ListView ... 541
10.19 DataGrid ... 541
10.20 Calendar/DatePicker ... 542
10.21 Ellipse, Rectangle, Line und Co. ... 546
10.21.1 Ellipse ... 547
10.21.2 Rectangle ... 547
10.21.3 Line ... 548

11 Wichtige WPF-Techniken ... 549

11.1 Eigenschaften ... 549
11.1.1 Abhängige Eigenschaften (Dependency Properties) ... 549
11.1.2 Angehängte Eigenschaften (Attached Properties) ... 551
11.2 Einsatz von Ressourcen ... 551
11.2.1 Was sind eigentlich Ressourcen? ... 551
11.2.2 Wo können Ressourcen gespeichert werden? ... 552
11.2.3 Wie definiere ich eine Ressource? ... 553
11.2.4 Statische und dynamische Ressourcen ... 554
11.2.5 Wie werden Ressourcen adressiert? ... 556
11.2.6 Systemressourcen einbinden ... 557
11.3 Das WPF-Ereignismodell ... 557
11.3.1 Einführung ... 557
11.3.2 Routed Events ... 558
11.3.3 Direkte Events ... 561
11.4 Verwendung von Commands ... 561
11.4.1 Einführung zu Commands ... 561
11.4.2 Verwendung vordefinierter Commands ... 562
11.4.3 Das Ziel des Commands ... 564
11.4.4 Vordefinierte Commands ... 565
11.4.5 Commands an Ereignismethoden binden ... 565
11.4.6 Wie kann ich ein Command per Code auslösen? ... 566
11.4.7 Command-Ausführung verhindern ... 567
11.5 Das WPF-Style-System ... 567
11.5.1 Übersicht ... 567
11.5.2 Benannte Styles ... 568
11.5.3 Typ-Styles ... 570
11.5.4 Styles anpassen und vererben ... 571
11.6 Verwenden von Triggern ... 573
11.6.1 Eigenschaften-Trigger (Property Triggers) ... 574
11.6.2 Ereignis-Trigger ... 576
11.6.3 Daten-Trigger ... 577
11.7 Einsatz von Templates ... 578
11.7.1 Neues Template erstellen ... 578
11.7.2 Template abrufen und verändern ... 582

11.8 Transformationen, Animationen, StoryBoards 585
11.8.1 Transformationen 585
11.8.2 Animationen mit dem StoryBoard realisieren 590

12 WPF-Datenbindung 597

12.1 Grundprinzip 597
12.1.1 Bindungsarten 598
12.1.2 Wann eigentlich wird die Quelle aktualisiert? 600
12.1.3 Geht es auch etwas langsamer? 601
12.1.4 Bindung zur Laufzeit realisieren 602
12.2 Binden an Objekte 603
12.2.1 Objekte im XAML-Code instanziieren 604
12.2.2 Verwenden der Instanz im C#-Quellcode 605
12.2.3 Anforderungen an die Quell-Klasse 606
12.2.4 Instanziieren von Objekten per C#-Code 607
12.3 Binden von Collections 609
12.3.1 Anforderung an die Collection 609
12.3.2 Einfache Anzeige 610
12.3.3 Navigieren zwischen den Objekten 611
12.3.4 Einfache Anzeige in einer ListBox 613
12.3.5 DataTemplates zur Anzeigeformatierung 614
12.3.6 Mehr zu List- und ComboBox 615
12.3.7 Verwendung der ListView 617
12.4 Noch einmal zurück zu den Details 620
12.4.1 Navigieren in den Daten 620
12.4.2 Sortieren 622
12.4.3 Filtern 622
12.4.4 Live Shaping 623
12.5 Anzeige von Datenbankinhalten 625
12.5.1 Installieren der benötigten NuGet-Pakete 625
12.5.2 Anlegen der Entitätsklassen 627
12.5.3 Die Programmoberfläche 630
12.5.4 Der Zugriff auf die Daten 632
12.6 Formatieren von Werten 634
12.6.1 IValueConverter 634
12.6.2 BindingBase.StringFormat-Eigenschaft 636
12.7 Das DataGrid als Universalwerkzeug 637
12.7.1 Grundlagen der Anzeige 637
12.7.2 UI-Virtualisierung 639
12.7.3 Spalten selbst definieren 639
12.7.4 Zusatzinformationen in den Zeilen anzeigen 641
12.7.5 Vom Betrachten zum Editieren 643
12.8 Praxisbeispiel – Collections in Hintergrundthreads füllen 643

13 .NET MAUI ... 649
13.1 Einführung ... 650
13.2 Was kann eine .NET MAUI-Anwendung? ... 651
13.3 Die erste .NET MAUI-App ... 652
13.4 Definieren einer eigenen View ... 661
13.5 Daten in MAUI anzeigen ... 663
13.6 Styles in MAUI ... 667
13.6.1 Styles ... 667
13.6.2 Themes ... 668
13.7 Navigation unter MAUI ... 670

Teil III: Technologien ... 673

14 Asynchrone Programmierung ... 675
14.1 Übersicht ... 676
14.1.1 Multitasking versus Multithreading ... 676
14.1.2 Deadlocks ... 677
14.1.3 Racing ... 678
14.2 Programmieren mit Threads ... 679
14.2.1 Einführungsbeispiel ... 679
14.2.2 Wichtige Thread-Methoden ... 681
14.2.3 Wichtige Thread-Eigenschaften ... 683
14.2.4 Einsatz der ThreadPool-Klasse ... 684
14.3 Sperrmechanismen ... 685
14.3.1 Threading ohne lock ... 686
14.3.2 Threading mit lock ... 687
14.3.3 Die Monitor-Klasse ... 690
14.3.4 Mutex ... 694
14.3.5 Methoden für die parallele Ausführung sperren ... 695
14.3.6 Semaphore ... 696
14.4 Interaktion mit der Programmoberfläche ... 698
14.4.1 Die Werkzeuge ... 699
14.4.2 Einzelne Steuerelemente mit Invoke aktualisieren (Windows Forms) 699
14.4.3 Mehrere Steuerelemente aktualisieren ... 701
14.4.4 Ist ein Invoke-Aufruf nötig? ... 701
14.4.5 Und was ist mit WPF? ... 702
14.5 Timer-Threads ... 704
14.6 Asynchrone Programmierentwurfsmuster ... 705
14.6.1 Kurzübersicht ... 705
14.6.2 Polling ... 706
14.6.3 Callback verwenden ... 708
14.6.4 Callback mit Parameterübergabe verwenden ... 709
14.6.5 Callback mit Zugriff auf die Programmoberfläche ... 710

14.7 Es geht auch einfacher – async und await . . . 712
14.7.1 Der Weg von synchron zu asynchron . . . 712
14.7.2 Achtung: Fehlerquellen! . . . 714
14.7.3 Eigene asynchrone Methoden entwickeln . . . 717
14.8 Asynchrone Streams . . . 719
14.8.1 Datei erstellen . . . 720
14.8.2 Datei lesen mit IAsyncEnumerable<T> . . . 721
14.9 Praxisbeispiele . . . 722
14.9.1 Prozess- und Thread-Informationen gewinnen . . . 722
14.9.2 Ein externes Programm starten . . . 725

15 Die Task Parallel Library . . . 729

15.1 Überblick . . . 729
15.1.1 Parallel-Programmierung . . . 729
15.1.2 Möglichkeiten der TPL . . . 732
15.1.3 Der CLR-Threadpool . . . 733
15.2 Parallele Verarbeitung mit Parallel.Invoke . . . 734
15.2.1 Aufrufvarianten . . . 734
15.2.2 Einschränkungen . . . 736
15.3 Verwendung von Parallel.For . . . 737
15.3.1 Abbrechen der Verarbeitung . . . 738
15.3.2 Auswerten des Verarbeitungsstatus . . . 740
15.3.3 Und was ist mit anderen Iterator-Schrittweiten? . . . 741
15.4 Collections mit Parallel.ForEach verarbeiten . . . 741
15.5 Die Task-Klasse . . . 742
15.5.1 Einen Task erzeugen . . . 742
15.5.2 Den Task starten . . . 743
15.5.3 Datenübergabe an den Task . . . 745
15.5.4 Wie warte ich auf das Ende des Tasks? . . . 746
15.5.5 Tasks mit Rückgabewerten . . . 748
15.5.6 Die Verarbeitung abbrechen . . . 751
15.5.7 Fehlerbehandlung . . . 756
15.5.8 Weitere Eigenschaften . . . 757
15.6 Zugriff auf das User Interface . . . 758
15.6.1 Task-Ende und Zugriff auf die Oberfläche . . . 758
15.6.2 Zugriff auf das UI aus dem Task heraus . . . 760
15.7 Weitere Datenstrukturen im Überblick . . . 762
15.7.1 Threadsichere Collections . . . 762
15.7.2 Primitive für die Threadsynchronisation . . . 762
15.8 Parallel LINQ (PLINQ) . . . 763
15.9 Praxisbeispiele . . . 763
15.9.1 BlockingCollection . . . 763
15.9.2 PLINQ . . . 767

16 Debugging, Fehlersuche und Fehlerbehandlung ... 769
16.1 Der Debugger ... 769
16.1.1 Allgemeine Beschreibung ... 769
16.1.2 Die wichtigsten Fenster ... 770
16.1.3 Debugging-Optionen ... 774
16.1.4 Praktisches Debugging am Beispiel ... 777
16.2 Arbeiten mit Debug und Trace ... 782
16.2.1 Wichtige Methoden von Debug und Trace ... 782
16.2.2 Besonderheiten der Trace-Klasse ... 786
16.2.3 TraceListener-Objekte ... 786
16.3 Caller Information ... 788
16.3.1 Attribute ... 789
16.3.2 Anwendung ... 789
16.4 Fehlerbehandlung ... 790
16.4.1 Anweisungen zur Fehlerbehandlung ... 790
16.4.2 try-catch ... 791
16.4.3 try-finally ... 795
16.4.4 Das Standardverhalten bei Ausnahmen festlegen ... 797
16.4.5 Die Exception-Klasse ... 798
16.4.6 Fehler/Ausnahmen auslösen ... 799
16.4.7 Eigene Fehlerklassen ... 799
16.4.8 Exceptionhandling zur Debugzeit ... 801

17 Entity Framework Core ... 803
17.1 Überblick ... 803
17.1.1 Objektrelationaler Mapper (ORM) ... 804
17.1.2 Provider ... 806
17.1.3 Relationale Beziehungen ... 807
17.1.4 Benötigte NuGet-Pakete ... 810
17.2 CodeFirst ... 812
17.2.1 CodeFirst aus Model ... 812
17.2.2 CodeFirst mittels ReverseEngineering von bestehender Datenbank ... 822
17.3 Migrationen ... 827
17.3.1 Initiale Migration bei ReverseEngineering ... 827
17.3.2 Weitere Migrationen ... 828
17.4 Lesen und Schreiben von Daten mit EF Core ... 831
17.5 Neuerungen in Entity Framework Core 7 und 8 ... 835
17.5.1 JSON-Spalten ... 835
17.5.2 Bulk Updates ... 840
17.5.3 SaveChanges() ... 841
17.5.4 Mapping von Stored Procedures ... 841
17.5.5 Optimiertes SQL für Contains-Abfragen ... 842
17.5.6 Enums werden innerhalb von JSON-Spalten als ints gespeichert ... 842
17.5.7 Primitive Collections ... 842

17.6 Serialisierung mit System.Text.Json 845
17.7 Praxisbeispiele 850
17.7.1 Daten mit EF Core laden und als JSON speichern 850
17.7.2 Eine Datenbank mit EF Core anlegen und Testdaten generieren und anzeigen 857

Teil IV: Webanwendungen 865

18 Webanwendungen mit ASP.NET Core 867
18.1 Grundlagen 868
18.2 Razor Pages 871
18.2.1 Projektaufbau 872
18.2.2 Razor-Syntax 875
18.2.3 Layout-Vorlagen 881
18.2.4 Modelle für Razor Pages 884
18.2.5 Mit Formularen arbeiten 886
18.3 MVC 893
18.3.1 Projektaufbau 894
18.3.2 Action-Methoden 895
18.3.3 Zustandsmanagement 906
18.4 Praxisbeispiele 911
18.4.1 CRUD mit Entity Framework 911
18.4.2 Authentifizierung und Autorisierung 928
18.4.3 Zugriffsbeschränkung 935

19 Web APIs mit ASP.NET Core 939
19.1 REST 939
19.2 Vorlagen 943
19.3 Daten lesen 949
19.4 Daten schreiben, aktualisieren, löschen 958
19.5 Minimale APIs 968
19.6 Praxisbeispiele 973
19.6.1 Paginierung 973
19.6.2 XML statt JSON 975
19.6.3 CORS 976

20 Blazor 981
20.1 Hosting-Modelle 982
20.2 Projektvorlagen 985
20.3 Blazor-Komponenten 993
20.3.1 Code in/für Komponenten 993
20.3.2 Event-Handling 997
20.3.3 Datenbindung 1000

20.4 Services und APIs aufrufen ... 1003
20.5 Weitere Blazor-Features ... 1008
20.5.1 Zustandsmanagement ... 1008
20.5.2 JavaScript-Interoperabilität ... 1011
20.5.3 Render-Modi und Streaming ... 1015
20.6 Praxisbeispiele ... 1017
20.6.1 Online-Status ermitteln ... 1017
20.6.2 File-Uploads ... 1026
20.6.3 Fehlerbehandlung ... 1028
20.6.4 Datengrid ... 1033

Anhang A: Glossar ... 1035

Anhang B: Wichtige Dateiendungen ... 1039

Index ... 1041

Vorwort

C# ist die seit langem von Microsoft propagierte Sprache, sie bietet die Möglichkeiten und Flexibilität von C++ und erlaubt trotzdem eine schnelle und unkomplizierte Programmierpraxis wie einst bei Visual Basic. C# ist (fast) genauso mächtig wie C++, wurde aber komplett neu auf objektorientierter Basis geschrieben.

In Jürgens Fall traten 2001 seine Tochter Julia und C# mit der ersten Beta-Version in sein Leben ein. Beide begleiten ihn jetzt seit über 20 Jahren. Aus seiner kleinen Tochter wurde mittlerweile eine erwachsene Frau, die mitten in ihrem Medizinstudium steht. Die Liebe zur IT konnte Jürgen ihr leider nicht ganz vermitteln, und C# ist definitiv auch zu einer erwachsenen Programmiersprache geworden. C# hat es auch geschafft laut dem TIOBE-Index zur Programmiersprache des Jahres 2023 gekürt zu werden.

Mit .NET 8 hat Microsoft jetzt auch die neueste Version von .NET veröffentlicht, bei der vor allem viele Neuerungen im Bereich Performance und Stabilität (vor allem für MAUI) implementiert wurden. Da diese Versionsnummer – wie alle geraden – einen „Long Term Support" bietet, also drei Jahre lang Bugfixes und Sicherheitspatches, hat unser Buch eine gründliche Überarbeitung und Aktualisierung verdient. Der bewährte Aufbau bleibt natürlich erhalten, allerdings gibt es in jedem Kapitel Änderungen und Ergänzungen. Wir haben alle relevanten neuen Features von .NET 8 aufgenommen, beispielsweise die neuen Sprachfeatures von C# 12 oder die neuen Möglichkeiten von Blazor als Teil von ASP.NET Core 8. Sie erhalten somit weiterhin einen gründlichen Einstieg und Überblick über .NET, erfahren aber auch, was sich in den letzten Versionen getan hat.

C# ist das ideale Werkzeug zum Programmieren beliebiger Komponenten für .NET, beginnend bei WPF-, Web- und mobilen Anwendungen (auch für Android und iOS) bis hin zu systemnahen Applikationen.

Das vorliegende Buch ist ein Angebot für angehende wie auch für fortgeschrittene C#-Programmierer. Seine Philosophie knüpft an die vielen anderen Titel an, die in dieser Reihe in den vergangenen zwanzig Jahren zu verschiedenen Programmiersprachen erschienen sind:

- Programmieren lernt man nicht durch lineares Durcharbeiten eines Lehrbuchs, sondern nur durch unermüdliches Ausprobieren von Beispielen, verbunden mit ständigem Nachschlagen in der Referenz.
- Der Umfang einer modernen Sprache wie C# in Verbindung mit Visual Studio ist so gewaltig, dass ein seriöses Programmierbuch das Prinzip der Vollständigkeit aufgeben muss und nach dem Prinzip „so viel wie nötig" sich lediglich eine „Initialisierungsfunktion" auf die Fahnen schreiben kann.

Gegenüber anderen Büchern zur gleichen oder ähnlichen Thematik nimmt dieses Buch für sich in Anspruch, gleichzeitig Lehr- und Übungsbuch zu sein.

Zum Buchinhalt

Dieses Buch wagt den Spagat zwischen einem Grundlagen- und einem Profibuch. Sinn eines solchen Buches kann es nicht sein, eine umfassende Schritt-für-Schritt-Einführung in C# 12.0 zu liefern oder all die Informationen noch einmal zur Verfügung zu stellen, die Sie in der Produktdokumentation ohnehin schon finden und von denen Sie in der Regel nur ein Mausklick oder die F1-Taste trennt.

- Für den *Einsteiger* möchten wir den einzig vernünftigen und gangbaren Weg beschreiten, nämlich nach dem Prinzip „so viel wie nötig" eine schmale Schneise durch den Urwald der .NET-Programmierung mit C# schlagen, bis er eine Lichtung erreicht hat, die ihm erste Erfolgserlebnisse vermittelt.
- Für den *Profi* möchten wir in diesem Buch eine Vielzahl von Informationen und Know-how bereitstellen, wonach er bisher in den mitgelieferten Dokumentationen, im Internet bzw. in anderen Büchern vergeblich gesucht hat.

Die Kapitel des Buches sind in vier Themenkomplexe gruppiert; online gibt es noch umfangreiches Zusatzmaterial, das auch immer wieder ergänzt wird:

1. **Grundlagen der Programmierung mit C# (Buch)**
2. **Desktop-Anwendungen (Buch)**
3. **Technologien (Buch)**
4. **Webtechnologien (Buch)**
5. Windows-Forms-Anwendungen (online)
6. Zugriff auf das Dateisystem (online)
7. ADO.NET (online)
8. XML (online)

Die Kapitel innerhalb eines Teils bilden einerseits eine logische Abfolge, können andererseits aber auch quergelesen werden. Im Praxisteil eines jeden Kapitels werden anhand realer Problemstellungen die behandelten Programmiertechniken im Zusammenhang demonstriert.

Nobody is perfect

Sie werden in diesem Buch nicht alles finden, was C# bzw. .NET 6 zu bieten haben. Manches ist sicher in einem anderen Spezialtitel ausführlicher beschrieben. Aber Sie halten mit diesem Buch einen überschaubaren Breitband-Mix in den Händen, der sowohl vertikal vom Einsteiger bis zum Profi als auch horizontal von den einfachen Sprachelementen bis hin zu komplexen Anwendungen jedem etwas bietet, ohne dabei den Blick auf das Wesentliche im .NET-Dschungel zu verlieren.

Wenn Sie Vorschläge oder Fragen zum Buch haben, können Sie uns gerne kontaktieren:

juergen.kotz@primetime-software.de

info@christianwenz.de

Wir hoffen, dass wir Ihnen mit diesem Buch einen nützlichen Begleiter bei der .NET-Programmierung zur Seite gestellt haben, der es verdient, seinen Platz nicht im Regal, sondern griffbereit neben dem Computer einzunehmen.

Vielen Dank!

Ein Buch – gerade, wenn es so umfangreich ist – ist stets Teamarbeit. Herzlichen Dank an unsere Lektorin Sylvia Hasselbach, mit der wir seit über 20 Jahren zusammenarbeiten dürfen, sowie an Kristin Rothe und Irene Weilhart. Sollten trotz aller Sorgfalt Fehler auffallen, so veröffentlichen wir diese als Errata auf der Buchseite auf *www.hanser-fachbuch.de* und auf *https://plus.hanser-fachbuch.de* bei den Codebeispielen. Lenny Kotz hat uns bei einigen der Grafiken im Buch unterstützt.

Der besondere Dank von Jürgen geht noch an Prof. Dr. med. Franz Bader mit seinem Team vom Isarklinikum sowie an Herrn PD Dr. med. O. J. Stötzer, ohne deren Einsatz dieses Buch nicht pünktlich hätte erscheinen können.

Jürgen Kotz und Christian Wenz

München, im Januar 2024

Zusatzmaterial online

Der Einfachheit halber werden im ersten Teil die Beispiele weiter mit Windows Forms oder als Konsolenanwendungen erstellt. Der zweite Teil des Buches behandelt dann ausführlich WPF sowie .NET MAUI und im dritten Teil „Technologien“ werden die Beispiele dann mit WPF oder auch einer Konsolenanwendung dargestellt. Der vierte Teil ist dann der Webtechnologie vorbehalten. Alle Beispieldaten dieses Buches und die mittlerweile zahlreichen Bonuskapitel können Sie online herunterladen.

Geben Sie auf

https://plus.hanser-fachbuch.de

diesen Code ein:

`plus-W36m2-T9er4`

Beim Nachvollziehen der Buchbeispiele beachten Sie bitte Folgendes:

- Kopieren Sie die Buchbeispiele auf die Festplatte. Wenn Sie auf die Projektmappendatei (**.sln*) klicken, wird Visual Studio in der Regel automatisch geöffnet und das jeweilige Beispiel wird in die Entwicklungsumgebung geladen, wo Sie es z. B. mittels der **F5**-Taste kompilieren und starten können.
- Für einige Beispiele ist ein installierter Microsoft SQL Server Express LocalDB oder jegliche andere Instanz eines SQL-Servers erforderlich. Ersterer wird standardmäßig mit Visual Studio mit installiert.
- Beachten Sie die zu einigen Beispielen beigefügten *Liesmich.txt*-Dateien, die Sie auf besondere Probleme hinweisen.

Teil I: Grundlagen

- .NET 8 (Kapitel 1)
- Einstieg in Visual Studio 2022 Version 17.8 (Kapitel 2)
- Grundlagen der Sprache C# (Kapitel 3)
- OOP-Konzepte (Kapitel 4)
- Arrays, Strings, Funktionen (Kapitel 5)
- Weitere Sprachfeatures (Kapitel 6)
- Einführung in LINQ (Kapitel 7)
- Neuerungen von C# im Überblick (Kapitel 8)

1 .NET 8

Mit .NET 8 ist im November 2023 die aktuellste Version des Microsoft Frameworks, das mittlerweile auf eine über 20-jährige Geschichte zurückblickt, erschienen. Seit der Version .NET 6 handelt sich dabei wieder um eine Version, welche das gesamte .NET-Universum wieder vereint, es gibt somit wieder ein *„one" .NET*.

.NET 8 ist eine „Long Term Support"-Version (LTS) von .NET (weil gerade Versionsnummer) und wird somit bis zum November 2026 mit Fehlerbehebungen und Sicherheitsupdates unterstützt.

Mit dem .NET Framework können völlig unterschiedliche Applikationstypen (Desktop, Web, Mobile) in unterschiedlichen Programmiersprachen (C#, VB, F# etc.) entwickelt werden und egal was man macht, man hat eine einheitliche Funktionsbibliothek, eben das .NET Framework.

Im Jahre 2016 erschien parallel zum .NET Framework ein sogenanntes .NET Core Framework, das speziell für plattformübergreifende Webentwicklung konzipiert war. Keine Bindung somit mehr an Windows, sondern zeitgemäße plattformübergreifende Möglichkeiten (auch für mobile Plattformen wie iOs und Android), um seine Applikationen zu entwickeln und auch zu deployen. Da es sich bei .NET Core um Open Source handelte, hat Microsoft hier auch bewiesen, dass Open Source für Microsoft nicht nur ein Lippenbekenntnis ist, sondern ein wesentliches Fundament in der weiteren Entwicklung des Softwaregiganten.

Eigentlich war die Wiedervereinigung des klassischen .NET Frameworks mit den .NET-Core-Versionen bereits 2020 mit der Version 5.0 geplant, jedoch musste diese dann aufgrund der Coronapandemie um ein Jahr verschoben werden.

Somit wurden erst mit .NET 6 diese beiden Entwicklungsstränge (.NET und .NET Core) zusammengeführt und man hat wieder ein einheitliches Framework – laut Microsoft auch das performanteste .NET –, um seine Anwendungen zu schreiben. Nur die Entwicklung von Desktopapplikationen mit Windows Forms oder WPF ist noch an Windows gebunden, alle anderen Anwendungsmöglichkeiten sind plattformübergreifend.

Mit den seitdem erschienenen Versionen .NET 7 und nun eben auch .NET 8 hat Microsoft vor allem in Bezug auf Performance viele Besserungen durchgeführt.

Vor allem im Bereich der JSON-Serialisierung wurden im Namensraum *System.Text.Json* in der aktuellsten Version .NET 8 viele Serialisierungsverbesserungen eingearbeitet (mehr dazu in Kapitel 17).

1.1 Microsofts .NET-Technologie

Ganz ohne Theorie geht nichts! In diesem leider etwas „trockenen“ Abschnitt werden Einsteiger mit der grundlegenden .NET-Philosophie und den damit verbundenen Konzepten, Begriffen und Features vertraut gemacht. Dazu dürfen Sie Ihrem Rechner ruhig einmal eine Pause gönnen.

1.1.1 Zur Geschichte von .NET

Das Kürzel .NET ist die Bezeichnung für eine gemeinsame Plattform für viele Programmiersprachen. Beim Kompilieren von .NET-Programmen wird der jeweilige Quelltext in MSIL (*Microsoft Intermediate Language*) übersetzt. Es gibt nur ein gemeinsames Laufzeitsystem für alle Sprachen, die sogenannte CLR (Common Language Runtime), die die MSIL-Programme ausführt.

Die im Jahr 2002 eingeführte .NET-Technologie wurde notwendig, weil sich die Anforderungen an moderne Softwareentwicklung dramatisch verändert hatten, wobei das rasant wachsende Internet mit seinen hohen Ansprüchen an die Skalierbarkeit einer Anwendung, die Verteilbarkeit auf mehrere Schichten und ausreichende Sicherheit der hauptsächliche Motor war, sich nach einer grundlegend neuen Sprachkonzeption umzuschauen.

Mit .NET fand ein radikaler Umbruch in der Geschichte der Softwareentwicklung bei Microsoft statt. Nicht nur dass jetzt „echte“ objektorientierte Programmierung zum obersten Dogma erhoben wird, nein, auch eine langjährig bewährte Sprache wie das alte Visual Basic wurde völlig umgekrempelt und die einst gelobte COM-Technologie wurde zum Auslaufmodell erklärt!

HINWEIS: Dem unerfahrenen Leser mögen einige Ausführungen im Laufe dieses Kapitels noch sehr fremd sein. Machen Sie sich deswegen keine Sorgen und wenn Sie etwas nicht verstehen, kommen Sie einfach am Ende des Buches nochmals hierher zurück.

Warum eine eigene Programmiersprache?

C# wurde ausschließlich für das .NET Framework konzipiert, wobei versucht wurde, das Beste aus den etablierten Programmiersprachen Java, JavaScript, Visual Basic und C++ zu kombinieren, ohne aber deren Nachteile zu übernehmen.

Da sich die etablierten Sprachen nicht ohne größere Kompromisse an das .NET Framework anpassen ließen, haben die .NET-Entwickler die Gelegenheit beim Schopf gepackt und eine „maßgeschneiderte“ .NET-Sprache entwickelt. C# setzt auf dem .NET Framework auf und kommt ohne „faule“ Kompatibilitätskompromisse aus, wie sie z. B. teilweise bei Visual Basic erforderlich waren.

Als konsequent objektorientierte Sprache erfüllt C# folgende Kriterien:

- **Abstraktion:** Die Komplexität eines Geschäftsproblems ist beherrschbar, indem eine Menge von abstrakten Objekten identifiziert werden können, die mit dem Problem verknüpft sind.
- **Kapselung:** Die interne Implementation einer solchen Abstraktion wird innerhalb des Objekts versteckt.
- **Polymorphie:** Ein und dieselbe Methode kann auf mehrere Arten implementiert werden.
- **Vererbung:** Es wird nicht nur die Schnittstelle, sondern auch der Code einer Klasse vererbt (Implementationsvererbung statt der COM-basierten Schnittstellenvererbung).

Microsoft konnte natürlich nicht über Nacht die COM-Technologie auf die Müllkippe entsorgen, denn zu viele Programmierer wären dadurch auf immer und ewig verprellt worden und hätten sich frustriert einer stabileren Entwicklungsplattform zugewandt. Aus diesem Grund hat COM auch in .NET noch einige Zeit sein Gnadenbrot erhalten.

HINWEIS: Das gesamte Office-Paket wird auch weiterhin mit COM programmiert und die Schnittstellen zur Office-Automatisierung stehen über COM, das aus .NET heraus aufgerufen werden kann, zur Verfügung. Mittlerweile gibt es sogenannte PIA (Primary Interop-Assemblies)-Schnittstellen, die einen Wrapper für die COM-Schnittstelle bieten.

Ja, und das Ganze funktioniert auch weiterhin mit .NET 8.

Wie funktioniert eine .NET-Sprache?

Jeder in einer beliebigen .NET-Programmiersprache geschriebene Code wird beim Kompilieren in einen Zwischencode, den sogenannten MSIL-Code (*Microsoft Intermediate Language Code*), übersetzt, der unabhängig von der Plattform bzw. der verwendeten Hardware ist und dem man es auch nicht mehr ansieht, in welcher Sprache seine Source geschrieben wurde.

HINWEIS: Es gibt über 50 verschiedene Programmiersprachen für .NET. Sogar das ehrwürdige COBOL besitzt eine Version für .NET!

Erst wenn der MSIL-Code von einem Programm zur Ausführung genutzt werden soll, wird er vom *Just-in-Time(JIT)-Compiler* in Maschinencode übersetzt[1]. Ein .NET-Programm wird also vom Entwurf bis zu seiner Ausführung auf dem Zielrechner tatsächlich zweimal kompiliert (siehe Bild 1.1).

HINWEIS: Für die Installation eines Programms ist in der Regel lediglich die Weitergabe des MSIL-Codes erforderlich. Voraussetzung ist allerdings das Vorhandensein der .NET-Laufzeitumgebung (CLR), die Teil des .NET Frameworks ist, auf dem Zielrechner.

[1] Der Begriff „jeder Code“ schließt z. B. auch den Code der ASP.NET-Seiten ein.

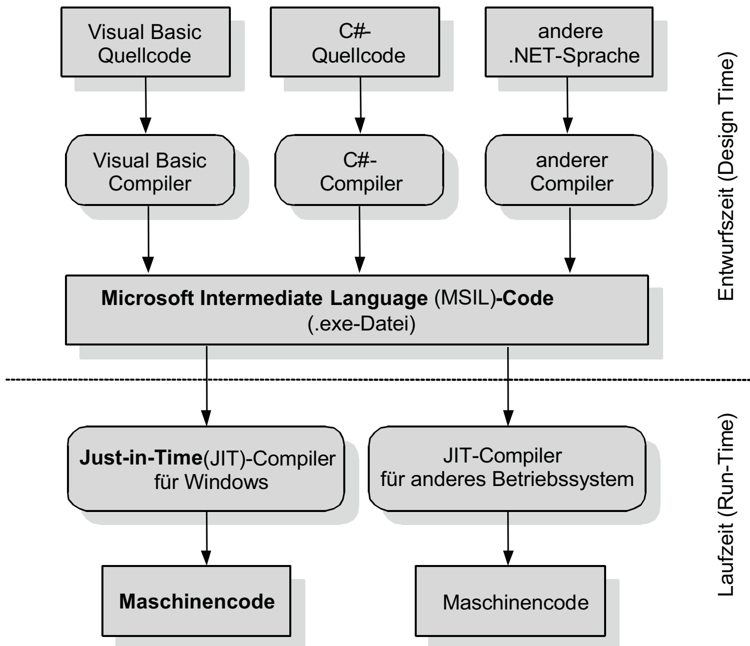


Bild 1.1 Übersetzungsschritte einer .NET Programmiersprache

Eine der wesentlichsten Änderungen in .NET 7 mit einer Weiterführung in .NET 8 ist die Einführung eines *AoT*-Compilers (Ahead of Time). Bislang gab es in der .NET-Welt nur *JiT*-Compiler (Just in Time), die den erzeugten IL-Code (*Intermediate Language*-Code) während der Ausführung des Programms in nativen Maschinencode übersetzten. Bislang war es nur mit den Tools *ngen.exe* (klassisches .NET) bzw. *crossgen.exe* (.NET Core) möglich, nativen Maschinencode zu erstellen.

Nun hat man tatsächlich auch die Möglichkeit, bereits zur Compilezeit nativen Maschinencode zu erstellen.

Bei der Projektanlage (zumindest bei ausgewählten Projekttypen)) kann man bereits die Option *native AOT-Veröffentlichung aktivieren anhaken* (siehe Bild 1.2), und dadurch wird der folgende zusätzliche Eintrag in die Projektdatei geschrieben:

```
<PropertyGroup>
    <PublishAot>true</PublishAot>
</PropertyGroup>
```

Dadurch aktiviert man die Option für die *Ahead-of-Time* Kompilierung.

Weitere Informationen

Konsolen-App C# Linux macOS Windows Konsole

Framework

.NET 8.0 (Langfristiger Support)

☑ Keine Anweisungen der obersten Ebene verwenden

☑ native AOT-Veröffentlichung aktivieren

Bild 1.2 Neue AOT-Veröffentlichungsoption bei der Projektanlage

Wenn das Projekt dann veröffentlicht wird, erhält man nun die binären Dateien in nativem Maschinencode.

Leider ist diese Option noch nicht für alle Projektarten verfügbar, aber zumindest ein Anfang ist gemacht. Zum jetzigen Zeitpunkt ist AoT-Kompilierung nur für Konsolenanwendungen (und dabei auch nicht für alle Bibliotheken) verfügbar.

HINWEIS: In dem Zusammenhang sei erwähnt, dass C# laut dem TIOBE-Index (*https://www.tiobe.com/tiobe-index/*) die Programmiersprache des Jahres 2023 ist. Nach über zwei Jahrzehnten in den Top 10 in der Liste hat C# nun zum ersten Mal den Sprung an die Spitze geschafft.[2]

1.1.2 .NET-Features und Begriffe

Mit Einführung von Microsofts .NET-Technologie prasselte auch eine Vielzahl neuer Begriffe auf die Entwicklergemeinde ein. Es werden hier nur die wichtigsten erklärt.

.NET Framework

.NET ist die Infrastruktur für die gesamte .NET-Plattform. Es handelt sich hierbei gleichermaßen um eine Entwurfs- wie um eine Laufzeitumgebung, in der Windows- und Web-Anwendungen erstellt und verteilt werden können.

Die nachfolgende Abbildung versucht, einen groben Überblick über die Komponenten des .NET Frameworks zu geben.

Zu den wichtigsten Komponenten des .NET Frameworks und den damit zusammenhängenden Begriffen zählen:

- Common Language Specification (CLS),
- Common Type System (CTS),

[2] Der TIOBE-Index ist ein Indikator über die Beliebtheit einer Programmiersprache und wird jeden Monat ermittelt. In der Berechnung des Ratings spielen vor allem die Anzahl von Suchen in Suchmaschinen, Angebote an Schulungen, Anzahl von Drittherstellern für die Sprache sowie Anzahl der Entwickler eine Rolle. Die Programmiersprache des Jahres ist dabei die Sprache mit dem größten Ratingzuwachs.

- Common Language Runtime (CLR),
- .NET-Klassenbibliothek,
- diverse Basisklassenbibliotheken wie ADO.NET und ASP.NET,
- diverse Compiler z. B. für C#, VB.NET usw.

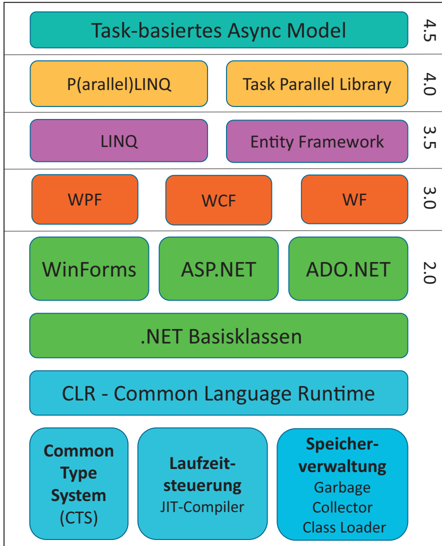


Bild 1.3
Komponenten des .NET Frameworks

In der obigen Abbildung sind die Entwicklungsstufen des ursprünglichen .NET Frameworks aufgezeichnet, die letzte klassische .NET Version 4.8 wurde 2019 veröffentlicht.

Im Folgenden sollen die einzelnen .NET-Bestandteile einer näheren Betrachtung unterzogen werden.

Die Common Language Specification (CLS)

Um den sprachunabhängigen MSIL-Zwischencode erzeugen zu können, müssen allgemein gültige Richtlinien und Standards für die .NET-Programmiersprachen existieren. Diese werden durch die *Common Language Specification* (CLS) definiert, die eine Reihe von Eigenschaften festlegt, die jede .NET-Programmiersprache erfüllen muss.

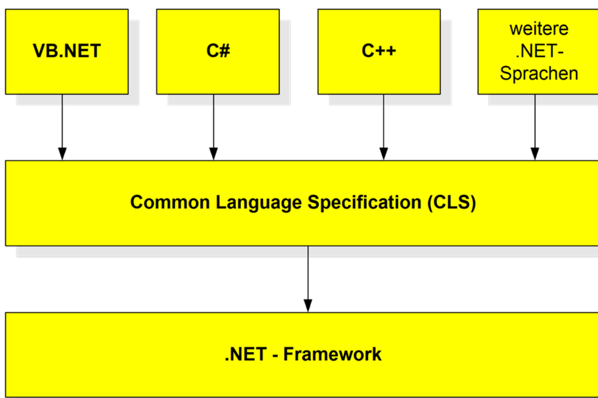


Bild 1.4
CLS – Common Language Specification

HINWEIS: Ganz egal, mit welcher .NET-Programmiersprache Sie arbeiten, der Quellcode wird immer in ein und dieselbe Intermediate Language (MSIL) kompiliert.

Besonders für die Entwicklung von .NET-Anwendungen im Team haben die Standards der CLS weitreichende positive Konsequenzen, denn es ist nun zweitrangig, in welcher .NET-Programmiersprache Herr Müller die Komponente X und Herr Meier die Komponente Y schreibt. Alle Komponenten werden problemlos miteinander interagieren!

Auf einen wichtigen Bestandteil des CLS kommen wir im folgenden Abschnitt zu sprechen.

Das Common Type System (CTS)

Ein Kernbestandteil der CLS ist das *Common Type System (CTS)*. Es definiert alle Typen[3], die von der .NET-Laufzeitumgebung (CLR) unterstützt werden.

Alle diese Typen lassen sich in zwei Kategorien aufteilen:

- Wertetypen (werden auf dem Stack im Speicher abgelegt)
- Referenztypen (werden auf dem Heap im Speicher abgelegt)

Zu den Wertetypen gehören beispielsweise die ganzzahligen Datentypen und die Gleitkommazahlen. Zu den Referenztypen zählen die Objekte, die aus Klassen instanziiert wurden.

HINWEIS: Dass unter .NET auch die Wertetypen letztendlich als Objekte betrachtet und behandelt werden, liegt an einem als *Boxing* bezeichneten Verfahren, das die Umwandlung eines Wertetyps in einen Referenztyp zur Laufzeit besorgt.

Warum hat Microsoft mit dem heiligen Prinzip der Abwärtskompatibilität gebrochen und selbst die fundamentalen Datentypen einer Programmiersprache neu definiert?

Als Antwort kommen wir noch einmal auf eine wesentliche Säule der .NET-Philosophie zu sprechen, auf die durch CLS/CTS manifestierte Sprachunabhängigkeit und auf die Konsequenzen, die dieses Paradigma nach sich zieht.

Microsofts .NET-Entwickler hatten gar keine andere Wahl, denn um Probleme beim Zugriff auf sprachfremde Komponenten zu vermeiden und um eine sprachübergreifende Programmentwicklung überhaupt zu ermöglichen, mussten die Spezifikationen der Programmiersprachen durch die CLS einander angepasst werden. Dazu müssen alle wesentlichen sprachbeschreibenden Elemente – wie vor allem die Datentypen – in allen .NET-Programmiersprachen gleich sein.

Da .NET eine Normierung der Programmiersprachen erzwingt, verwischen die Grenzen zwischen den verschiedenen Sprachen und Sie brauchen nicht immer umzudenken, wenn Sie tatsächlich einmal auf eine andere .NET-Programmiersprache umsteigen wollen.

[3] Unter .NET spricht man allgemein von Typen und meint damit Klassen, Interfaces und Datentypen, die als Wert übergeben werden.

Als Lohn für die Mühen und den Mut, die eingefahrenen Gleise seiner altvertrauten Sprache zu verlassen, winken dem Entwickler wesentliche Vereinfachungen. So sind die Zeiten des alltäglichen Ärgers mit den Datentypen – wie z. B. bei der Übergabe eines Integers an eine C-DLL – endgültig vorbei. Bei so viel Licht gibt es natürlich auch Schatten. Mit einem der inzwischen zahlreich und kostenlos verfügbaren MSIL-Decompiler ist es sehr einfach möglich, aus einer EXE-Datei den Quellcode zu generieren. Wenn überhaupt, so ist es nur mit hohem Aufwand möglich, dass Sie als Programmierer Ihr Know-how vor der Konkurrenz schützen können. In Zeiten von Open Source sehen manche Unternehmen dieses Problem jedoch nicht mehr, anderen ist jedoch der Schutz des eigenen Quellcodes noch immer wichtig.

Die Common Language Runtime (CLR)

Die Laufzeitumgebung bzw. *Common Language Runtime* (CLR) ist die Umgebung, in der .NET-Programme auf dem Zielrechner ausgeführt werden. Sie muss auf einem Computer nur einmal installiert sein und schon laufen sämtliche .NET-Anwendungen, egal ob sie in C#, VB oder F# programmiert wurden. Die CLR zeichnet für die Ausführung der Anwendungen verantwortlich und kooperiert auf Basis des CTS mit der MSIL.

Mit ihren Fähigkeiten bildet die Common Language Runtime gewissermaßen den Kern von .NET. Den Code, der von der CLR ausgeführt wird, bezeichnet man auch als verwalteten bzw. *Managed Code*.

Die CLR ist innerhalb des .NET Frameworks nicht nur für das Ausführen von verwaltetem Code zuständig. Der Aufgabenbereich der CLR ist weitaus umfangreicher und umfasst zahlreiche Dienste, die als Bindeglied zwischen dem verwalteten MSIL-Code und dem Betriebssystem des Rechners die Anforderungen des .NET Frameworks sicherstellen, wie z. B.

- ClassLoader,
- Just-in-Time(JIT)-Compiler,
- ExceptionManager,
- Code Manager,
- Security Engine,
- Debug Machine,
- Thread Service,
- COM-Marshaller.

Die Verwendung der sprachneutralen MSIL erlaubt die Nutzung des CTS und der Basisklassen für alle .NET-Sprachen gleichermaßen. Einziger hardwareabhängiger Bestandteil des .NET Frameworks ist der Just-in-Time-Compiler. Deshalb kann der MSIL-Code im Prinzip frei zwischen allen Plattformen bzw. Geräten, für die ein .NET Framework existiert, ausgetauscht werden.

Namespaces

Alle Typen des .NET Frameworks werden in sogenannten Namensräumen (Namespaces) zusammengefasst. Unabhängig von irgendeiner Klassenhierarchie wird jede Klasse einem bestimmten Namensraum zugeordnet.

Die folgende Tabelle zeigt beispielhaft einige wichtige Namespaces für die Basisklassen des .NET Frameworks:

Namespace	. . . enthält Klassen für . . .
System.Windows.Forms	. . . Windows-basierte WinForms-Anwendungen
System.Collections	. . . Objekt-Arrays
System.Drawing	. . . die Grafikprogrammierung
System.Data	. . . den Datenbankzugriff
System.Web	. . . die Webprogrammierung
System.IO	. . . Ein- und Ausgabeoperationen

Mit den Namespaces hat auch der Ärger mit der Registrierung von (COM-)Komponenten bei Versionskonflikten sein Ende gefunden, denn eine unter .NET geschriebene Komponente wird von der .NET-Runtime nicht mehr über die ProgID der Klasse mithilfe der Registry lokalisiert, sondern über einen in der Runtime enthaltenen Mechanismus, der einen Namespace einer angeforderten Komponente sowie deren Version für das Heranziehen der „richtigen“ Komponente verwendet.

Assemblierungen

Unter einer Assemblierung (*Assembly*) versteht man eine Basiseinheit für die Verwaltung von Managed Code und für das Verteilen von Anwendungen. Sie kann sowohl aus einer einzelnen als auch aus mehreren Dateien (Modulen) bestehen. Eine solche Datei (*.dll* oder *.exe*) enthält MSIL-Code (kompilierter Zwischencode).

Die Klassenverwaltung in Form von selbst beschreibenden Assemblies vermeidet Versionskonflikte von Komponenten und ermöglicht vor allem dynamische Programminstallationen aus dem Internet. Statt der bei einer klassischen Installation bisher erforderlichen Einträge in die Windows-Registry genügt nunmehr einfaches Kopieren der Anwendung.

Normalerweise müssen Sie die Assemblierungen referenzieren, in denen die von Ihrem Programm verwendeten Typen bzw. Klassen enthalten sind. Eine Ausnahme ist die Assemblierung *mscorlib.dll*, welche die Basistypen des .NET Frameworks in verschiedenen Namensräumen umfasst (siehe obige Tabelle).

Zugriff auf COM-Komponenten

Verweise auf COM-DLLs werden so eingebunden, dass sie zur Entwurfszeit quasi wie .NET-Komponenten behandelt werden können.

Über das Menü *Projekt/COM-Verweis hinzufügen* erreichen Sie unter Visual Studio die Liste der verfügbaren COM-Bibliotheken. Nachdem Sie die gewünschte Bibliothek selektiert haben, können Sie die COM-Komponente wie gewohnt ansprechen.

HINWEIS: Wenn Sie COM-Objekte, wie z. B. alte ADO-Bibliotheken, in Ihre .NET-Projekte einbinden wollen, müssen Sie auf viele Vorteile von .NET verzichten. Durch die zusätzliche Interoperabilitätsschicht sinkt die Performance meist deutlich.

Metadaten und Reflexion

Das .NET Framework stellt im *System.Reflection*-Namespace einige Klassen bereit, die es erlauben, die Metadaten (Beschreibung bzw. Strukturinformationen) einer Assembly zur Laufzeit auszuwerten, womit z. B. eine Untersuchung aller dort enthaltenen Typen oder Methoden möglich ist.

Die Beschreibung durch die .NET-Metadaten ist allerdings wesentlich umfassender, als es in den gewohnten COM-Typbibliotheken üblich war. Außerdem werden die Metadaten direkt in der Assembly untergebracht, die dadurch selbstbeschreibend wird und z. B. auf Registry-Einträge verzichten kann. Metadaten können daher nicht versehentlich verloren gehen oder mit einer falschen Dateiversion kombiniert werden.

HINWEIS: Es gibt unter .NET nur noch eine einzige Stelle, an der sowohl der Programmcode als auch seine Beschreibung gespeichert werden!

Metadaten ermöglichen es, zur Laufzeit festzustellen, welche Typen benutzt und welche Methoden aufgerufen werden. Daher kann .NET die Umgebung an die Anwendung anpassen, sodass diese effizienter arbeitet.

Der Mechanismus zur Abfrage der Metadaten wird Reflexion (*Reflection*) genannt. Das .NET Framework bietet dazu eine ganze Reihe von Methoden an, mit denen jede Anwendung – nicht nur die CLR – die Metadaten von anderen Anwendungen abfragen kann.

Auch Entwicklungswerkzeuge wie Visual Studio verwenden die Reflexion, um z. B. den Mechanismus der IntelliSense zu implementieren. Sobald Sie einen Methodennamen eintippen, zeigt die IntelliSense eine Liste mit den Parametern der Methode an oder mit allen Elementen eines bestimmten Typs.

Weitere nützliche Werkzeuge, die auf der Basis von Reflexionsmethoden arbeiten, sind der IL-Disassembler (ILDASM) des .NET Frameworks oder ILSpy.

HINWEIS: Eine besondere Bedeutung hat Reflexion im Zusammenhang mit dem Auswerten von Attributen zur Laufzeit (siehe den folgenden Abschnitt).

Attribute

In vielen anderen Sprachen (VB 6, Delphi 7 etc.) versteht man Attribute als Variablen, die zu einem Objekt gehören und damit seinen Zustand beschreiben.

Unter .NET haben Attribute eine grundsätzlich andere Bedeutung: .NET-Attribute stellen einen Mechanismus dar, mit dem man Typen und Elemente einer Klasse schon beim Entwurf kommentieren und mit Informationen versorgen kann, die sich zur Laufzeit mittels Reflexion abfragen lassen.

Auf diese Weise können Sie eigenständige, selbstbeschreibende Komponenten entwickeln, ohne die erforderlichen Infos separat in Ressourcendateien oder Konstanten unterbringen zu müssen. So erhalten Sie mobilere Komponenten mit besserer Wartbarkeit und Erweiterbarkeit.

Man kann Attribute auch mit „Anmerkungen“ vergleichen, die man einzelnen Quellcode-Elementen, wie Klassen oder Methoden, „anheftet“. Solche Attribute (Annotations) gibt es

eigentlich in jeder Programmiersprache, sie regeln z. B. die Sichtbarkeit eines bestimmten Datentyps. Allerdings waren diese Fähigkeiten bislang fest in den Compiler integriert, während sie unter .NET nunmehr direkt im Quellcode zugänglich sind. Das heißt, dass .NET-Attribute typsichere, erweiterbare Metadaten sind, die zur Laufzeit von der CLR (oder von beliebigen .NET-Anwendungen) ausgewertet werden können.

Mit Attributen können Sie Design-Informationen (z. B. zur Dokumentation), Laufzeit-Infos (z. B. Namen einer Datenbankspalte für ein Feld) oder sogar Verhaltensvorschriften für die Laufzeit (z. B. ob ein gegebenes Feld an einer Transaktion teilnehmen darf) definieren. Die Möglichkeiten sind quasi unbegrenzt.

Wenn Ihre Anwendung beispielsweise einen Teil der erforderlichen Informationen in der Registry abspeichert, muss bereits beim Entwurf festgelegt werden, wo die Registrierschlüssel abzulegen sind. Solche Informationen werden üblicherweise in Konstanten oder in einer Ressourcendatei untergebracht oder sogar fest in die Aufrufe der entsprechenden Registrierfunktionen eingebaut. Wesentliche Bestandteile der Klasse werden also von der übrigen Klassendefinition abgetrennt. Der Attribute-Mechanismus macht damit Schluss, denn er erlaubt es, derlei Informationen direkt an die Klassenelemente „anzuheften“, sodass letztendlich eine sich vollständig selbst beschreibende Komponente vorliegt.

Serialisierung

Fester Bestandteil des .NET Frameworks ist auch ein Mechanismus zur Serialisierung von Objekten. Unter Serialisierung versteht man das Umwandeln einer Objektinstanz in sequenzielle Daten, d. h. in binäre, XML- oder JSON-Daten mit dem Ziel, die Objekte als Datei permanent zu speichern oder über Netzwerke zu verschicken.

Auf umgekehrtem Weg rekonstruiert die Deserialisierung aus den Daten wieder die ursprüngliche Objektinstanz.

Das .NET Framework unterstützt drei verschiedene Serialisierungsmechanismen:

- Die Shallow-Serialisierung mit der Klasse *System.Xml.Serialization.XmlSerializer*.
- Die Deep-Serialisierung mit den Klassen *BinaryFormatter* und *SoapFormatter* aus dem *System.Runtime.Serialization*-Namespace.
- JSON-Serialisierung mittels der Klassen im Namensraum *System.Text.Json*
- oder alternativ (vor allem für die klassischen .NET Frameworks) zu *System.Text.Json*
- Durch das ladbare NuGet-Paket *Newtonsoft.Json* wird auch eine JSON-Serialisierung ermöglicht.

Aufgrund ihrer Einschränkungen (geschützte und private Objektfelder bleiben unberücksichtigt) ist die Shallow-Serialisierung für uns weniger interessant. Hingegen werden bei der Deep-Serialisierung alle Felder berücksichtigt – Bedingung ist lediglich die Kennzeichnung der Klasse mit dem Attribut *[Serializable]*.

Das populärste Austauschformat momentan ist jedoch JSON. Die Funktionalität zur Serialisierung in dieses Format wird seit .NET 5 über eigene .NET-Framework-Klasse zur Verfügung gestellt, in vielen Teilen und vor allem bei den klassischen Framework-Versionen wird aber auch noch das Nuget-Paket *Newtonsoft.Json* verwendet.

Anwendungsgebiete der Serialisierung finden sich bei ASP.NET, ADO.NET, XML etc.

Multithreading

Multithreading ermöglicht es einer Anwendung, ihre Aktivitäten so aufzuteilen, dass diese unabhängig voneinander ausgeführt werden können, bei gleichzeitig besserer Auslastung der Prozessorzeit. Allgemein sind Threads keine Besonderheit von .NET, sondern auch in anderen Programmierumgebungen durchaus üblich.

Unter .NET laufen Threads in einem Umfeld, das Anwendungsdomäne genannt wird. Erstellung und Einsatz erfolgen mithilfe der Klasse *System.Threading.Thread*.

Nicht in jedem Fall ist die Aufnahme zusätzlicher Threads die beste Lösung, da man sich dadurch auch zusätzliche Probleme einhandeln kann. So ist beim Umgang mit mehreren Threads die Threadsicherheit von größter Bedeutung. Das heißt, aus Sicht der Threads müssen die Objekte stets in einem gültigen Zustand vorliegen und das auch dann, wenn sie von mehreren Threads gleichzeitig benutzt werden.

Ab der .NET-Framework-Version 4.0 wurde das Konzept von *Tasks* (eine Art Hintergrundthread) mittels der *TPL* (Task Parallel Library) hinzugefügt. Mittels Tasks soll die Programmierung von asynchronen Funktionsaufrufen wesentlich vereinfacht werden. Die Klassen der TPL befinden sich im Namensraum *System.Threading.Tasks*.

Objektorientierte Programmierung

Last, but not least, möchten wir am Ende unseres Rundflugs über die .NET-Highlights noch einmal auf das allem zugrunde liegende OOP-Konzept verweisen, denn .NET ist komplett objektorientiert aufgebaut – unabhängig von der verwendeten Sprache oder der Zielumgebung, für die programmiert wird (Desktop- oder Web-Anwendung).

Jeder .NET-Code ist innerhalb einer Klasse verborgen und sogar einfache Variablen sind zu Objekten mutiert, die Eigenschaften und Methoden bereitstellen. Es ist deshalb wenig sinnvoll, mit der Einführung in die Sprache C# fortzufahren, ohne sich vorher mit dem Konzept der OOP vertraut gemacht zu haben (siehe Kapitel 4).

1.2 .NET Core

Um die Konkurrenz nicht enteilen zu lassen, entschloss sich Microsoft neben dem klassischen .NET Framework das .NET Core Framework als Open-Source-Projekt zu entwickeln. Mit diesem Framework war es möglich, plattformübergreifende Webapplikationen und auch mobile Applikationen zu entwickeln.

1.2.1 Geschichte von .NET Core

.NET Core war eine parallele Entwicklung, vornehmlich für Webapplikationen, die plattformübergreifend eingesetzt werden konnte und nicht mehr an das Windows-Betriebssystem gebunden war.

Bereits Mitte 2014 kündigte Microsoft .NET Core an, es dauerte dann aber noch zwei Jahre, bis .NET Core 1.0 final erschien. Der große Unterschied zum bisherigen klassischen .NET Framework bestand darin, dass nicht mehr auf das gesamte Framework referenziert wurde, sondern alle Bibliotheken in NuGet-Pakete aufgedröselt wurden, die dann – je nach dem, was im Projekt gebraucht wurde – diesem hinzugefügt wurden. Der Funktionsumfang dieser ersten Version enthielt natürlich nur ein Bruchteil des Funktionsumfangs des klassischen .NET Frameworks.

In zyklischen Abständen kamen dann weitere Versionen von 1.1 über 2.0, 2.1 sowie 2.2, und zu guter Letzt 3.0 und im Dezember 2019 .NET Core 3.1 hinzu, um die Lücken in der Funktionalität des bisherigen .NET Frameworks einigermaßen zu schließen. Alle Versionen wurden als .NET Core bezeichnet.

Dabei wurde der Support dieser Versionen in zwei Kategorien eingeteilt:

- LTS (Long Term Support) für 3 Jahre
- Current für 18 Monate

Interessant dabei war, dass alle Versionen mit einer ungeraden Minor-Versionsummer (1.1, 2.1, 3.1) Long Term Support bekamen, während die anderen Versionen nur für 18 Monate von Microsoft supportet wurden.

Im November 2020 erschien dann .NET 5, jedoch nicht mit all den Features, die Microsoft geplant hatte. Einige Features wurden aufgrund der Coronapandemie auf die Version 6 verschoben. Obwohl .NET 5 der Nachfolger von .NET Core 3.1 war, wurde auf die Benennung .NET (Core) 4 verzichtet.

Zum einen wollte man keine Verwechslung mit der bisherigen klassischen Version des .NET Frameworks aufkommen lassen, das ja mittlerweile auch die Version 4 (zu diesem Zeitpunkt 4.8) erreicht hat. Und um klar darzustellen, dass es sich um die Basis aller weiteren .NET Frameworks handelt, wurde auch die Bezeichnung Core fallen gelassen.

Auch im klassischen .NET Framework wurde damit begonnen, wesentliche Bestandteile aus dem Framework auszulagern, um schnellere Produktzyklen gewährleisten zu können. Das bekannteste Beispiel dazu ist vermutlich Entity Framework, die von Microsoft empfohlene Datenbankzugriffstechnologie. Genauso wurde auch eine Core-Variante des Entity Frameworks entwickelt, die in Kapitel 17 beschrieben ist. Während bei der Framework-Version . NET 6 auf den Zusatz Core verzichtet wurde, werden wir diesen bei Entity Framework, zumindest zum jetzigen Stand, noch finden.

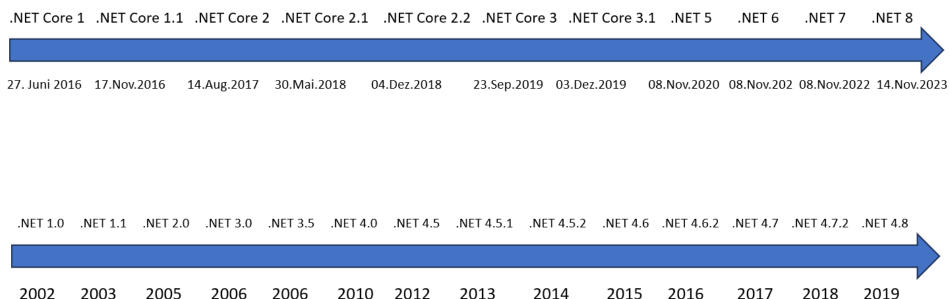


Bild 1.5 Entwicklung der Frameworks

1.2.2 LTS – Long Term Support und zukünftige Versionen

Microsoft hat mit Erscheinen von .NET 5 angekündigt, dass nunmehr jedes Jahr im November eine neue .NET-Version erscheint. Hierbei werden alle Versionen mit einer geraden Versionsnummer (wie .NET 6 oder .NET 8) Long Term Support erhalten, während der Support für die ungeraden Versionen wieder auf 18 Monate beschränkt ist.

Bild 1.6 Geplanter .NET-Versionsverlauf

1.2.3 .NET Standard

Dadurch, dass es jetzt plötzlich zwei nicht kompatible Framework-Versionen gab (klassisches .NET und .NET Core) hatten viele Entwickler folgendes Problem: Man wollte neue Komponenten mit der neuen Technologie entwickeln, aber natürlich konnte man die Codebase nicht von heute auf morgen einfach umstellen.

Eine .NET-Bibliothek konnte nicht in einem .NET Core-Projekt referenziert werden und umgekehrt, somit hatte man keine Möglichkeit, Bibliotheken zu entwickeln, die für beide Plattformen genutzt werden konnten.

Zu diesem Zweck wurde .NET Standard eingeführt. .NET Standard ist nicht noch ein zusätzliches Framework, sondern eine Spezifikation. In dieser Spezifikation wurde eine Reihe von Schnittstellen definiert, die in allen .NET-Plattformen implementiert werden müssen. Somit war .NET Standard der kleinste gemeinsame Nenner zwischen .NET, .NET Core und Xamarin [4](für die Entwicklung von mobilen Anwendungen unter iOs und Android).

Bibliotheken, die sowohl in .NET-Projekten wie auch in .NET-Core-Projekten genutzt werden sollten, konnte man somit in .NET Standard entwickeln.

Auch für .NET Standard gab es verschiedene Versionen, die immer mehr Schnittstellen definierten. Beginnend bei der Version 1.0, die noch sehr wenige Schnittstellen geboten hat, über die Version 1.6 hin bis zu 2.0 und 2.1.

Unterstützte die Version 1.6 noch ca. 13 000 Schnittstellen, so waren es bei der Version 2.0 schon 32 000 Schnittstellen, die man benutzen konnte. Allerdings wird für die .NET-Standard-2.0-Version auch bereits mindestens die Verwendung des .NET Frameworks in der Version 4.7.2 empfohlen (obwohl 4.6.1 offiziell unterstützt wird).

HINWEIS: Wir hoffen Sie mit diesen unzähligen unterschiedlichen Versionen und Versionsnummern nicht abgeschreckt zu haben, und können Sie beruhigen, dass wir mit der Aufzählung der unterschiedlichen Versionen durch sind.

[4] Xamarin wird in .NET 6 durch .NET MAUI ersetzt. MAUI (Multiplatform App UI) wurde jedoch zur Veröffentlichung von .NET 6 nicht fertiggestellt und wird für das 2. Quartal 2022 angekündigt.

1.3 Features von .NET 6

So, dann wollen wir uns noch anschauen, was uns die weiterentwickelten .NET Frameworks an neuen Features bieten.

In puncto Desktopapplikationen hat man unterschiedlichste Frameworks zum Entwickeln von herkömmlichen, aber auch mobilen Apps. Wie schon in den Vorgängerversionen kann man WPF-Applikationen (siehe Kapitel 9-12) oder auch Windows-Forms-Applikationen (siehe Onlinekapitel) implementieren. Diese beiden Anwendungsarten sind aber weiterhin an die Windows-Plattform gebunden.

Komplett neu ist jedoch .NET MAUI (Multiplatform App UI), das als Nachfolger von Xamarin die Möglichkeit bietet, mobile Applikationen für iOS und Android zu entwickeln (siehe Kapitel 13). Eine Codebasis für die unterschiedlichen Plattformen nimmt dem Entwickler viel Arbeit ab.

Und dann kommen natürlich noch die Anwendungsarten für die Webentwicklung hinzu. Dieses sind die Kapitel ASP.NET Core (Kapitel 18), ASP.NET Web API (Kapitel 19) sowie Blazor (Kapitel 20).

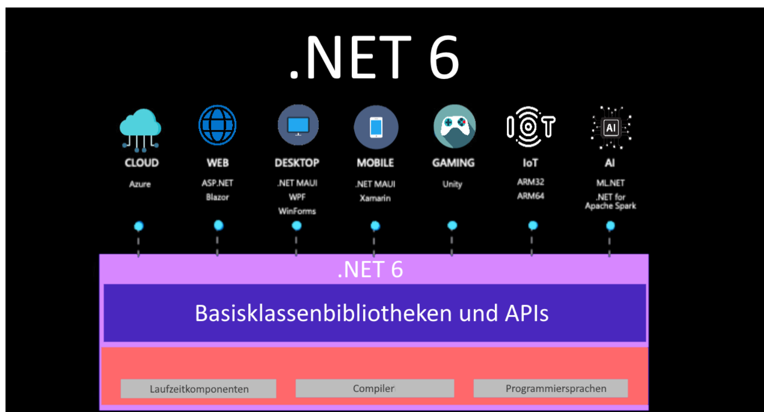


Bild 1.7 Überblick .NET 6

Außer diesen Anwendungsarten verspricht Microsoft unzählige Performanceverbesserungen, sodass es sich trotz der Plattformunabhängigkeit um das schnellste .NET handelt. Vor allem im Bereich Dateizugriff gibt es viele Verbesserungen, um das Lesen und Schreiben von Dateien zu optimieren.

Ein wesentlicher Zeitgewinn für alle Entwickler bietet Hot Reload. Hot Reload bietet die Möglichkeit, sowohl XAML- als auch C#-Code während der Laufzeit zu ändern und das Programm mit den geänderten Codeteilen weiterlaufen zu lassen, ohne jedes Mal das Programm neu starten zu müssen.

.NET 6 unterstützt auch alle Sprachfeatures von C# 10 (siehe Kapitel 8).

Minimal APIs ist ein weiterer Punkt, der dem Entwickler viele Codezeilen spart, jedoch auch teilweise kritisch gesehen wird, auch von den beiden Autoren dieses Buches. Dazu

aber später mehr, wenn wir die geeigneten Beispiele dazu bringen und Sie sich dann Ihre eigene Meinung bilden können.

Um .NET 6 nutzen zu können, benötigen Sie Visual Studio 2022 (siehe Kapitel 2) oder Visual Studio Code. Eine Entwicklung mit .NET 6 unter Visual Studio 2019 oder früheren Versionen ist nicht möglich.

1.4 Features von .NET 7

Wie bereits erwähnt, wurde mit .NET 7 zum allerersten Mal die Möglichkeit eines AOT-Compilers in Konsolenprojekten implementiert.

.NET 7 unterstützt auch den gesamten Sprachumfang der neuen C# Sprachversion 11, der in Kapitel 8 noch ausführlich vorgestellt wird.

Des Weiteren wurden einige Performanceverbesserungen durchgeführt und vor allem Verbesserungen an den Klassen im Namensraum *System.Text.Json* vorgenommen.

Zusätzlich gibt es neue oder verbesserte Klassen für folgende Einsatzmöglichkeiten (siehe Tabelle 1.1)

Tabelle 1.1 Neue und erweiterte Klassen in .NET 7

Einsatzmöglichkeiten	Klassennamen
Unterstützung für Mikro- und Nanosekunden	*DateTime*, *DateTimeOffset* *TimeOnly*, *TimeSpan* Erweiterungen um die Properties *Microsecond* und *Nanosecond*
Unterstützung zum Zugriff auf Tar-Archive	Klassen im Namespace *System.Formats.Tar*
Unterstützung zum Lesen aller Daten aus einem Stream	Neue Methoden *ReadExactly* und *ReadAtLeast* in der *Stream*-Klasse
Neue Typkonverter	*DateOnlyConverter*, *TimeOnlyConverter*, *Int128Converter*, *UInt128Converter*, *HalfConverter*
Neues Attribut zum Auslesen, welche Syntax in einem String verwendet wird	*StringSyntaxAttribute* mit Konstanten wie zum Beispiel *DateOnlyFormat*, *JsonFormat*, *Regex* etc.

Eine komplette Zusammenfassung mit weiterführenden Links finden Sie unter *https://learn.microsoft.com/de-de/dotnet/core/whats-new/dotnet-7*.

Mit .NET 7 wurde auch eine neue Version 17.4 von Visual Studio 2022 veröffentlicht.

1.5 Features von .NET 8

In der aktuellen Version des .NET Frameworks wurden die Möglichkeiten und Performance der AOT-Kompilierung weiter entwickelt und auch auf MAUI (zumindest für iOs und Android) übertragen.

Genauso wie in den Vorgängerversionen gibt es für .NET 8 eine neue C# Sprachversion 12, auch hier der Hinweis zur genaueren Darstellung in Kapitel 8.

Neben vielen kleinen Erweiterungen wurde auch der Namensraum *System.Text.Json* nochmals wesentlich erweitert (mehr dazu in Kapitel 17), und es gab auch sehr viele Performance- und Stabilisierungsverbesserungen für MAUI (mehr dazu dann in Kapitel 13).

Auch hier wollen wir die zusätzlichen Neuerungen von Klassen in Tabelle 1.2 als Überblick darstellen.

Tabelle 1.2 Neue und erweiterte Klassen in .NET 8

Einsatzmöglichkeiten	Klassennamen
Zufallsgeneratoren	Zusätzliche Methode *GetItems* in der Klasse *Random*, um nicht Zufallszahlen zu generieren, sondern um einen Eintrag aus einer Liste von Einträgen auszuwählen. Weitere Methode *Shuffle*, um die Reihenfolge von Listen wahlfrei zu mischen.
Verbesserung der Leistung von Collections	Neue Collection *FrozenCollection<TKey, TValue>*, bei der keine Schlüssel und Werte mehr verändert werden können, um ein schnelleres Lesen zu ermöglichen.
Datenvalidierung	Erweiterung von *System.ComponentModel.DataAnnotations* um folgende Attribute/Attributwerte: *RangeAttribute.MinimumIsExclusive* *RangeAttribute.MaximumIsExclusive* *LengthAttribute* *Base64StringAttribute* *AllowedValuesAttribute* *DeniedValuesAttribute*
Neue Metriken	*MeterOptions*
Kryptografie	Unterstützung von SHA-3-Hash-Grundelementen durch die Klassen *SHA3_256*, *SHA3_384*, *SHA3_512* etc.
Unterstützung für Https-Proxies	Neue Funktionalität in den Klassen *HttpClient* und *WebProxy*
Streambasierte ZipFile-Funktionalität	Neue Überladungen in der Methode *CreateFromDirectory* der Klasse *System.IO.Compression.ZipFile*
DependencyInjection mit Schlüsseln	Erweiterungen der Methoden in *Builder.Services* (in ASP.NET Core)

(Fortsetzung nächste Seite)

Tabelle 1.2 Neue und erweiterte Klassen in .NET 8 *(Fortsetzung)*

Einsatzmöglichkeiten	Klassennamen
GarbageCollection	Neue Funktionen zum Festlegen eines maximalen Arbeitsspeichergrenzwertes in der Klasse *GC*
Reflection nun auch für Funktionszeiger	*System.Type* besitzt nun eine Methode *IsFunctionPointer.*

Eine komplette Zusammenfassung mit weiterführenden Links finden Sie unter *https://learn.microsoft.com/de-de/dotnet/core/whats-new/dotnet-8.*

Mit .NET 8 wurde auch Version 17.8 von Visual Studio 2022 veröffentlicht, dazu aber gleich mehr im nächsten Kapitel.

2 Einstieg in Visual Studio 2022

Dieses Kapitel bietet Ihnen einen effektiven Schnelleinstieg in die Arbeit mit Visual Studio 2022. Gleich nachdem Sie die Hürden der Installation gemeistert haben, erstellen Sie mit C# Ihre ersten .NET-Anwendungen, werden dabei en passant mit den grundlegenden Features der Entwicklungsumgebung vertraut gemacht und nach dem Prinzip „so viel wie nötig" in die .NET-Philosophie eingeweiht. Nach der Lektüre dieses Kapitels und dem Nachvollzug der abschließenden Praxisbeispiele sollte der Einsteiger über eine brauchbare Ausgangsbasis verfügen, um den sich vor ihm gewaltig auftürmenden Berg von Spezialkapiteln in Angriff zu nehmen.

HINWEIS: Für .NET 8 wird, wie auch für die Vorgängerversionen, weiterhin Visual Studio 2022 verwendet, Sie müssen aber mindestens die Version 17.8 installiert haben, um mit .NET 8 unter Visual Studio arbeiten zu können. Für .NET 7 benötigen Sie mindestens die Version 17.4.

Eine komplette Überarbeitung der Oberfläche ist für die Version 17.9 von Visual Studio 2022 vorgesehen, die jedoch zum Zeitpunkt der Erstellung dieses Buches erst in einer Preview-Version verfügbar ist.

2.1 Die Installation von Visual Studio 2022

Ohne eine angemessen ausgestattete „Werkstatt" ist die Lektüre dieses Buches nutzlos. Programmieren lernt man nur durch Beispiele, die man unmittelbar selbst am Rechner ausprobiert!

HINWEIS: Voraussetzung für ein erfolgreiches Studium dieses Buches ist ein Rechner mit einer lauffähigen Installation von Visual Studio 2022!

2.1.1 Überblick über die Produktpalette

Für welches der im Folgenden aufgeführten Produkte man sich entscheidet, hängt von den eigenen Anforderungen und Wünschen ab und ist nicht zuletzt auch eine Frage des Geldbeutels.

HINWEIS: Bei Visual Studio 2022 handelt es sich erstmalig um eine 64-Bit-Applikation. Dies dürfte die Speicherproblematik, die man bei sehr großen Anwendungen in der Vergangenheit hatte, verschwinden lassen (vorausgesetzt Ihr Rechner besitzt genügend physikalischen Arbeitsspeicher).

Visual Studio 2022 Community

Bei dieser kostenfreien Version handelt es sich bereits um eine vollständig ausgestattete Entwicklungsumgebung für Windows-Desktop-, Web- und plattformübergreifende iOS-, Android- und Windows-Applikationen.

Sie kann auch für kommerzielle Projekte eingesetzt werden, wenn es sich dabei um Unternehmen mit weniger als 250 Mitarbeitern handelt.

HINWEIS: Der Inhalt dieses Buches kann komplett mit der **Community Edition** nachvollzogen werden!

Visual Studio 2022 Professional

Wie es der Name bereits suggeriert, handelt es sich bei diesem Standardpaket bereits um ein professionelles Werkzeug zur Entwicklung beliebiger Anwendungstypen im Team:

- Mit *CodeLens* ist ein leistungsstarkes Feature zum Verbessern der Produktivität Ihres Teams enthalten.
- Verschiedene Planungstools (Agile-Projekte, Teamräume, Diagramme usw.) dienen der Verbesserung der Team-Produktivität.
- Mit bestimmten MSDN-Abonnementleistungen erhalten Sie Zugang zu nützlicher Software für Entwicklung/Tests sowie Guthaben für die Azure-Cloudplattform

Visual Studio 2022 Enterprise

Hier handelt es sich um die Vollausstattung für Softwareentwickler, die im Team Anwendungen auf Enterprise-Niveau erstellen wollen. Neben allen Features der Professional-Version sind auch weitere Funktionen enthalten, die eine komplexe Datenbankentwicklung und eine durchgängige Qualitätssicherung ermöglichen sollen.

Die Screenshots in diesem Buch sind mit der Enterprise-Version erstellt. Wenn Ihnen also manche Menüpunkte, die auf dem Screenshot sichtbar sind, fehlen, liegt das an der unterschiedlichen Version. Zum Nachvollziehen werden Sie diese nicht benötigen.

2.1.2 Anforderungen an Hard- und Software

Die folgende Auflistung kann lediglich eine Orientierungshilfe sein:

- Betriebssystem: Windows 10, Version 1909 oder höher, Windows 11, Windows Server 2022, Windows Server 2019, Windows Server 2016 (weitere Hinweise unter *https://docs.microsoft.com/de-de/visualstudio/releases/2022/compatibility*).
- Unterstützte Architekturen: 64-Bit (x64); ARM-Betriebssysteme werden nicht unterstützt.
- Prozessor: 1,8-GHz-Prozessor oder schneller; Quad-Core empfohlen.
- RAM: Mindestens 4 GB verfügbarer physischer Arbeitsspeicher; empfohlen werden 16 GB.
- Festplatte: Je nach Installation zwischen 20 – 50 GB Speicherplatzbedarf, es werden SSD-Platten empfohlen.
- Grafikkarte: WXGA mit einer Mindestauflösung von 1366 × 768 Pixeln, empfohlen WXGA mit 1920 × 1080 Pixel oder höher.

Die Parameter von Prozessor und RAM sind als untere Grenzwerte zu verstehen.

2.2 Unser allererstes C#-Programm

Jeder Weg, und ist er noch so weit, beginnt mit dem ersten Schritt! Nachdem die Mühen der Installation gemeistert sind, wird es Zeit für ein allererstes C#-Programm. Wir verzichten allerdings auf das abgedroschene „Hello World“ und wollen gleich mit etwas Nützlicherem beginnen, nämlich der Umrechnung von Euro in Dollar.

Auch allein mit dem .NET Framework SDK, also ohne Visual Studio 2022, könnte man (zumindest rein theoretisch) vollwertige Programme entwickeln. Das wollen wir jetzt unter Beweis stellen, indem wir eine kleine Euro-Dollar-Applikation als sogenannte *Konsolenanwendung* – dem einfachsten Anwendungstyp – schreiben.

2.2.1 Vorbereitungen

Voraussetzungen sind lediglich ein simpler Texteditor und der C#-Kommandozeilencompiler *csc.exe*.

Compilerpfad eintragen

Der C#-Compiler *csc.exe* befindet sich, ziemlich versteckt, im Verzeichnis

```
\\Windows\Microsoft.NET\Framework\v4.0.30319
```

Dieser Compiler ist der sogenannte Legacy-Compiler der klassischen .NET Frameworks.

Da das Kompilieren direkt an der Kommandozeile ausgeführt werden soll, werden wir *csc.exe* in den Windows-Pfad aufnehmen, um so seinen Aufruf von jedem Ordner des Systems aus zu ermöglichen.

Es gibt nun zwei Möglichkeiten, an der Kommandozeile mit dem gesetzten Pfad zur *csc.exe* zu arbeiten.

Die einfache Variante ist, Sie starten in der Programmgruppe *Visual Studio 2022* (über das Startmenü) das Tool *Developer Command Prompt for VS 2022*. Daraufhin startet ein Konsolenfenster, in dem der korrekte Pfad bereits gesetzt ist.

Der aktuelle Compiler trägt den Namen *Roslyn*, und dieser Pfad (der auf unterschiedlichen Systemen an anderer Stelle steht) wird über den Aufruf des Tools *Developer Command Prompt für VS 2022* automatisch gesetzt.

Oder Sie fügen den Pfad den Windows-Umgebungsvariablen hinzu. Das ist zwar etwas aufwendiger, aber Sie müssen das nur ein einziges Mal machen und können dann egal, von welcher Konsole aus, auf diesen Pfad direkt zugreifen. Gehen Sie dabei wie folgt vor:

- Sie finden den Dialog zur Einstellung der *Path*-Umgebungsvariablen in der Windows-Systemsteuerung, wenn Sie nach dem Begriff *Path* suchen.

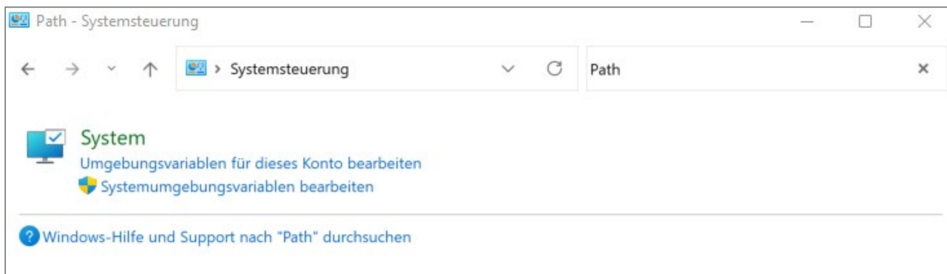


Bild 2.1 Suche in der Systemsteuerung

- Im Dialog *Systemeigenschaften* klicken Sie auf der Registerkarte *Erweitert* auf die Schaltfläche *Umgebungsvariablen*.

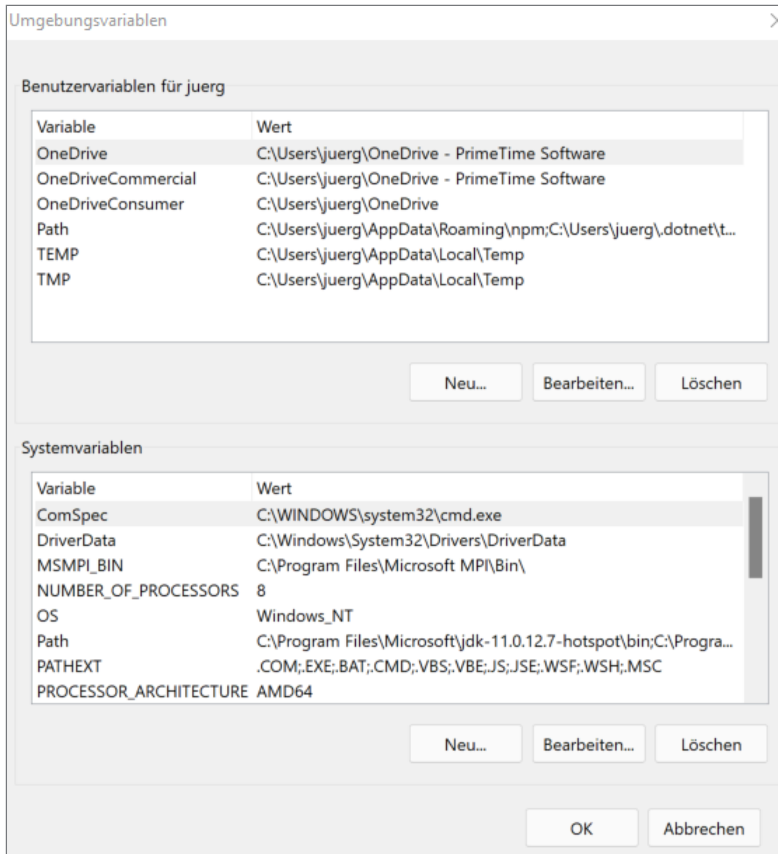


Bild 2.2 Umgebungsvariablen

- Wählen Sie in der unteren Liste *Systemvariablen* den *Path*-Eintrag und klicken Sie auf die *Bearbeiten*-Schaltfläche (siehe Abbildung Bild 2.2).
- Fügen Sie mit der Schaltfläche *Neu* den Namen des .NET-Framework-Verzeichnisses, in welchem sich *csc.exe* befindet (*C:\Windows\Microsoft.NET\Framework\v4.0.30319*), an die Liste an:

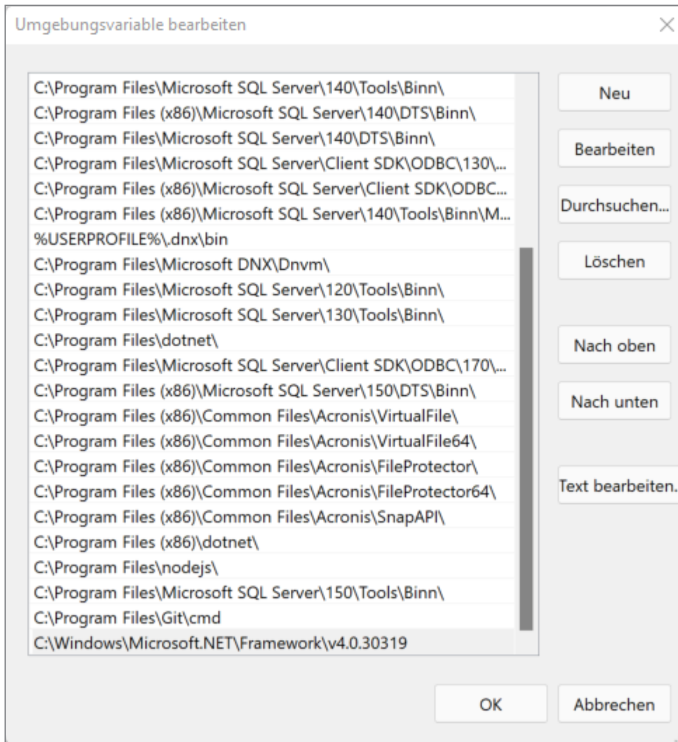


Bild 2.3 Umgebungsvariablen bearbeiten

Die erfolgreiche Übernahme der Änderungen an den *Path*-Umgebungsvariablen können Sie in einem kleinen Test überprüfen, bei dem Sie sich als durchaus nützlichen Nebeneffekt gleich die vielfältigen Optionen des Compilers anzeigen lassen.

Öffnen Sie dazu die Eingabeaufforderung (in Windows 10/11 finden Sie eine Verknüpfung zur Eingabeaufforderung z. B. in der Alle-Apps-Übersicht des Startmenüs unter *Windows-System*; oder Sie suchen einfach nach „Eingabeaufforderung“) und geben Sie (von einem beliebigen Verzeichnis aus) den folgenden Befehl ein, den Sie mit *Enter* abschließen:

```
csc /?
```

Aus der endlosen Parameterliste, die angezeigt wird, ist für uns die Option */target: exe* (abgekürzt */t:exe*) besonders interessant, da wir damit später unsere Konsolenanwendung kompilieren wollen.

Vom Funktionieren des Compilers können Sie sich erst dann überzeugen, wenn Sie eine C#-Source-Datei erstellt haben (siehe den folgenden Abschnitt).

2.2.2 Quellcode schreiben

Öffnen Sie den im Windows-Zubehör enthaltenen Editor und tippen Sie, ohne lange darüber nachzudenken, einfach den folgenden Text ein:

Listing 2.1 Umrechnung Euro in Dollar

```
using System;
class KonsolenDemo
{
  static void Main()
  {
    int i;
    Console.WriteLine("Umrechnung Euro in Dollar");
    do
    {
       double kurs;
       double euro;
       double dollar;
       Console.Write(«Kurs 1: «);
       kurs = Convert.ToDouble(Console.ReadLine());
       Console.Write(«Euro: «);
       euro = Convert.ToDouble(Console.ReadLine());
       dollar = euro * kurs;
       Console.WriteLine("Sie erhalten " +
                          dollar.ToString("0.00$"));
       Console.Write("Programm beenden? (j/n)");
       string s = Console.ReadLine();
       i = string.Compare(s, "j");
    } while(i != 0);
  }
}
```

HINWEIS: Achten Sie auf die exakte Einhaltung der Groß-/Kleinschreibung!

Speichern Sie die Datei unter dem Namen *EuroDollar.cs* in ein extra dafür angelegtes Verzeichnis, z. B. *EuroDollarKonsole*, ab.

2.2.3 Programm kompilieren und testen

Um bequem auf der Kommandozeile arbeiten zu können, kopieren Sie zunächst die Datei *cmd.exe* (Eingabeaufforderung aus ...*Windows**System32*) in dasselbe Verzeichnis, in welchem sich auch die Datei *EuroDollar.cs* befindet. Eventuell müssen Sie die Konsole im Administrator-Modus öffnen.

Klicken Sie doppelt auf *cmd.exe* und rufen Sie dann den C#-Compiler wie folgt auf:

```
csc /t:exe EuroDollar.cs
```

Nach dem erfolgreichen Kompilieren wird sich eine neue Datei *EuroDollar.exe* im Anwendungsverzeichnis befinden, ansonsten gibt der Compiler eine Fehlermeldung aus.

Klicken Sie doppelt auf die Datei *EuroDollar.exe* und führen Sie Ihr erstes C#-Programm aus!

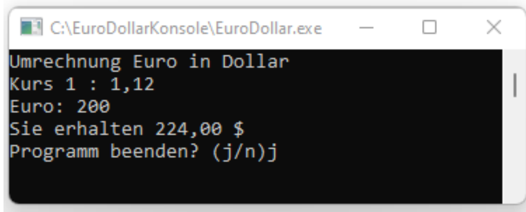


Bild 2.4
Unser erstes Programm, ein Euro-Dollar-Umrechnungsprogramm

2.2.4 Einige Erläuterungen zum Quellcode

Da wir auf die Grundlagen der Sprache C# erst ab den nächsten Kapiteln ausführlich zu sprechen kommen, sollen die folgenden Informationen zunächst nur allererste Eindrücke vermitteln.

Befehlszeilen und Gültigkeitsbereiche

Das Ende einer C#-Befehlszeile wird durch ein Semikolon (;) markiert. Der Zeilenumbruch spielt also keine Rolle! Gültigkeitsbereiche/Blöcke sind durch geschweifte Klammern { ... } eingegrenzt. Es muss also die Anzahl der öffnenden Klammern gleich der Anzahl schließender Klammern sein.

using-Anweisung

Mit der ersten Anweisung

```
using System;
```

binden Sie den *System*-Namensraum (*Namespace*) ein. Das hat den Vorteil, dass Sie statt

```
System.Console.WriteLine("Umrechnung Euro in Dollar");
```

nur noch

```
Console.WriteLine("Umrechnung Euro in Dollar");
```

schreiben müssen.

Wir werden später sehen, dass in der aktuellen C#-Version diese *using*-Angabe nicht mehr nötig ist.

class-Anweisung

Mit dieser Anweisung erzeugen Sie eine neue Klasse, in welcher in unserem Beispiel die *Main*-Prozedur deklariert wird. Diese definiert den Einsprungpunkt der Konsolenanwendung (also dort, wo das Programm startet).

HINWEIS: Ein C#-Programm besteht aus mindestens einer **.cs*-Datei mit einer Klasse.

WriteLine- und ReadLine-Methoden

Diese Methoden der *Console*-Klasse erlauben die Aus- und Eingabe von Text.

Während *Write* nur den Text an der aktuellen Position ausgibt, erzeugt *WriteLine* zusätzlich einen Zeilenumbruch.

ReadLine erwartet die Betätigung der **Enter**-Taste und liefert die vorher eingegebenen Zeichen als Zeichenkette zurück.

Assemblierung

Bei der vom C#-Compiler erzeugten Datei *EuroDollar.exe* handelt es sich **nicht** um eine herkömmliche EXE-Datei, sondern um eine sogenannte *Assemblierung*, die erst in Zusammenarbeit mit der CLR (*Common Language Runtime*) des .NET Frameworks in Maschinencode verwandelt wird (siehe Abschnitt 1.1.2).

2.2.5 Konsolenanwendungen sind out

Zwar sind mit der Klasse *System.Console* auch verschiedenste farbliche Effekte möglich, trotzdem: Bei wem weckt das triste Outfit einer Konsolenanwendung noch positive Emotionen[1]?

Als einfaches Hilfsmittel zum Erlernen von C# und für verschiedene Tools und Testzwecke hat dieser einfache Anwendungstyp aber durchaus noch seine Daseinsberechtigung.

Mit Visual Studio 2022 werden wir im Praxisteil dieses Kapitels die Euro-Dollar-Umrechnung nochmals programmieren, dann allerdings mit einer Windows-Oberfläche. Bevor es aber damit losgehen kann, sollten Sie sich ein wenig mit der Windows-Philosophie anfreunden.

Ja, wir wissen, auch Desktop-Applikationen gelten mittlerweile nicht mehr unbedingt als modern, aber es wäre für den Anfänger verfrüht, jetzt gleich Webanwendungen zu programmieren. Die Kapitel, die dieses Thema behandeln, befinden sich nicht ohne Grund am Ende des Buches.

2.3 Die Windows-Philosophie

Eine moderne Programmiersprache wie C# gibt Ihnen die faszinierende Möglichkeit, eigene Windows-Programme mit relativ geringem Aufwand und nach nur kurzer Einarbeitungszeit selbst zu entwickeln. Allerdings fällt der Einstieg umso leichter, je schneller man sich

[1] Von nostalgischen Erinnerungen an die DOS-Steinzeit einmal abgesehen.

Klarheit über die zunächst fremdartig anmutende, aber dann doch einfach und gleichzeitig genial erscheinende Windows-Philosophie verschafft.

2.3.1 Mensch-Rechner-Dialog

Die Art und Weise, **wie** die Kommunikation mit dem Benutzer (Mensch-Rechner-Dialog) abläuft, dürfte wohl der gravierendste Unterschied zwischen einer klassischen Konsolenanwendung und einer typischen Windows-Anwendung sein. Wie Sie es bereits im Einführungsbeispiel kennengelernt haben, „wartet" das Konsolenprogramm auf eine Eingabe, indem die Tastatur zyklisch abgefragt wird.

Unter Windows werden hingegen Ein- und Ausgaben in sogenannte „Nachrichten" umgesetzt, die zum Programm geschickt und dort in einer Nachrichtenschleife kontinuierlich verarbeitet werden. Daraus ergibt sich ein grundsätzlich anderes Prinzip der Interaktion zwischen Mensch und Rechner:

- Während bei einer Konsolenanwendung alle Initiativen für die Benutzerkommunikation vom Programm ausgehen, hat in einer Windows-Anwendung der Bediener den Hut auf. Er bestimmt durch seine Eingaben den Ablauf der Rechnersitzung.
- Während eine Konsolenanwendung in der Regel in einem einzigen Fenster läuft, erfolgt die Ausgabe bei einer Windows-Anwendung meist in mehreren Fenstern.

2.3.2 Objekt- und ereignisorientierte Programmierung

Vergleicht man den Programmaufbau einer Konsolenanwendung, welche aus einer langen Liste von Anweisungen besteht, mit einer Windows-Anwendung, so stellt man folgende Hauptunterschiede fest:

- Im Konsolenprogramm werden die Befehle sequenziell abgearbeitet, d. h. Schritt für Schritt hintereinander. Den Gesamtablauf kontrolliert in der Regel ein Hauptprogramm.
- In einer Windows-Anwendung laufen alle Aktionen objekt- und ereignisorientiert ab. Eine streng vorgeschriebene Reihenfolge für die Eingabe und Abarbeitung der Befehle gibt es nicht mehr. Für jede Aktivität des Anwenders ist ein Programmteil zuständig, der weitestgehend unabhängig von anderen Programmteilen agieren kann und muss. Daraus folgt auch das Fehlen eines Hauptprogramms im herkömmlichen Sinn!

Ein Windows-Programmierer hat sich vor allem mit den folgenden Begriffen auseinanderzusetzen:

Objekte (Objects)

Das sind zunächst die Elemente der Windows-Bedienoberfläche, denen wiederum Eigenschaften, Ereignisse und Methoden zugeordnet werden.

Beschränken wir uns der Einfachheit halber zunächst nur auf die visuelle Benutzerschnittstelle, so haben wir es in C# mit folgenden Objekten zu tun:

- **Formular/Fenster:** Das sind die Fenster, in welchen eine C#-Anwendung ausgeführt wird. In einem Fenster (*Form/Window*) können weitere untergeordnete Fenster, Komponenten (siehe unten), Text oder Grafik enthalten sein.
- **Steuerelemente:** Diese tauchen in vielfältiger Weise als Schaltflächen (*Button*), Textfelder (*TextBox*) etc. auf. Sie stellen die eigentliche Benutzerschnittstelle dar, über die mittels Tastatur oder Maus Eingaben erfolgen oder die der Ausgabe von Informationen dienen.

Der Objektbegriff wird auch auf die nichtvisuellen Elemente ausgedehnt und das geht schließlich so weit, dass innerhalb des .NET Frameworks sogar alle Variablen als Objekte betrachtet werden. Natürlich dürfen Sie als Programmierer auch eigene Objekte/Komponenten entwickeln und hinzufügen.

Eigenschaften (Properties)

Unter diesem Begriff versteht man die Attribute von Objekten, wie z. B. die Höhe (*Height*) und die Breite (*Width*) oder die Hintergrundfarbe (*BackColor*) eines Formulars oder Fensters. Jedes Objekt verfügt über seinen eigenen Satz von Eigenschaften, die teilweise nur zur Entwurfs- oder nur zur Laufzeit veränderbar sind.

Methoden (Methods)

Das sind die im Objekt definierten Funktionen und Prozeduren, die gewissermaßen das „Verhalten" beim Eintreffen einer Nachricht bestimmen. So säubert z. B. die *Clear*-Methode den Inhalt einer *ListBox*. Eine Methode kann z. B. auch das Verhalten des Objekts bei einem Mausklick, einer Tastatureingabe oder sonstigen Ereignissen (siehe unten) definieren. Im Unterschied zu den oben genannten Eigenschaften (Properties), die eine „statische" Beschreibung liefern, bestimmen Methoden die „dynamischen" Fähigkeiten des Objekts.

Ereignisse (Events)

Dies sind Nachrichten, die vom Objekt empfangen werden. Sie stellen die eigentliche Windows-Schnittstelle dar. So ruft z. B. das Anklicken eines Steuerelements mit der Maus in Windows ein *Click*-Ereignis hervor. Aufgabe eines Windows-Programms ist es, auf alle Ereignisse gemäß dem Wunsch des Anwenders zu reagieren. Dies geschieht in sogenannten *Ereignisbehandlungsroutinen* (Eventhandler).

Diese (zugegebenermaßen ziemlich oberflächlichen und unvollständigen) Erklärungen zur objektorientierten Programmierung sollen vorerst zum Einstieg genügen, theoretisch sauber wird die OOP erst in Kapitel 4 erläutert.

2.3.3 Programmieren mit Visual Studio 2022

Nicht nur Konsolenanwendungen, sondern auch Windows- und Web-Anwendungen lassen sich zumindest rein theoretisch mit den (kostenlos erhältlichen) Werkzeugen des *Microsoft .NET Framework SDK* erstellen. Allerdings ist dies extrem umständlich, da dazu zeitaufwendige Überlegungen zur Gestaltung der Benutzerschnittstelle[2] und ständiges Nachschlagen

[2] Das geht bis hin zum Abzählen von Pixeln!

in der Dokumentation erforderlich wären. Die intuitive Entwicklungsumgebung Visual Studio oder Visual Studio Code (noch zu erwähnen wäre Rider von der Firma JetBrains), das vor allem im Webbereich viele Anhänger gefunden hat, befreit Sie von diesem, besonders bei größeren Projekten sehr lästigen und nervtötenden „Herumwursteln“ und erlaubt (unabhängig von der verwendeten Programmiersprache) eine systematische Vorgehensweise in vier Etappen:

1. Visueller Entwurf der Bedienoberfläche
2. Zuweisen der Objekteigenschaften
3. Verknüpfen der Objekte mit Ereignissen
4. Kompilieren und Testen der Anwendung

Bereits die *erste Etappe* weist einen deutlichen Unterschied zur Konsolenprogrammiertechnik auf: Am Anfang steht der Oberflächenentwurf!

Ausgangsbasis ist das vom Editor bereitgestellte Startfenster, welches mit diversen Steuerelementen, wie Schaltflächen (*Buttons*) oder Editierfenstern (*TextBoxen*), ausgestattet werden kann. Im Werkzeugkasten finden Sie ein nahezu komplettes Angebot von Windows-typischen Steuerelementen. Diese werden ausgewählt, mittels Maus an ihre endgültige Position gezogen und (falls nötig) in ihrer Größe verändert.

Bereits während der ersten Etappe hat man, mehr oder weniger unbewusst, Eigenschaften, wie Position und Abmessungen von Formularen und Steuerelementen, verändert. In der zweiten Etappe braucht man sich eigentlich nur noch um die Eigenschaften zu kümmern, die von den Standardeinstellungen (Defaults) abweichen.

Die dritte Etappe haucht unseren bislang nur mit statischen Attributen ausgestatteten Objekten Leben ein. Hier muss in sogenannten *Ereignisbehandlungsroutinen* (Eventhandlern) festgelegt werden, **wie** das Formular oder das betreffende Steuerelement auf bestimmte Ereignisse zu reagieren hat. Visual Studio stellt auch hier „vorgefertigten“ Rahmencode für alle zum jeweiligen Objekt passenden Ereignisse zur Verfügung. Der Programmierer füllt diesen Rahmen mit C#-Quellcode aus. Hier können Methoden oder Prozeduren aufgerufen werden, aber auch Eigenschaften anderer Objekte lassen sich während der Laufzeit neu zuweisen.

In der vierten Etappe schlägt schließlich die Stunde der Wahrheit. Das von Ihnen geschriebene Programm wird vom C#-Compiler in einen Zwischencode übersetzt und läuft damit auf jedem Rechner, auf dem das .NET Framework installiert ist.

Allerdings ist die Arbeit des Programmierers nur in seltenen Fällen bereits nach einmaligem Durchlaufen aller vier Etappen getan. In der Regel müssen Fehler ausgemerzt und Ergänzungen vorgenommen werden, sodass sich der beschriebene Entwicklungszyklus auf ständig höherem Level so lange wiederholt, bis ein zufriedenstellendes Ergebnis erreicht ist.

Der in diesem Zyklus praktizierte visuelle Oberflächenentwurf, verbunden mit dem ereignisorientierten Entwurfskonzept, macht *Visual Studio 2022* zu einer hoch effektiven Entwicklungsumgebung für Windows- und Web-Anwendungen.

2.4 Die Entwicklungsumgebung Visual Studio 2022

Visual Studio 2022 ist eine universelle Entwicklungsumgebung (IDE[3]) für Windows-, Web- und für mobile Anwendungen, die auf Microsofts .NET-Technologie basieren. Alle notwendigen Tools, wie z. B. für den visuellen Oberflächenentwurf, für die Codeprogrammierung und für die Fehlersuche, werden bereitgestellt.

C# ist nur eine der möglichen objektorientierten Sprachen, die Sie unter Visual Studio 2022 einsetzen können. So werden z. B. noch Visual Basic, Visual C++ und Visual F# unterstützt.

HINWEIS: Der vorliegende Abschnitt soll lediglich einen allerersten Eindruck der IDE vermitteln, der sich erst durch die konkrete Arbeit mit den Praxisbeispielen am Ende dieses Kapitels verfestigen wird!

Die für dieses Buch verwendete Version von Visual Studio 2022 ist die Version 17.8.4. Die momentan als Preview verfügbare Version 17.9 wird einige Änderungen an der Benutzeroberfläche mit sich bringen, wird aber in diesem Buch noch nicht verwendet.

2.4.1 Neues Projekt

Wenn Sie Visual Studio starten, erscheint folgender Bildschirm:

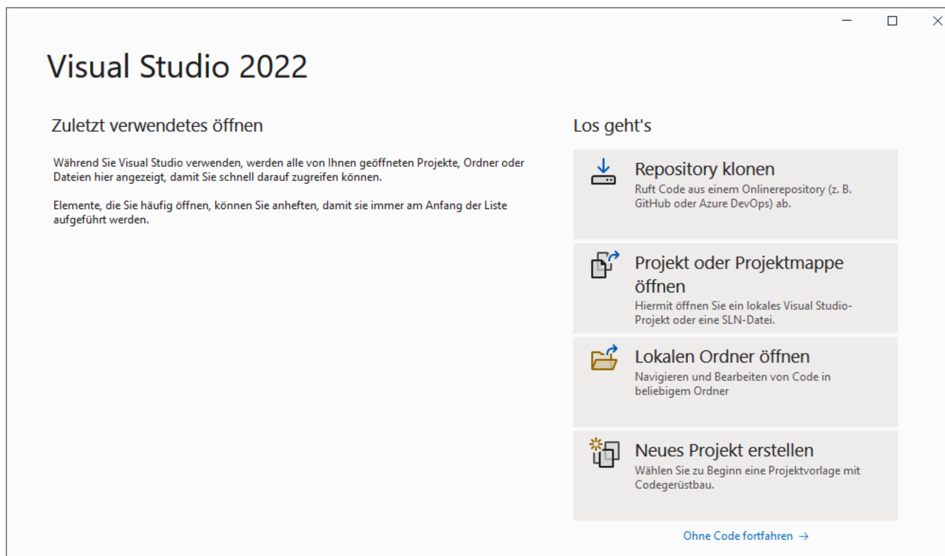


Bild 2.5 Visual Studio beim ersten Start

[3] *Integrated Development Environment*

Wählen Sie rechts unten den Eintrag *Neues Projekt erstellen*.

Daraufhin erscheint der Dialog zur Auswahl der Projektvorlage. In der Abbildung haben wir über die drei rechts oben befindlichen Klappboxen folgende Filter gesetzt:

- Sprache: *C#*
- Plattform: *Windows*
- Projekttyp: *Desktop*

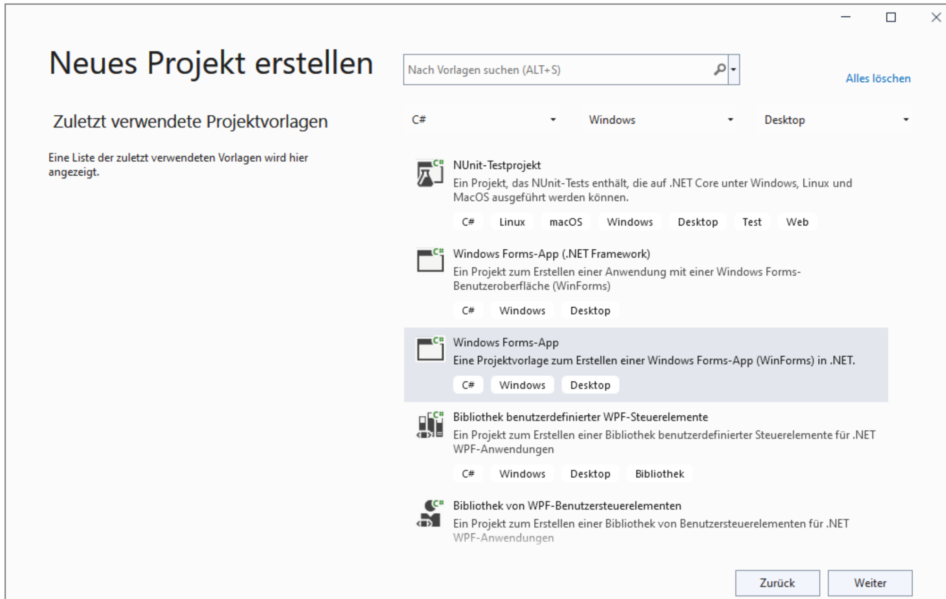


Bild 2.6 Auswahl der Projektvorlage

Wie Sie sehen, sind die Auswahlmöglichkeiten (vor allem vor dem Setzen der Filter) hier sehr umfangreich. Um den Einstieg so einfach wie möglich zu machen, wählen Sie die Vorlage *Windows Forms-App* (ohne den Zusatz *.NET Framework*, das wäre das klassische .NET Framework bis zur Version 4.8).

Im nachfolgenden Dialog können Sie noch bestimmte Werte angeben, um Ihr Projekt zu konfigurieren. Im ersten Beispiel ändern wir nur den Standort (Speicherort), in dem die Projektmappe gespeichert werden soll.

Haben Sie das Häkchen bei *Platzieren Sie die Projektmappe und das Projekt im selben Verzeichnis* gesetzt, so legt Visual Studio sowohl die Projektmappendatei (**.sln*) als auch die Projektdatei (**.csproj*) in dasselbe Verzeichnis. Da wir das in aller Regel nicht wollen, lassen wir den Haken bei dieser Einstellung weg. Somit wird für jedes Projekt in der Projektmappe ein eigener Unterordner angelegt.

Zum besseren Verständnis sei hier noch gesagt, dass eine Projektmappe ein Ordner für ein oder mehrere zusammenhängende Projekte ist. Sie können jederzeit der Projektmappe weitere Projekte hinzufügen.

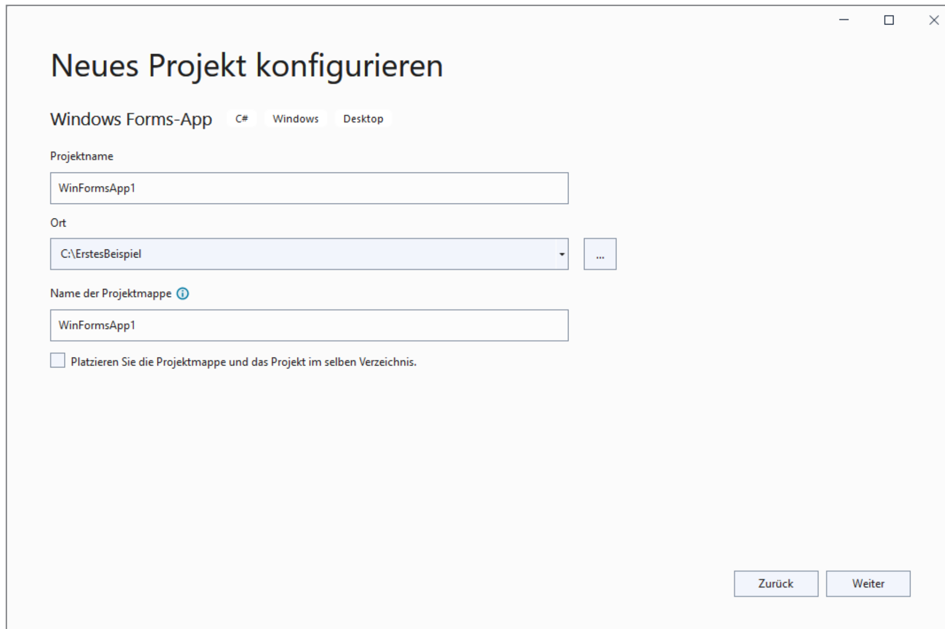


Bild 2.7 Speicherort des Projekts anpassen

Mit *Weiter* kommen Sie in den Dialog, in dem Sie die Framework-Version auswählen können. Da *Windows Forms-App* ohne den Zusatz *.NET Framework* ausgewählt wurde, werden nur die .NET-Core-Versionen angeboten, die Windows-Forms-Anwendungen unterstützen. Visual Studio 2022 erlaubt es, Programme für verschiedene .NET-Framework-Versionen zu entwickeln (Multi-Targeting). Wir wählen den Eintrag *.NET 8.0 (Langfristige Unterstützung)* aus.

Bild 2.8 kann je nach installierter Framework-Version auf Ihrem Rechner unterschiedlich dargestellt werden.

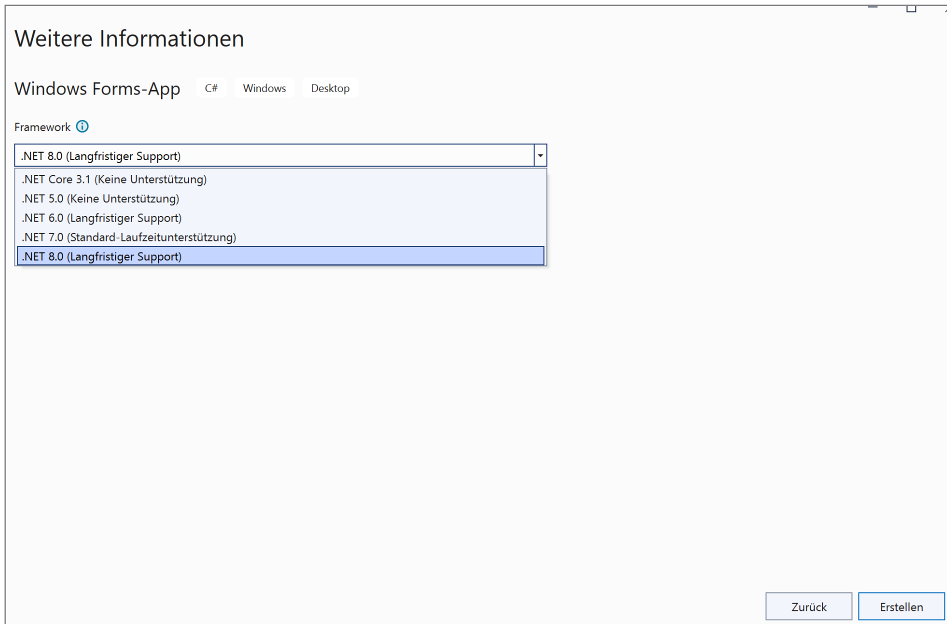


Bild 2.8 Auswahl der gewünschten .NET-Framework-Version

2.4.2 Die wichtigsten Fenster

Haben Sie als Projekttyp beispielsweise *Windows Forms-App* gewählt, könnte Ihnen Visual Studio 2022 etwa den folgenden Anblick bieten, wobei auf die für den Einsteiger zunächst wichtigsten Fenster besonders hingewiesen wird.

HINWEIS: Falls sich eines der Fenster versteckt hat, können Sie es jederzeit über das *Ansicht*-Menü herbeiholen.

Bild 2.9 Visual Studio mit einer neuen Windows-Forms-App

Der Projektmappen-Explorer

Da es unter Visual Studio möglich ist, mehrere Projekte gleichzeitig zu bearbeiten, gibt es eine Projektmappendatei mit der Extension *.sln* (Solution), deren Inhalt im Projektmappen-Explorer übersichtlich angezeigt wird. Sie können dieses Fenster deshalb ohne Übertreibung als „Schaltzentrale" Ihres Projekts betrachten.

Die zu jedem einzelnen C#-Projekt gehörigen Dateien und Einstellungen werden in einer XML-Datei mit der Extension *.csproj* (C#-Projekt) verwaltet. Der Aufbau der Projektdateien hat sich unter .NET Core verändert und ist wesentlich übersichtlicher geworden. Sie haben jetzt auch die Möglichkeit, die Projektdatei direkt in Visual Studio mit dem Kontextmenüeintrag *Projektdatei bearbeiten* zu editieren.

HINWEIS: Öffnen Sie Ihre Projekte immer über die *.sln*-Projektmappendatei, statt über die *.csproj*-Projektdatei, selbst wenn nur ein einziges Projekt enthalten ist!

Zur Bedeutung der einzelnen Einträge bzw. Dateien:

- *Abhängigkeiten:* Hier sind die aktuell für das Projekt gültigen Abhängigkeiten auf Frameworks, NuGet-Pakete und andere Projektverweise aufgelistet. Standardmäßig hat Visual Studio bereits die benötigten Verweise für die Projektvorlage erstellt, weitere können Sie mit der rechten Maustaste über das Kontextmenü hinzufügen.
- *Form1.cs:* Diese Datei enthält eine partielle Klasse[4], die den von Ihnen selbst hinzugefügten Code von *Form1* kapselt.
- *Form1.Designer.cs* (unter *Form1.cs* in der Baumstruktur): Diese Datei enthält eine partielle Klasse, die den vom Windows Forms Designer automatisch generierten Code von *Form1* kapselt. Der gesamte Code von *Form1* ist also zwischen den partiellen Klassen in *Form1.cs* und *Form1.Designer.cs* aufgeteilt.
- *Program.cs:* Eine statische Klasse, die die *Main*-Methode (den Einsprungpunkt der Anwendung) enthält. In dieser Methode wird durch Aufruf von *Application.Run* eine Nachrichtenschleife gestartet, die ununterbrochen auf Ereignisse wartet, damit die Anwendung darauf reagieren kann.

Durch Doppelklick auf eine dieser Dateien können Sie diese im Designer bzw. im Codefenster zur Bearbeitung öffnen.

Der (Formular)-Designer

Im Designer-Fenster entwerfen Sie die Programmoberfläche bzw. Benutzerschnittstelle. Ähnlich wie bei einem Zeichenprogramm entnehmen Sie dem Werkzeugkasten Steuerelemente und ziehen diese per Drag-and-drop auf ein Formular. Hier können Sie weitere Eigenschaften, wie z. B. Größe und Position, direkt mit der Maus und andere, wie z. B. Farbe und Schriftart, über das Eigenschaften-Fenster ändern.

Der Werkzeugkasten (Toolbox)

Den Werkzeugkasten werden Sie häufig benötigen (Menü *Ansicht/Toolbox* bzw. **STRG+W**, **X**). Auf verschiedenen Registerseiten, die später von Ihnen auch frei konfiguriert werden können, finden Sie eine umfangreiche Palette verschiedenster Steuerelemente für Windows-Forms-Anwendungen.

Das Eigenschaften-Fenster

Im Eigenschaften-Fenster (Menü *Ansicht/Eigenschaftenfenster* bzw. **STRG+W**, **P**) werden die zur Entwurfszeit editierbaren Eigenschaften des gerade aktiven Steuerelements aufgelistet[5]. Normalerweise hat jede Eigenschaft bereits einen Standardwert, den Sie in vielen Fällen übernehmen können. Alle vom Standard abweichenden Eigenschaften werden fett dargestellt.

Das Aktivieren eines bestimmten Steuerelements geschieht entweder durch Anklicken desselben auf dem Formular oder durch dessen Auswahl in der Klappbox am oberen Rand des Eigenschaften-Fensters.

---

[4] Das Konzept partieller Klassen wird im OOP-Kapitel (Abschnitt 4.7.3) erläutert.

[5] Lassen Sie sich nicht davon irritieren, dass das Eigenschaften-Fenster auf einer extra Registerseite auch die zum Steuerelement gehörigen Ereignisse anbietet.

Das Codefenster

Für die eigentliche Programmierung ist das Codefenster zuständig. Logischerweise wird dieses damit auch zu Ihrem Hauptbetätigungsfeld als C#-Programmierer. Um zum Beispiel das zu *Form1* (siehe Bild 2.10) gehörige Codefenster zu öffnen, klicken Sie auf den Designer von *Form1* mit der rechten Maustaste und wählen im Kontextmenü *Code anzeigen* (**F7**), oder Sie klicken einfach doppelt in die freie Fläche des Formulars. Bild 2.10 zeigt das Codefenster für *Form1* im Praxisbeispiel.

Der Code-Editor unterstützt Sie auf vielfältige Weise beim Schreiben von Quellcode. So markiert er Wörter farblich, unterbreitet Ihnen Vorschläge, weist Sie auf Fehler hin oder rückt den Text automatisch ein.

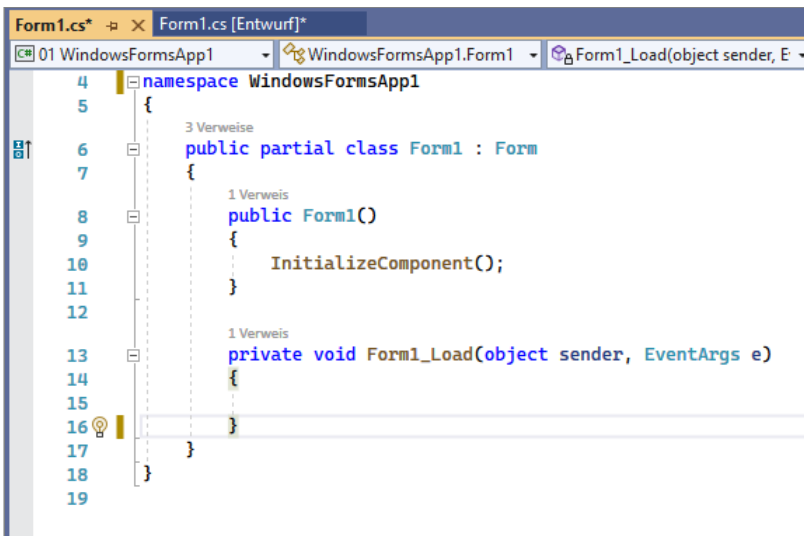


Bild 2.10 Code-Editor

2.4.3 Projektvorlagen in Visual Studio 2022 – Minimal APIs

In Visual Studio wurden ab .NET 6 für die meisten Projekttypen die Vorlagen (Templates) umgestellt und wesentlich vereinfacht (oder auch zusammengeschrumpft). Microsoft spricht von Minimal APIs, um mit so wenig Code wie möglich sofort zur eigenen Implementierung zu kommen.

An vielen Stellen ist das mir persönlich zu radikal, denn in dem Moment, wo es wieder komplexer wird, wird man den „gesparten“ Code dann doch benötigen.

Als Beispiel möchten wir die Vorlage für eine .NET-8-Konsolenanwendung zeigen.

Listing 2.2 Neues Template für eine Konsolenanwendung unter .NET 8

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

Es gibt keine Klassendefinition, keinen Einstiegspunkt in das Programm und keinen Namespace mehr. Wir fragen uns, was sich ein unerfahrener Programmierer denkt, wenn er so etwas sieht. Didaktisch ist das sehr fragwürdig.

In Kapitel 1 sehen Sie in Bild 1.2 eine Abbildung, bei der die Option *Keine Anweisungen der obersten Ebene verwenden* angehakt ist. Damit bekommen Sie den ausführlichen Code, so wie Sie ihn vielleicht aus früheren .NET-Versionen (vor .NET 6) kennen. Einfach diese Option bei der Projektanlage auswählen, und Sie bekommen immer den etwas ausführlicheren Code angelegt.

Listing 2.3 Komplette Konsolenanwendung ohne Verwendung von Anweisungen der obersten Ebene

```
using System;

namespace ConsoleApp2
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Hier sehen Sie alles, was benötigt und von einer objektorientierten Sprache erwartet wird: Namespace, Klassendefinition, *Main*-Routine als Haupteinstiegspunkt und den Code.

Wenn Sie das Setzen des Häkchens vergessen haben, können Sie noch immer das alte Template bekommen, wenn Sie dazu auf den Pinsel wie in Bild 2.11 dargestellt klicken und die Option *In Program.Main Programmformat konvertieren* auswählen.

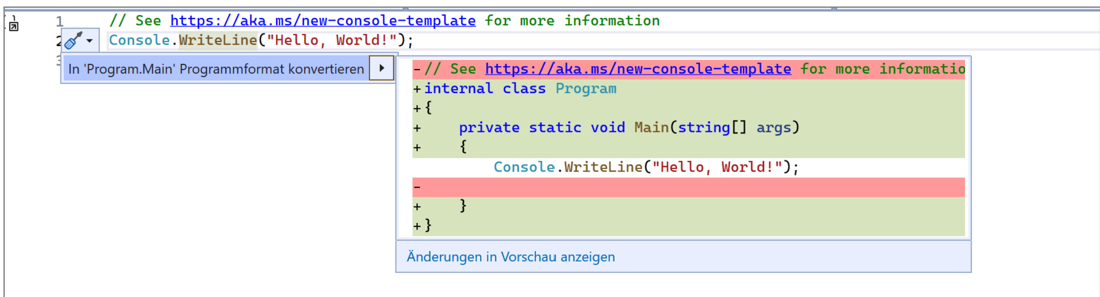


Bild 2.11 Konvertierung in vollständiges Template

Der minimale Code funktioniert aufgrund einiger neuer C#-10-Sprachfeatures, die wir uns in Kapitel 8 ansehen werden.

2.5 Praxisbeispiele

Bereits in Abschnitt 2.3.3, „Programmieren mit Visual Studio 2022“, hatten wir Ihnen die vier Etappen der Programmentwicklung in Visual Studio ganz allgemein erklärt. Jetzt wollen wir Nägel mit Köpfen machen und diese Schritte anhand von zwei Beispielen (ein ganz einfaches und ein etwas anspruchsvolleres) praktisch nachvollziehen.

Für diese kleinen Applikationen sind nicht die geringsten Programmierkenntnisse erforderlich. Es geht vielmehr darum, ein erstes Gefühl für die Anwendungsentwicklung unter Visual Studio zu bekommen.

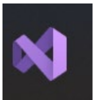
2.5.1 Unsere erste Windows-Forms-Anwendung

Die bescheidene Funktionalität beschränkt sich auf ein Fensterchen mit einer Schaltfläche, über die per Mausklick die Beschriftung der Titelleiste in „Hallo C#-Freunde“ geändert werden kann. Das Beispiel demonstriert, mit welch geringem Aufwand man in Visual Studio eigene Windows-Anwendungen erstellen kann. Der damit ausgelöste Aha-Effekt wird Sie sicher ausreichend motivieren, manche Durststrecken der nächsten Kapitel zu überstehen.

Wenn Sie nach dem Anlegen der Windows-Forms-App, in der Mitte dieses Kapitels, Visual Studio nicht wieder geschlossen haben, können wir an dieser Stelle direkt fortfahren. Ansonsten gehen Sie zum Öffnen dieses Projekts wie folgt vor.

1. Etappe: Visueller Entwurf der Bedienoberfläche

Der Programmstart von *Microsoft Visual Studio 2022* erfolgt entweder über das Windows-Startmenü oder noch schneller über eine Desktop- bzw. Taskleisten-Verknüpfung.



Danach klicken Sie einfach auf den Eintrag *WindowsFormsApp1.sln* in der Liste der zuletzt verwendeten Elemente.

Nach dem Klick dauert es ein kleines Weilchen, bis die Entwicklungsumgebung mit dem Startformular *Form1* erscheint. Darauf platzieren Sie ein Steuerelement vom Typ *Button*. Die dazu notwendige Vorgehensweise unterscheidet sich kaum von der bei einem normalen Zeichenprogramm.

Falls die Toolbox gerade nicht sichtbar ist, klicken Sie im Menü *Ansicht* auf den Eintrag *Toolbox* und wählen dann einfach das gewünschte Steuerelement aus:

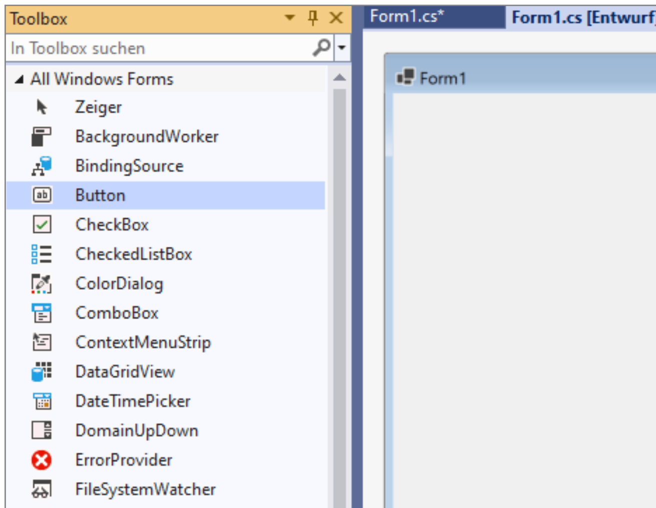


Bild 2.12 Button auswählen und auf die Form ziehen

Ein schneller Doppelklick befördert das Steuerelement direkt auf das Formular. Sie können aber auch den gewünschten *Button* in gewohnter Windows-Manier auf das Formular ziehen, dort absetzen und auf die gewünschte Größe zoomen.

2. Etappe: Zuweisen der Objekteigenschaften

Der *Button* trägt noch seine standardmäßige Beschriftung *button1*. Um diese in „Start" zu ändern, muss die *Text*-Eigenschaft geändert werden. Markieren Sie dazu das Steuerelement mit der Maus und rufen Sie mit **STRG**+**W**, **P** (bzw. über das Menü *Ansicht*) das Eigenschaften-Fenster auf. Ändern Sie im Eigenschaften-Fenster die *Text*-Eigenschaft von ihrem Standardwert „button1" in „Start".

Verwechseln Sie die *Text*-Eigenschaft nicht mit der *Name*-Eigenschaft. Immer wenn Sie ein neues Steuerelement platzieren, setzt Visual Studio standardmäßig den Wert von *Text* zunächst auf den von *Name*.

Es dürfte Ihnen nun auch keine Schwierigkeiten bereiten, über die *Font*-Eigenschaft von *button1* auch noch die Schriftgröße etc. zu ändern.

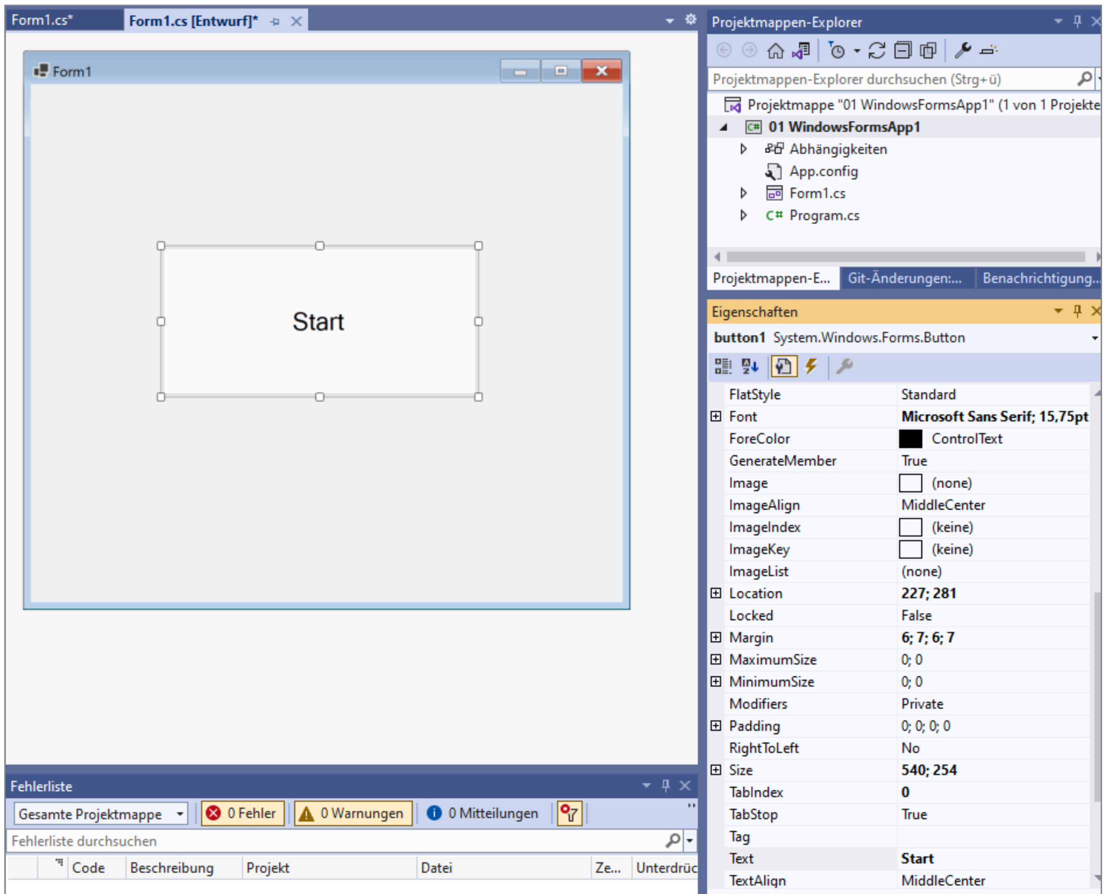


Bild 2.13 Setzen von Eigenschaften

3. Etappe: Verknüpfen der Objekte mit Ereignissen

Klicken Sie doppelt auf die Komponente *button1*, so öffnet sich das Codefenster. Richten Sie nun Ihr Augenmerk auf die Schreibmarke, welche im vorgefertigten Rahmencode innerhalb der *Click*-Ereignisbehandlungsroutine (Eventhandler) blinkt. Hier tragen Sie Ihren C#-Code ein, der festlegt, **was** passieren soll, wenn zur Programmlaufzeit (also nicht jetzt zur Entwurfszeit!) der Anwender auf diese Schaltfläche klickt.

In unserem Fall wollen wir erreichen, dass sich die Beschriftung der Titelleiste des Formulars ändert. Das bedeutet, dass wir die *Text*-Eigenschaft des Objekts *Form1*, dessen Standardwert bislang ebenfalls „Form1" lautete, neu zuweisen müssen. Diesmal aber tun wir das nicht im Eigenschaften-Fenster, sondern per C#-Code.

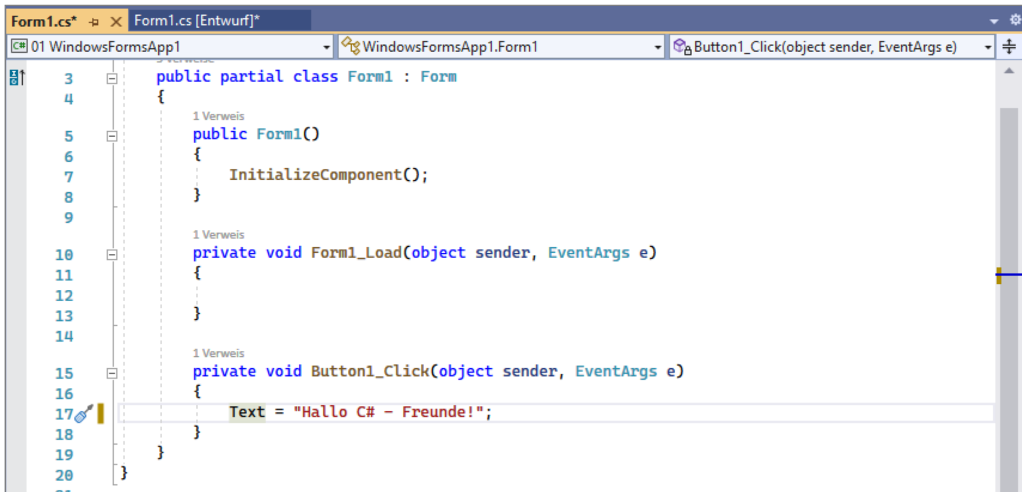


Bild 2.14 Schreiben einer Ereignisroutine

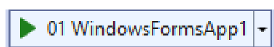
Fügen Sie die fett hervorgehobene Zeile in den bereits vorhandenen Rahmencode ein:

Listing 2.4 Ändern der Fensterüberschrift

```
private void Button1_Click(object sender, EventArgs e)
{
    Text = "Hallo C# – Freunde!";
}
```

4. Etappe: Programm kompilieren und testen

Kompilieren Sie das Programm durch Klicken auf das kleine Dreieck in der Symbolleiste (bzw. Menü *Debuggen/Debuggen starten* oder **F5**).



Sie befinden sich nun im Ausführungsmodus. Ihr Programm „lebt“ jetzt, denn die Schaltfläche lässt sich anklicken und die Beschriftung der Titelleiste ändert sich tatsächlich.

Das Programm beenden Sie, indem Sie auf das kleine rote Quadrat in der Symbolleiste klicken (bzw. Menü *Debuggen/Debuggen beenden* oder **Umschalt**+**F5**) oder aber das Formular einfach in altbekannter Windows-Manier schließen.

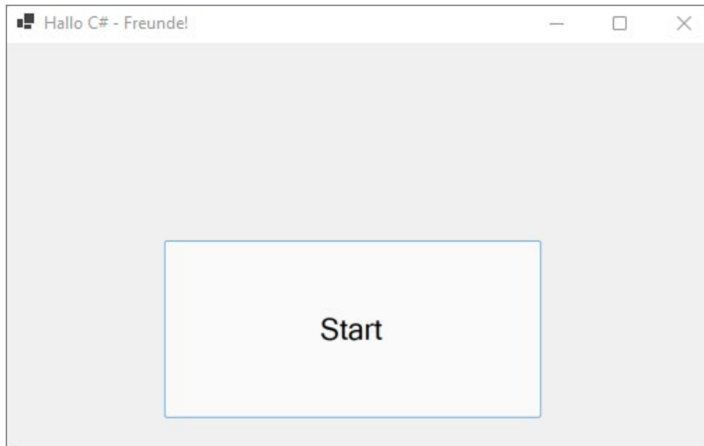


Bild 2.15 Hallo C#-Freunde

Gratulation – Sie haben soeben Ihre erste Windows-Forms-Anwendung geschrieben!

Dem erfahrenen Leser wird wahrscheinlich während der Laufzeit des Programms ein neues Icon aufgefallen sein (siehe Bild 2.16).

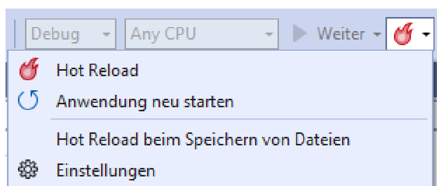


Bild 2.16 Hot Reload

Hier können Sie Einstellungen für Hot Reload vornehmen. Sie können jetzt während der Laufzeit der Anwendung Änderungen am Programmcode durchführen und diese werden automatisch durch Klick auf *Hot Reload* oder auch beim Speichern der Datei übernommen (wenn Sie die Option *Hot Reload beim Speichern von Dateien* angeklickt haben).

Ändern Sie einfach den Text, der in der Titelleiste angezeigt wird, und speichern Sie die Codedatei. Dann klicken Sie noch mal auf den Button *Start* und Sie werden sehen, dass jetzt der geänderte Text in der Titelleiste angezeigt wird, ohne dass Sie die Anwendung neu starten mussten.

Hot Reload ist eines der zeitsparendsten Features in Visual Studio 2022.

Bemerkungen

- In diesem Beispiel haben Sie ganz nebenbei auch gelernt, dass man Eigenschaften (Properties) nicht nur zur Entwurfszeit im Eigenschaften-Fenster zuweist, sondern dies auch zur Laufzeit per Code tun kann.
- Die Properties, die Sie im Eigenschaften-Fenster zuweisen, bezeichnet man auch als *Starteigenschaften*. Zur Laufzeit können diese – wie im Beispiel für die *Text*-Eigenschaft des Formulars gezeigt – durchaus ihren Wert ändern.

- Das .NET-SDK empfiehlt, dass alle Klassen innerhalb eines Namensraums (*namespace*) definiert werden[6]. Visual Studio verwendet automatisch den Namen Ihres Projekts (wir haben das bei der Standardvorgabe *WindowsFormsApp1* belassen) als oberste Ebene des Namensraums. Insgesamt hat also der Quellcode Ihrer ersten Windows-Anwendung folgendes Aussehen, wobei nur die fett hervorgehobene Zeile von Ihnen selbst getippt werden musste:

Listing 2.5 Gesamter Code des Formulars

```
namespace WindowsFormsApp1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Button1_Click(object sender, EventArgs e)
        {
            Text = "Hallo C# – Freunde!";
        }
    }
}
```

HINWEIS: Beachten Sie die durch die {}-Klammern eingegrenzten Gültigkeitsbereiche!

- Die als Ergebnis des Kompilierprozesses generierte *.exe*-Datei finden Sie im Unterverzeichnis *...\WindowsFormsApp1\bin\Debug\net8.0-windows* Ihres Projektordners. Es handelt sich hierbei allerdings **nicht** um eine klassische EXE-Datei, sondern um eine sogenannte Assemblierung. Haben Sie die Programmentwicklung erfolgreich abgeschlossen und Visual Studio beendet, so können Sie später jederzeit in dieses Verzeichnis wechseln, um durch Doppelklick auf die Datei *WindowsFormsApp1. exe* das Programm zu starten.
- Direkt im Projektverzeichnis befindet sich die Projektmappendatei *WindowsFormsApp1. sln*. Wenn Sie auf diese Datei klicken[7], so wird standardmäßig Visual Studio geöffnet und das Programm wird in die Entwicklungsumgebung geladen.

2.5.2 Umrechnung Euro-Dollar

Diesmal soll es keine Spielerei, sondern ein durchaus nützliches Programmchen sein – die Umrechnung von Euro in Dollar, ein simpler Währungsrechner also. Durch Vergleichen mit der in Abschnitt 2.2 beschriebenen ersten C#-Anwendung dürften auch die Unterschiede

[6] Mehr zum Konzept der Namensräume erfahren Sie in Kapitel 6.
[7] Das werden Sie z. B. häufig beim Laden von Beispielprojekten tun.

der klassischen Konsolentechnik zur visualisierten, objekt- und ereignisorientierten Windows-Programmierung ganz deutlich zu Tage treten.

1. Etappe: Visueller Entwurf der Bedienoberfläche

Legen Sie ein neues Visual-C#-Projekt vom Typ *Windows Forms-App* an und geben Sie ihm den Namen „EuroDollar".

Ziel ist die folgende Bedienoberfläche, die Sie jetzt mühelos im Designer-Fenster erstellen (siehe Bild 2.17)[8].

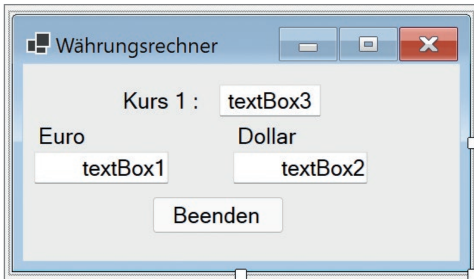


Bild 2.17
Formular für den Währungsrechner

Sie brauchen außer dem bereits vorhandenen Startformular *Form1* drei *Label* zur Beschriftung, drei *TextBoxen* für die Eingabe und einen *Button* zum Beenden des Programms. Für die Namensgebung sorgt Visual Studio automatisch, es sei denn, Sie möchten den Objekten eigene Namen geben.

HINWEIS: Konzentrieren Sie sich in der ersten Etappe nur auf Lage und Abmessung der Komponenten, nicht auf deren Beschriftung, da Eigenschaften erst in der nächsten Etappe angepasst werden!

Beim Platzieren und bei der Größenanpassung der Komponenten gehen Sie ähnlich vor, wie Sie es bereits von vektororientierten Zeichenprogrammen (z. B. Visio, PowerPoint) gewöhnt sind:

- Im Werkzeugkasten klicken Sie auf das Symbol für die *TextBox*-Komponente. Der Mauszeiger wechselt sein Aussehen.
- Danach bewegen Sie den Mauszeiger zu der Stelle von *Form1*, an welcher sich die linke obere Ecke von *textBox1* befinden soll, drücken die Maustaste nieder und zoomen (bei gedrückt gehaltener Maustaste) die *TextBox* auf ihre endgültige Größe. Analog verfahren Sie mit *textBox2* und *textBox3*.
- Nun klicken Sie im Werkzeugkasten auf das Symbol für die *Label*-Komponente und erzeugen auf die gleiche Weise *label1*, *label2* und *label3*.
- Schließlich bleibt noch *button1*, den Sie am unteren Rand von *Form1* absetzen.

[8] Der Inhalt der drei Textboxen ist standardmäßig leer. Er wurde hier nur aus Gründen der Übersichtlichkeit mit deren Namen beschriftet.

2. Etappe: Zuweisen der Objekteigenschaften

Unser Programm besteht nun aus insgesamt acht Komponenten: einem Formular und sieben Steuerelementen. Alle Eigenschaften haben bereits ihre Standardwerte. Einige davon müssen wir allerdings noch ändern. Dies geschieht mithilfe des Eigenschaften-Fensters. Wenn Sie mit der Maus auf eine Komponente klicken und danach die **F4**-Taste (diese funktioniert auch zur Anzeige des Eigenschaften-Fensters) betätigen, erscheint das Eigenschaften-Fenster der Komponente mit der Liste aller zur Entwurfszeit verfügbaren Eigenschaften.

- Beginnen Sie mit *label1*, das die Beschriftung „Euro" tragen soll. Die Beschriftung kann mit der *Text*-Eigenschaft geändert werden. Standardmäßig entspricht diese der *Name*-Property, in unserem Fall also „label1". Um das zu ändern, klicken Sie auf das *Label* und tragen anschließend in der Spalte rechts neben dem *Text*-Feld die neue Beschriftung ein (die alte ist vorher „wegzuradieren"). Analog verfahren Sie mit den beiden anderen Labels (Beschriftung „Dollar" und „Kurs 1: ").
- Auch *button1* muss natürlich seine neue *Text*-Eigenschaft („Beenden") erhalten.
- Schließlich klicken Sie auf eine leere Fläche von *Form1*, um anschließend mit **F4** das Eigenschaften-Fenster für das Formular aufzurufen und dessen *Text*-Eigenschaft entsprechend der gewünschten Beschriftung der Titelleiste zu modifizieren.

Die Tabelle gibt eine Zusammenstellung aller Objekteigenschaften, die wir geändert haben:

Name des Objekts	Eigenschaft	Neuer Wert
Form1	*Text* *Font.Size*	Währungsrechner *10*
label1	*Text*	Euro
label2	*Text*	Dollar
label3	*Text*	Kurs 1:
textBox1	*TextAlign*	*Right*
textBox2	*TextAlign*	*Right*
textBox3	*TextAlign*	*Center*
button1	*Text*	Beenden

3. Etappe: Verknüpfen der Objekte mit Ereignissen

Während Sie die beiden Vorgängeretappen noch locker bewältigen konnten, beginnt jetzt Ihre Hauptarbeit als C#-Programmierer. Wechseln Sie zum Codefenster *Form1.cs* (auch mit **F7**, *Ansicht/Code* oder dem Kontextmenü des Formulars möglich). Was Sie erwartet, ist die von Visual Studio vorbereitete Klassendeklaration von *Form1*. Wie Sie sehen, können Sie einzelne Bereiche (Regionen) durch das Plus- bzw. Minus-Symbol am linken Rand auf- bzw. zuklappen.

Zunächst fügen Sie eine Anweisung ein, um drei Variablen des *double*-Datentyps (Fließpunktzahlen) zu deklarieren. Gleichzeitig werden diese Variablen mit dem Wert 1.0 initialisiert:

Listing 2.6 Definition von privaten Variablen

```
private double euro = 1.0;
private double dollar = 1.0;
private double kurs = 1.0;
```

Im Unterschied zur einfachen Konsolenanwendung, bei der uns das Programm die Einhaltung einer bestimmten Eingabereihenfolge aufgezwungen hat, soll in unserer Windows-Forms-Anwendung die Berechnung immer dann neu gestartet werden, wenn wir bei der Eingabe in eine der drei Textboxen irgendeine Taste losgelassen haben. Wir müssen also für jede der Textboxen einen eigenen Eventhandler für das *KeyUp*-Ereignis schreiben!

Dabei ist eine fast schon rituelle Erstellungsreihenfolge zu beachten, die Sie mit fortschreitender Programmierpraxis sehr bald auch im Schlaf ausführen können:

- **Objekt auswählen:** Zur Objektauswahl klicken Sie auf das Objekt im Designer-Fenster und öffnen mit **F4** das Eigenschaften-Fenster.

  Klicken Sie im Eigenschaften-Fenster oben auf das ⚡-Symbol, um die Ereignisliste zur Anzeige zu bringen.
- **Ereignis auswählen:** Zur Ereignisauswahl doppelklicken Sie auf das gewünschte Ereignis. Als Resultat werden automatisch die erste und die letzte Zeile (Rahmencode) des Eventhandlers generiert und im Codefenster angezeigt.
- **Ereignisbehandlungen programmieren:** Füllen Sie den Eventhandler mit den gewünschten C#-Anweisungen aus.

Wir beginnen in unserem Beispiel mit dem *KeyUp*-Eventhandler für *textBox1*, der immer dann ausgeführt wird, wenn der Benutzer den Euro-Betrag ändert. Doppelklicken Sie also auf *KeyUp* und das Codefenster öffnet sich mit dem automatisch erzeugten Rahmencode des Eventhandlers.

Füllen Sie nun den Rahmencode wie folgt aus:

Listing 2.7 Ereignishandler für das KeyUp-Ereignis der TextBox

```
private void TextBox1_KeyUp(object sender, KeyEventArgs e)
{
    euro = Convert.ToDouble(textBox1.Text);
    dollar = euro * kurs;
    textBox2.Text = dollar.ToString("#,##0.00");
}
```

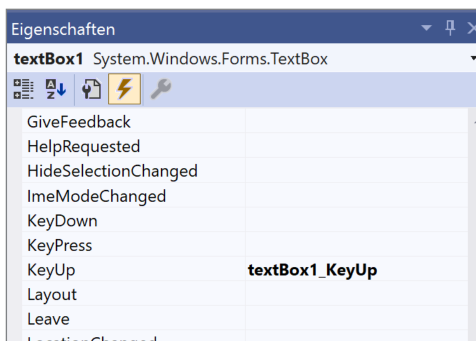


Bild 2.18 Anzeige der Events

HINWEIS: Sie müssen nur die Anweisungen innerhalb der geschweiften Klammern selbst einfügen. Der übrige Rahmencode wird automatisch erzeugt, wenn Sie die oben erläuterte Erstellungsreihenfolge beachten!

Der Prozedurkopf des Eventhandlers verweist standardmäßig vor dem Unterstrich (_) auf den Namen des Objekts und danach auf das entsprechende Ereignis. Das vorangestellte *private* verdeutlicht, dass auf die Prozedur nur innerhalb der *Form1*-Klasse zugegriffen werden kann.

Auf analoge Weise erzeugen Sie die Eventhandler für die Steuerelemente *textBox2* und *textBox3*, geben aber dann den jeweils geänderten Umrechnungscode ein.

Ändern des Dollar-Betrags:

Listing 2.8 Umrechnungscode

```
private void textBox2_KeyUp(object sender, KeyEventArgs e)
{
     dollar = Convert.ToDouble(textBox2.Text);
     euro = dollar / kurs;
     textBox1.Text = euro.ToString("#,##0.00");
}
```

Ändern des Umrechnungskurses:

```
private void textBox3_KeyUp(object sender, KeyEventArgs e)
{
     kurs = Convert.ToDouble(textBox3.Text);
     dollar = euro * kurs;
     textBox2.Text = dollar.ToString("#,##0.00");
}
```

HINWEIS: Grübeln Sie jetzt noch nicht über den tieferen Sinn der einzelnen Anweisungen nach, denn dazu haben Sie in den späteren Kapiteln noch genug Zeit!

Damit Sie bereits unmittelbar nach dem Programmstart sinnvolle Werte in den drei Textboxen sehen, ist folgender Eventhandler für das *Load*-Ereignis des *Form1*-Objekts hinzuzufügen:

Listing 2.9 Vorbelegung der Textfelder

```
private void Form1_Load(object sender, EventArgs e)
{
     textBox1.Text = euro.ToString();
     textBox2.Text = dollar.ToString();
     textBox3.Text = kurs.ToString();
}
```

Beim Klick auf den *Beenden*-Button soll das Formular entladen werden. Wählen Sie in der Objektauswahlliste des Eigenschaften-Fensters den Eintrag *button1* und anschließend in der Ereignisauswahlliste das *Click*-Event:

Listing 2.10 Schließen des Fensters und somit auch der Anwendung

```
private void button1_Click(object sender, EventArgs e)
{
    Close();
}
```

4. Etappe: Programm kompilieren und testen

Starten Sie die Anwendung durch Klicken auf den grünen Pfeil in der Symbolleiste (oder **F5**-Taste) und im Handumdrehen ist Ihre C#-Windows-Anwendung kompiliert und ausgeführt!

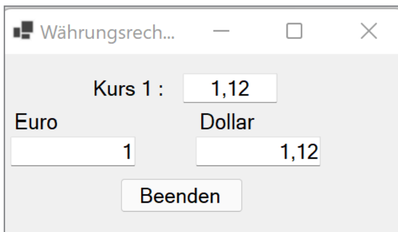


Bild 2.19
Euro-Dollar-Rechner als Windows-Anwendung

Spielen Sie ruhig ein wenig mit verschiedenen Werten herum, um sich den Unterschied zwischen Konsolen- und Windows-Programmen zu verinnerlichen. Da es keine vorgeschriebene Reihenfolge für die Benutzereingaben mehr gibt, werden die anderen Felder sofort aktualisiert. Eine spezielle „="-Schaltfläche (etwa wie bei einem Taschenrechner) ist deshalb überflüssig.

HINWEIS: Achten Sie darauf, dass Sie als Dezimaltrennzeichen das Komma und nicht den Punkt eingeben. Letzterer dient als Tausender-Separator (zumindest auf einem deutschen Betriebssystem).

IntelliSense – die hilfreiche Fee

Eines der bemerkenswertesten Features des Code-Editors ist seine sogenannte *IntelliSense*, auf die Sie mit Sicherheit bereits beim Eintippen des Quellcodes aufmerksam geworden sind. Sobald Sie den Namen eines Objekts bzw. eines Steuerelements mit einem Punkt abschließen, erscheint wie von Geisterhand eine Liste mit allen Eigenschaften und Methoden des Objekts. Das befreit Sie von dem lästigen Nachschlagen in der Hilfe und bewahrt Sie vor Schreibfehlern.

HINWEIS: Wenn Sie das markierte Element übernehmen wollen, brauchen Sie den Namen nicht zu Ende zu schreiben, da die IntelliSense den kompletten Text automatisch ergänzt.

```
1 Verweis
private void Form1_Load(object sender, EventArgs e)
{
    textBox1.Text = euro.ToString();
    textBox2.Text = dollar.ToString();
    textBox3.Text = kurs.ToString();
}

1 Verweis
private void button1_Clic
{
    Close();
}
```

★ ToString
★ GetType
CompareTo
Equals
GetHashCode
GetType
GetTypeCode
ToString

string double.ToString() (+ 5 Überladungen)
Converts the numeric value of this instance to its equivalent string representation.
★ IntelliCode-Vorschlag basierend auf diesem Kontext

Bild 2.20 IntelliSense in Aktion

Die Glühbirne sorgt für Erleuchtung

Bereits beim Schreiben des Quellcodes werden Sie von Visual Studio auf grundsätzliche Syntaxfehler (z. B. ein vergessenes Semikolon) hingewiesen. Im Allgemeinen geschieht dies durch Unterstreichen mit einer wellenförmigen (roten) Linie. Wenn Sie mit der Maus auf diese Linie zeigen, erscheint links unterhalb das Symbol einer gelben Glühbirne mit Hinweisen zur Behebung des Fehlers.

In Bild 2.22 wurde z. B. die Variable *euro* falsch eingetippt. Wählen Sie aus den Vorschlägen einfach die passende Fehlerbereinigung aus.

```
1 Verweis
private void Form1_Load(object sender, EventArgs e)
{
    textBox1.Text = eur.ToString();
    textBox2.Text = dollar.ToString();
    textBox3.Text
}

1 Verweis
private void butto
{
    Close();
}
```

Feld "Form1.eur" generieren
Schreibgeschütztes Feld "Form1.eur" generieren
Eigenschaft "Form1.eur" generieren
Lokales "eur" generieren
Parameter "eur" generieren
class-Objekt "eur" in neuer Datei generieren
class-Objekt "eur" generieren
Geschachteltes class-Objekt "eur" generieren
Neuen Typ generieren...
Ändern Sie "eur" in "euro".
Ändern Sie "eur" in "EuroDollar".
Ändern Sie "eur" in "FtpStyleUriParser".

CS0103 Der Name "eur" ist im aktuellen Kontext nicht vorhanden.

```
Zeilen 12 bis 14
private double kurs = 1.0;
private object eur;

private void textBox1_KeyUp(object sender, KeyEventArgs e)
```

Änderungen in Vorschau anzeigen

Eigenschaften

Bild 2.21 Glühbirne in Aktion

Fehler beim Kompilieren

Andere Fehler treten erst beim Kompilieren ans Tageslicht, wobei Sie durch folgende Meldung (Bild 2.23) aufgeschreckt werden:

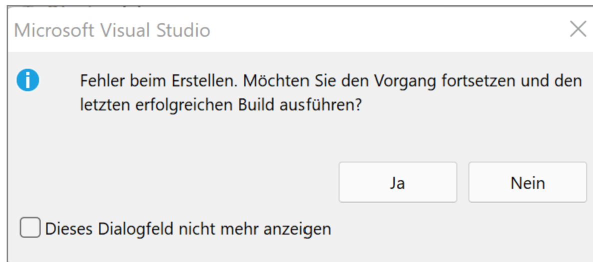


Bild 2.22 Fehler beim Erstellen

In der Regel sollten Sie *Nein* klicken, um unverzüglich den (oder die) Übeltäter im Quellcode zu suchen und dingfest zu machen. Das Lokalisieren ist meist sehr einfach, da der fehlerhafte Ausdruck durch eine Wellenlinie unterschlängelt ist. Auch hier erhalten Sie Hinweise zur Fehlerursache, wenn Sie mit der Maus auf die betreffende Stelle fahren.

HINWEIS: Wenn das Programm sich partout nicht kompilieren lässt und weit und breit keine Wellenlinie bzw. ein anderer Hinweis auf den Übeltäter in Sicht ist, hilft meist ein Blick in die Fehlerliste am unteren Rand des Hauptfensters (oder Menü *Ansicht/Fehlerliste*).

Bild 2.23 Fehlerliste

Auf einen typischen Fehler, der manchen Anfänger zur Verzweiflung bringen kann, soll Sie das folgende Beispiel hinweisen.

Durch einen versehentlichen Doppelklick auf *textBox1* haben Sie im Codefenster unbewusst einen Eventhandler für das *TextChanged*-Ereignis (das ist das Standardereignis für dieses Steuerelement) erzeugt. Sie haben das zwar sofort bemerkt und den Rahmencode des Eventhandlers gleich wieder gelöscht, aber durch diese Aktion haben Sie einen Compilerfehler erzeugt.

In früheren Versionen von Visual Studio hat dies dazu geführt, dass der Designer einen Fehler ausgegeben hat und das Formular im Designmodus nicht mehr dargestellt werden konnte. Visual Studio 2022 reagiert darauf nicht mehr so empfindlich.

Erst ein Blick auf die Fehlerliste bringt Sie auf die richtige Fährte, denn Sie haben zwar den Rahmencode des *textBox1_TextChanged*-Eventhandlers wieder gelöscht, nicht aber seine (von Visual Studio automatisch angelegte) Deklaration. Diese ist nach wie vor vorhanden und verweist (für Sie zunächst unsichtbar) auf den nun nicht mehr vorhandenen Code. Klicken Sie deshalb auf den Link *Gehe zu Code* (oder doppelklicken Sie auf den Eintrag in der Fehlerliste), worauf sich das Codefenster der Datei *Form1.Designer.cs* öffnet[9]:

```
46            this.textBox1.TabIndex = 0;
47            this.textBox1.Text = "textBox1";
48            this.textBox1.TextAlign = System.Windows.Forms.HorizontalAlignment.Right;
49            this.textBox1.TextChanged += new System.EventHandler(this.textBox1_TextChanged);
50            this.textBox1.KeyUp += new System.Windows.Forms.KeyEventHandler(this.textBox1_KeyUp);
51            //
52            // textBox2
53            //
54            this.textBox2.Location = new System.Drawing.Point(244, 102);
55            this.textBox2.Margin = new System.Windows.Forms.Padding(4);
56            this.textBox2.Name = "textBox2";
57            this.textBox2.Size = new System.Drawing.Size(156, 37);
58            this.textBox2.TabIndex = 1;
59            this.textBox2.Text = "textBox2";
60            this.textBox2.TextAlign = System.Windows.Forms.HorizontalAlignment.Right;
61            this.textBox2.KeyUp += new System.Windows.Forms.KeyEventHandler(this.textBox2_KeyUp);
62            //
63            // textBox3
64            //
65            this.textBox3.Location = new System.Drawing.Point(228, 24);
66            this.textBox3.Margin = new System.Windows.Forms.Padding(4);
67            this.textBox3.Name = "textBox3";
68            this.textBox3.Size = new System.Drawing.Size(118, 37);
```

0 % | 1 | 0 | Zeile: 50 | Zeichen: 1 | SPC

ehlerliste

Gesamte Projektmappe | 1 Fehler | 0 Warnungen | 0 von 4 Mitteilungen | Erstellen + IntelliSense | Fehlerliste durchsuchen

Code	Beschreibung	Projekt	Datei	Z...	Unterdrückungszustar
CS1061	"Form1" enthält keine Definition für "textBox1_TextChanged", und es konnte keine zugängliche textBox1_TextChanged-Erweiterungsmethode gefunden werden, die ein erstes Argument vom Typ "Form1" akzeptiert (möglicherweise fehlt eine using-Direktive oder ein Assemblyverweis).	EuroDollar	Form1.Designer.cs	49	Aktiv

Bild 2.24 Fehler im Designercode

Der Name des „falschen" Eventhandlers ist mit einer roten Linie unterschlängelt. Entfernen Sie die blau hinterlegte Zeile komplett.

HINWEIS: Weitere hilfreiche Hinweise zum Debuggen finden Sie in Kapitel 16.

[9] In dieser Datei haben Sie normalerweise nichts zu suchen, da die Einträge von Visual Studio vorgenommen werden.

2.6 Neuerungen in Visual Studio 2022 Version 17.8

Zum Abschluss dieses Kapitels wollen wir noch einen kleinen Überblick über die Neuerungen von Visual Studio in der aktuellsten Version zeigen.

Wenn Sie ein GitHub-Copilot-Abonnement haben, können Sie den Copilot Autocomplete als optionale Komponente in Visual Studio laden und sich über dessen Vorschläge und Codevervollständigungen freuen.

Auch PullRequests können jetzt direkt aus Visual Studio heraus durchgeführt werden.

Zum Vergleichen von Codeständen gibt es ein verbessertes Vergleichsfenster, das Ihnen übersichtlicher die durchgeführten Änderungen an einer Codedatei zeigt.

Ein wesentlicher Fokus in der neuen Version lag auch auf einer besseren Performance von Visual Studio.

Damit auch nicht SDK-Style-Projekte (in der Regel die klassischen .NET Framework-Projekte bis .NET 4.8) von den Performancevorteilen profitieren können, sollten Sie in eine Projektdatei folgenden Code einfügen:

```
<Project>
  <PropertyGroup>
    <AccelerateBuildsInVisualStudio>true</AccelerateBuildsInVisualStudio>
  </PropertyGroup>
</Project>
```

Bite beachten Sie dabei, dass das Attribut *Project* bereits in der Projektdatei gesetzt ist und es mehrere *PropertyGroups* dazu gibt. Am besten fügen Sie das Attribut *AccelerateBuildsIn VisualStudio* in jede *PropertyGroup* ein.

Eine ausführliche Beschreibung finden Sie unter *https://devblogs.microsoft.com/visualstudio/visual-studio-17-8-now-available/*.

3 Grundlagen der Sprache C#

In diesem Kapitel möchten wir Ihnen den für den Einstieg wichtigen Sprachkern von C# erklären[1]. Wir zeigen Ihnen, wie Sie Anweisungen schreiben, mit Datentypen umgehen, Schleifen und Verzweigungen einsetzen, Arrays definieren und Funktionen bzw. Prozeduren aufrufen. Mit Rücksicht auf den Einsteiger und um die Übersichtlichkeit nicht zu gefährden, folgen die etwas anspruchsvolleren Bestandteile von C# erst in späteren Kapiteln.

HINWEIS: Testen Sie möglichst viele der kleinen Codeschnipsel des vorliegenden Kapitels am eigenen Rechner auf Herz und Nieren. Als Codegerüst kann entweder eine Konsolenanwendung oder aber eine Windows-Forms-Anwendung dienen.

3.1 Grundbegriffe

So, dann schauen wir uns mal die grundlegendsten Dinge von C# an.

3.1.1 Anweisungen

Unter einer Anweisung wird ein Befehl verstanden, der eine bestimmte Aktion ausführt. Wie in jeder anderen Programmiersprache müssen auch die C#-Anweisungen bestimmten Regeln entsprechen, die man in ihrer Gesamtheit als *Syntax*[2] bezeichnet.

Eine der wichtigsten Syntaxregeln kennen Sie bereits aus den Einführungsbeispielen, nämlich dass jede Anweisung mit einem Semikolon abzuschließen ist und dass der Zeilenumbruch dabei keinerlei Rolle spielt.

[1] Der Profi, der bereits mit Java, C oder C++ gearbeitet hat, wird dieses Kapitel natürlich mit Siebenmeilenstiefeln durcheilen.

[2] Im Unterschied zur *Syntax* versteht man unter der *Semantik* einer Sprache die Beschreibung dessen, *was* die einzelnen Anweisungen bewirken.

Da C# eine sogenannte formatfreie Sprache ist, haben neben dem Zeilenumbruch auch Leerzeichen, Tabulatoren etc. keine Bedeutung, es sei denn, Sie verwenden sie bewusst, um die optische Lesbarkeit Ihres Codes zu verbessern.

Durch gute Strukturierung Ihres Quellcodes, wie z. B. blockweises Einrücken, machen Sie Ihre Programme übersichtlicher und verringern somit die Fehlerquote.

HINWEIS: Die Entwicklungsumgebung von Visual Studio erleichtert Ihnen das blockweise Einrücken unter anderem durch das Menü *Bearbeiten/Erweitert/ Zeileneinzug vergrößern* bzw. *verkleinern* oder durch die entsprechenden Schaltflächen der Symbolleiste. Beachten Sie, dass diese Menüpunkte nur sichtbar sind, wenn ein Codefenster das aktive Fenster in Visual Studio ist.

3.1.2 Bezeichner

Für die Namensgebung von Elementen Ihres Programms, wie Variablen, Methoden, Klassen etc. verwenden Sie Bezeichner. Bei der Namensgebung müssen Sie sich an folgende Regeln halten:

- Als Zeichen sind Groß- und Kleinbuchstaben, der Unterstrich „_“ sowie die Ziffern 0 … 9 zulässig.
- Jeder Bezeichner muss mit einem Buchstaben (oder einem Unterstrich) beginnen, Zahlen sind am Anfang nicht zulässig.
- Als case-sensitive Sprache unterscheidet C# penibel zwischen Groß- und Kleinschreibung.

Beispiel 3.1: Zulässige Bezeichner

C#

```
euro
_radius
zwerg7
```

Beispiel 3.2: Unzulässige Bezeichner

C#

```
%Anteil
7Zwerge
Gehalt$
```

HINWEIS: Bezüglich der Verwendung von Groß- und Kleinschreibung gibt es zwar keine Verbote, aber folgende Empfehlung: Verwenden Sie möglichst keine Bezeichner, die sich lediglich durch die Groß- und Kleinschreibung voneinander unterscheiden! Einzige Ausnahme, wo Sie das sogar tun sollten, ist bei Properties und den zugehörigen Feldbezeichnungen. Doch seit der Einführung von AutoProperties (siehe Kapitel 4) benötigen Sie das Feld nicht mehr. Schreiben Sie Elemente, die öffentlich (*public*) definiert werden, immer groß, *private*-Variablen dagegen klein.

Beispiel 3.3: Beide Bezeichner sollten nicht nebeneinander verwendet werden:

C#
```
MeineAdresse
meineAdresse
```

3.1.3 Schlüsselwörter

Bei Schlüsselwörtern handelt es sich um vordefinierte reservierte Bezeichner, die den Kern der Sprache C# ausmachen und die im Codefenster von Visual Studio normalerweise blau eingefärbt werden. Die folgende (nicht ganz vollständige) Übersicht zeigt die Schlüsselwörter von C#.

abstract	*as*	*async*	*await*	*base*	*bool*
break	*cyte*	*case*	*char*	*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	*var*	*virtual*	*Void*	*volatile*	*while*

HINWEIS: Schlüsselwörter dürfen Sie **nicht** für selbst definierte Bezeichner verwenden!

Allerdings gibt es zu obigem Hinweis eine gewisse Ausnahme: Wenn Schlüsselwörter ein @ als Präfix enthalten, können sie als Bezeichner im Programm verwendet werden.

Beispiel 3.4: Schlüsselwörter mit Präfix

C#

@*if* stellt z. B. einen gültigen Bezeichner dar, *if* jedoch nicht, da es sich um ein Schlüsselwort handelt. Die Verwendung finden wir persönlich jedoch fragwürdig.

3.1.4 Kommentare

Kommentare (im Editor standardmäßig grün eingefärbt) dienen als zusätzliche Erläuterungen für den Programmierer. Sie sollen die Lesbarkeit des Quellcodes verbessern. Für das Kennzeichnen von Kommentaren können Sie zwei unterschiedliche Verfahren verwenden.

Einzeilige Kommentare

Um eine Zeile (nicht Befehlszeile) als Kommentar zu markieren, leiten Sie diese mit einem doppelten Slash // ein.

Beispiel 3.5: Eine Anweisung mit Kommentar

C#

```
private double euro = 1.0;     // Variablendeklaration
```

HINWEIS: Geizen Sie in Ihren Quelltexten nicht mit Kommentaren, damit Sie (oder andere) auch später noch den von Ihnen geschriebenen Code verstehen können!

Mehrzeilige Kommentare

Um einen mehrzeiligen Bereich als Kommentar zu kennzeichnen, wird dieser mit /* und */ eingegrenzt.

Beispiel 3.6: Mehrzeiliger Kommentar

C#

```
/* Dieser Kommentar
besteht aus zwei Zeilen */
```

Mehrzeilige Kommentare können Sie auch vorteilhaft beim Testen von Code einsetzen, indem Sie (in der Regel nur vorübergehend) bestimmte Codeabschnitte außer Kraft setzen, d. h. „auskommentieren“.

HINWEIS: Die Visual-Studio-Entwicklungsumgebung erleichtert Ihnen das Auskommentieren von Codeabschnitten mit dem Menübefehl *Bearbeiten/Erweitert/Auswahl kommentieren* (**Strg**+**E**, **C**) bzw. *Auskommentierung der Auswahl aufheben* (**Strg**+**E**, **U**) oder mit den entsprechenden Schaltflächen der Symbolleiste .

3.2 Datentypen, Variablen und Konstanten

Jedes Programm „lebt“ in erster Linie von seinen Variablen und Konstanten, die bestimmten Datentypen entsprechen. Es ist daher logisch, dass wir dieses Thema an den Anfang unserer Betrachtungen zur Sprache C# stellen müssen.

3.2.1 Fundamentale Typen

Die folgende Tabelle gibt eine Übersicht der einfachen (fundamentalen) Datentypen, die C# zur Verfügung stellt[3].

Wie Sie der Tabelle entnehmen können, entsprechen alle C#-Datentypen einer Klassendefinition im .NET Framework. Die CLR[4]-Datentypen sind im *System*-Namensraum angeordnet. Bei der Deklaration (siehe unten) ist es egal, welchen der beiden möglichen Typbezeichner Sie angeben.[5]

C#-Datentyp	.NET-CLR-Typ	Erläuterung	Länge [Byte]5
byte	*System.Byte*	Positive Ganzzahl zwischen 0 ... 255	1
sbyte	*System.SByte*	Vorzeichenbehaftete Ganzzahl zwischen -128 ... 127	1
short	*System.Int16*	Kurze Ganzzahl zwischen -2^{15} ... $2^{15}-1$ (-32.768 ... 32.767)	2
ushort	*System.UInt16*	Vorzeichenlose Ganzzahl zwischen 0 ... 65 535	2
int	*System.Int32*	Ganzzahl zwischen -2^{31} ... $2^{31}-1$ (-2.147.483.648 ... 2.147.483.647)	4
uint	*System.UInt32*	Vorzeichenlose Ganzzahl zwischen 0 ... 4.294.967.295	4
ulong	*System.UInt64*	Vorzeichenlose Ganzzahl zwischen 0 ... 18.446.744.073.709.551.615	8
long	*System.Int64*	Lange Ganzzahl -2^{63} ... $2^{63}-1$ (-9.223.372.036.854.775.808 ... 9.223.372.036.854.775.807)	8
float	*System.Single*	Gleitkommazahl mit 7-stelliger Genauigkeit zwischen ca. +/- 3.4E-45 und +/- 3.4E+38	4
double	*System.Double*	Gleitkommazahl mit 16-stelliger Genauigkeit zwischen ca. +/- 4.9E-324 und +/- 1.8E+308	8

(Fortsetzung nächste Seite)

[3] Auf „anspruchsvollere“ Datentypen wie *var* oder *dynamic* gehen wir erst an späterer Stelle ein.
[4] Die .NET-Laufzeitumgebung *(Common Language Runtime)*
[5] Systemintern sind auch die ersten vier Typen 4 Byte lang.

(Fortsetzung)

C#-Datentyp	.NET-CLR-Typ	Erläuterung	Länge [Byte]5
decimal	*System.Decimal*	Gleitkommazahl zwischen 0 ... +/-79E+27 (ohne Dezimalpunkt) und ca. +/-1.0E-29 ...7.9E+27 (mit Dezimalpunkt)	16
char	*System.Char*	Ein beliebiges Unicode-Zeichen	2
bool	*System.Boolean*	Wahrheitswert (*true*, *false*)	2
string	*System.String*	Beliebige Unicode-Zeichenfolge mit einer maximalen Länge von ca. 2 000 000 000 Zeichen	2 pro Zeichen plus 10
object	*System.Object*	Universeller Datentyp	4 (auf 32 Bit) 8 (auf 64 Bit)

3.2.2 Wertetypen versus Verweistypen

Mit Ausnahme des *string*- und des *object*-Datentyps, die sogenannte *Verweistypen* sind, gehören die übrigen Datentypen in obiger Tabelle zu den *Wertetypen*. Das Verständnis des Unterschieds zwischen diesen beiden fundamentalen Gruppen ist enorm wichtig für das tiefere Eindringen in die Sprache C#.

Wertetypen

Dazu zählen all die einfachen Datentypen wie *byte*, *int*, *double* usw. Hinzu kommen später noch andere wie beispielsweise *struct* (siehe Abschnitt 3.6.2) und *DateTime* (siehe Abschnitt 5.3). Beim Abarbeiten des Programms wird für die lokalen Variablen und die übergebenen Parameter Speicherplatz benötigt, der immer dem Stack entnommen wird. Nach Beendigung einer Methode wird der Speicher automatisch an den Stack[6] zurückgegeben.

Verweistypen

Bislang kennen wir nur die Verweistypen *string* und *object*, in Kapitel 5 kommen später noch Datenfelder (Arrays) hinzu. Aber das ist nur die Spitze des Eisbergs, denn in der objektorientierten Programmierung, in die wir ab Kapitel 4 tiefer einsteigen werden, sind alle Objekte Verweistypen. Das bedeutet, dass auf dem Stack nicht der Wert des Objekts abgespeichert wird, sondern lediglich ein Verweis (Referenz, Zeiger) auf eine Speicheradresse des Heap, wo das eigentliche Objekt gespeichert ist. Das Anlegen des Objekts auf dem Heap wird auch als Instanziierung bezeichnet, in der Regel muss dazu der *new*-Operator verwendet werden[7]. Das Entsorgen des Speichers übernimmt sporadisch der sogenannte Garbage Collector. Doch zu all dem kommen wir erst im nachfolgenden OOP-Kapitel.

[6] Stack und Heap sind bestimmte Bereiche im Arbeitsspeicher jedes Computers.

[7] Der *string*-Datentyp bildet hier eine gewisse Ausnahme (siehe Abschnitt 5.2).

3.2.3 Benennung von Variablen

Variablen sind benannte Speicherplatzstellen. Der Variablenname dient dazu, die Speicheradresse im Programmcode quasi wie über einen Alias anzusprechen.

Zusätzlich zu den in Abschnitt 3.1.2 aufgeführten Regeln für selbst definierte Bezeichner sollten Sie folgenden Empfehlungen gemäß *Common Language Specification* (CLS) folgen:

- Beginnen Sie den Namen einer Variablen mit einem Kleinbuchstaben.
- Falls Bezeichner aus mehreren Wörtern zusammengesetzt sind, so sollten ab dem zweiten Wort alle Wörter mit einem Großbuchstaben (sogenanntes CamelCasing) beginnen.

Beispiel 3.7: Ein Variablenname, der sich aus mehreren Wörtern zusammensetzt

C#
```
haushaltskassenSaldo
```

3.2.4 Deklaration von Variablen

Variablen werden in C# wie folgt deklariert:

Syntax:

```
Datentyp variablenname;
```

Beispiel 3.8: Drei Variablen unterschiedlichen Datentyps werden deklariert.

C#
```
int anzahl;
double breite;
string nachName;
```

Wollen Sie mehrere Variablen vom gleichen Datentyp deklarieren, so können diese durch Kommas separiert werden. Sie können die Variablen natürlich auch einzeln in eigenen Zeilen definieren.

Beispiel 3.9: Drei *int*-Variablen werden deklariert.

C#
```
int i, j, k;
```

Als Datentyp kann man auch den CLR-Typ angeben (siehe obige Tabelle). Visual Studio wird Ihnen dann aber sofort vorschlagen, den C#-Datentyp zu verwenden.

Beispiel 3.10: Die folgenden drei Deklarationen sind gleichwertig.

```
C#
int i;
System.Int32 i;
Int32 i;
```

Initialisierte Variablen

Zusammen mit der Deklaration können Sie den Variablen auch gleich Anfangswerte zuweisen (im Fachjargon heißt das „initialisieren“).

Beispiel 3.11: Initialisierte Variablen

```
C#
int anzahl;
anzahl = 99;
```

Stattdessen können Sie kürzer formulieren:

```
int anzahl = 99;
```

3.2.5 Typsuffixe

Wenn Sie Variablen im Quellcode direkte Zahlenwerte (Literale) zuweisen wollen, so werden diese vom Compiler standardmäßig als Datentyp *int* bzw. *double* interpretiert. Bei Datentypen wie *long*, *float* oder *decimal* müssen Sie ein sogenanntes Typsuffix (*L*, *F*, *M*) anhängen, ansonsten werden die Literale wie *int* oder *double* behandelt und es gibt einen Compilerfehler. Eine Übersicht enthält die folgende Tabelle.

Typsuffix	Gleitkommatyp
f oder *F*	*Float*
L	*Long*
m oder *M*	*Decimal*

Beispiel 3.12: Richtige und falsche Literalzuweisungen

```
C#
float max = 99.99;          // Fehler!
float max = 99.99F;         // richtig; 99.99f wäre auch korrekt
decimal geld = 300.50;      // Fehler!
decimal geld = 300.50M;     // richtig; 300.50m wäre auch korrekt
```

3.2.6 Zeichen und Zeichenketten

Die Datentypen *char* und *string* basieren auf dem Unicode-Zeichensatz, der pro Zeichen 2 Byte beansprucht (im Unterschied zum klassischen ASCII- bzw. ANSI-Zeichensatz mit 1 Byte pro Zeichen). Mit dem Unicode können deshalb nicht mehr nur maximal 255, sondern bis zu 65 535 (!) verschiedene Zeichen gespeichert werden.

char

Variablen vom *char*-Datentyp können Sie Zeichenliterale, hexadezimale Escape-Sequenzen oder Unicode-Darstellungen zuweisen.

HINWEIS: *char*-Literale sind in Hochkommata (') einzufassen.

Beispiel 3.13: Char

C#

Drei gleichwertige Anweisungen deklarieren eine *char*-Variable und initialisieren diese mit dem Zeichen *A*:

```
char c = 'A';           // Zeichenliteral
char c = '\x0041';      // hexadezimal
char c = '\u0041';      // Unicode
```

Als weitere Möglichkeit kommt eine explizite Typkonvertierung direkt aus dem (ganzzahligen) Zeichencode in Betracht.

Beispiel 3.14: Eine weitere Ergänzung zum Vorgängerbeispiel

C#

```
char c = (char)65;    // Typecasting liefert 'A'
```

string

HINWEIS: *string*-Literale sind in doppelte Hochkommata (") einzufassen.

Beispiel 3.15: Zuweisen einer Stringvariablen

C#

```
string s = "Hallo";
```

Einen einzelnen *char* können Sie direkt aus einem *string* herauskopieren, indem Sie den Index in eckige Klammern schreiben. Dabei hat das erste Zeichen den Index 0.

Beispiel 3.16: Das erste Zeichen eines Strings ermitteln

C#

```
string s = "Hallo";
char c = s[0];            // liefert "H"
```

HINWEIS: Ausführliches zur Stringverarbeitung finden Sie in Kapitel 5.

Der umgekehrte Schrägstrich bzw. Backslash (\) spielt innerhalb eines Strings eine besondere Rolle, denn nachfolgende Zeichen werden vom C#-Compiler als Befehl interpretiert[8].

Die folgende Tabelle zeigt die häufigsten in C# benutzten Escape-Sequenzen.

Escape-Sequenz	Bedeutung
\'	Einfaches Anführungszeichen
\"	Doppeltes Anführungszeichen
\\	Backslash
\a	Signalton
\b	Backspace (BS)
\f	Seitenvorschub
\n	Neue Zeile (LF)
\r	Wagenrücklauf CR)
\t	Horizontaler Tabulator (TAB)

HINWEIS: Bei einem Unicode-Zeichen folgt dem Backslash ein kleines „u" und die vierstellige Nummer des Zeichens, z. B. 'u0013'.

Beispiel 3.17: Korrekte Schreibweise für Dateipfade

C#

Die Anweisung zur Definition eines Dateipfads

```
string pfad = "C:\Benutzer\Juergen";
```

... wird bereits von der Entwicklungsumgebung als „nicht erkannte Escape-Sequenz" zurückgewiesen, da der Backslash als Beginn einer Escape-Sequenz interpretiert wird.

Die folgende Anweisung wäre korrekt:

```
string pfad = "C:\\Benutzer\\Juergen";
```

Oder in diesem Fall auch die Kurzform:

```
string pfad = @"C:\Benutzer\Juergen";
```

---

[8] Siehe dazu auch das Onlinekapitel „Reguläre Ausdrücke".

Beispiel 3.18: Hinzufügen eines Zeilenvorschubs

C#

```
MessageBox.Show("Beste Grüße von ...\r\nJürgen",
"Escape-Sequenzen", MessageBoxButtons.OK, MessageBoxIcon.Information);
```

Ergebnis

Folgendes Meldungsfenster erscheint:

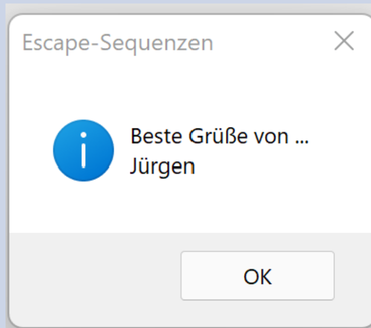


Bild 3.1
Messagebox mit Zeilenumbruch

3.2.7 object-Datentyp

Der *object*-Datentyp ist weitaus mehr, als es der Name vermuten lässt. Alle Klassen des .NET Frameworks sind von *System.Object* abgeleitet und *object* ist dafür lediglich ein Alias.

Eine *object*-Variable ist ein sogenannter Verweis- oder Referenztyp, denn sie speichert nicht den tatsächlichen Wert, sondern lediglich einen 8 Byte großen Zeiger auf die Adresse der zugewiesenen Variablen (oder 4 Byte bei einer 32-Bit-Anwendung).

Variablen des Typs *object* können Sie alles zuweisen.

Beispiel 3.19: Einer Objektvariablen *o* wird eine *int*-Zahl zugewiesen.

C#

```
int i = 5;
object o = i;
```

Etwas komplizierter ist der umgekehrte Weg, nämlich wenn Sie den Wert der Objektvariablen einer anderen Variablen zuweisen wollen. In diesem Fall ist man auf explizite Typumwandlung (Typecasting) angewiesen (siehe Abschnitt 3.3.1). Eine direkte Zuweisung (implizite Typkonvertierung) führt zu einem Compilerfehler.

Beispiel 3.20: Die Fortsetzung des Vorgängerbeispiels

C#

```
int j = o;           // Fehler!
int j = (int)o;      // mit Typecasting o.k.!
```

Wichtig in diesem Zusammenhang ist das Verständnis des Unterschieds zwischen Werte- und Referenztypen und des Prinzips des Boxing/Unboxing (siehe Abschnitt 3.3.6).

3.2.8 Konstanten deklarieren

Im Unterschied zu Variablen bleibt der Wert einer Konstante während der gesamten Laufzeit eines Programms unverändert. Sie legen ihn einmalig mit dem *const*-Schlüsselwort fest. Die Deklaration ist ähnlich wie bei initialisierten Variablen.

Syntax:

```
const Datentyp Konstantenname = Wert;
```

Beispiel 3.21: Verschiedene Konstantendeklarationen

```
C#
const int C = 119;
const float PI  = 3.1415F;
const double X1  = 3 * 0.4, X2 = 5.3 + 0.68;
const string S = "Hallo";
```

3.2.9 Nullable Types

C# erfordert seit eh und je die explizite Initialisierung von Variablen.

Beispiel 3.22: Falsche und richtige Verwendung von Variablen

```
C#
int z;
z++;                // falsch, weil z nicht initialisiert ist
...
int z = 0;
z++;                // richtig
```

Initialisieren von Wertetypen mit null

Etwas komplizierter wird es aber, wenn man einen Wert mit „nichts" (*null*) initialisieren möchte.

Beispiel 3.23: Das funktioniert nicht!

```
C#
int z = null;        // falsch
```

Durch ein der Typdeklaration nachgestelltes Fragezeichen (*?*) kann der Compiler einen Wertetyp in die generische *System.Nullable*-Struktur verpacken, wodurch es möglich wird, einer Variablen den Wert *null* zuzuweisen.

Beispiel 3.24: Aber das funktioniert!

C#

```
int? z = null;
z = 1;
```

Oder als explizite Deklaration:

```
System.Nullable<Int32> z = null;
z = 1;
```

Zuweisungen mit dem ??-Operator

Um einen *Nullable Type* (werteloser Typ) einer anderen Variablen zuweisen zu können, muss vorher eine *null*-Abfrage erfolgen, wie sie mittels *HasValue*-Eigenschaft möglich ist.

Beispiel 3.25: Die Variable *y* wird mit der Zahl 0 initialisiert, da *x* den Wert *null* hat.

C#

```
int? x = null;
int y;
if (x.HasValue)
{
  y = x.Value;
}
else
{
  y = 0;
}
```

Deutlich eleganter und kürzer ist eine solche Zuweisung aber, wenn man dazu das doppelte Fragezeichen (*??*), den sogenannten NULL-Zusammenführungsoperator oder im Englischen null-coalescing operator, verwendet. Er liefert den Wert des vorangestellten Ausdrucks, falls dieser nicht *null* ist, andernfalls den Wert des nachfolgenden Ausdrucks.

Beispiel 3.26: Das Vorgängerbeispiel wird einfacher realisiert.

C#

```
int? x = null;
int y = x ?? 0;
```

Beispiel 3.27: Im *Label* erscheint der Text „nichts zugewiesen“.

C#

```
string s = null;
label1.Text = s ?? "nichts zugewiesen!";
```

3.2.10 Typinferenz

Dieses in Zusammenhang mit der LINQ-Technologie (siehe Kapitel 7) eingeführte Sprachfeature erlaubt es, dass der Datentyp einer Variablen bei der Deklaration vom Compiler automatisch ermittelt wird, ohne dass Sie explizit den Typ angeben müssen. Als Ersatz für einen konkreten Typ wird das Schlüsselwort *var* verwendet.

HINWEIS: Damit der Compiler den Typ der Variablen feststellen kann, muss eine mit *var* deklarierte Variable unbedingt bei der Deklaration initialisiert werden.

Beispiel 3.28: *var*-Deklaration

C#

Die Initialisierung der Variablen *a* wird vom Compiler ausgewertet und der Typ wird aufgrund des Werts 35 auf *Integer* festgelegt.

```
var a = 35;
```

Obige Zeile ist semantisch identisch mit folgendem Ausdruck:

```
int a = 35;
```

Der Datentyp wird einmalig bei der ersten Deklaration der Variablen vom Compiler festgelegt und kann danach nicht mehr verändert werden.

Beispiel 3.29: Keine Änderung möglich!

C#

Da die Variable *b* vom Compiler als *Integer* festgelegt wurde, kann ihr später kein *double*-Wert zugewiesen werden.

```
var b = 7;
b = 12.3;            // Fehler!
```

HINWEIS: Typinferenz funktioniert nur bei lokalen Variablen (siehe Abschnitt 3.2.11)!

3.2.11 Gültigkeitsbereiche und Sichtbarkeit

Obwohl sich in C# alles innerhalb von Klassen abspielt, werden wir erst im OOP-Kapitel 4 ausführlicher auf diese Thematik zu sprechen kommen.

Trotzdem sollten Sie bereits jetzt Folgendes wissen:

- Lokale Variablen gelten standardmäßig nur innerhalb ihres – in geschweiften Klammern eingerahmten – Bereichs und der untergeordneten Bereiche. Ein Zugriff von außerhalb ist nicht möglich. Lokale Variablen sollten kleingeschrieben werden.

- Sie können die Schlüsselwörter *private* und *public* verwenden, um festzulegen, ob auf die Variablen auch von außerhalb zugegriffen werden kann.
- Wenn Sie eine Variable nicht als *public* oder *private* deklarieren, ist sie standardmäßig *private*. Man bezeichnet die Schlüsselwörter *private* und *public* auch als *Zugriffsmodifizierer*. Sie gelten nicht nur für Variablen, sondern auch für Klassen, Objekte, Eigenschaften und Methoden (siehe Kapitel 4, Abschnitt 4.1.3).

3.3 Konvertieren von Datentypen

C# ist eine typsichere Sprache und nimmt es deshalb mit den Datentypen sehr genau. Schon bei den geringsten Nachlässigkeiten schlagen Ihnen Visual Studio oder der Compiler gnadenlos auf die Finger.

3.3.1 Implizite und explizite Konvertierung

Unabhängig vom tatsächlichen Wert, wie er in der Variablen gespeichert ist, lassen sich verschiedene Datentypen nur dann gegenseitig zuweisen, wenn der Wertebereich des rechten Datentyps in den linken „passt". In einem solchen Fall findet eine sogenannte *implizite Konvertierung* statt, die der Compiler automatisch vornimmt.

Beispiel 3.30: Die Zuweisung *Byte* zu *Integer* funktioniert problemlos.

C#

```
byte b = 100;
int i = b;                // implizite Typkonvertierung
```

Geradezu oberlehrerhaft verhält sich C# im umgekehrten Fall. Egal, ob der Wert in den kleineren Datentyp passt oder nicht – es wird halt gemeckert.

Beispiel 3.31: Das geht nicht.

C#

Obwohl der Wert 100 problemlos in eine *Byte*-Variable passt, erscheint die Fehlermeldung „Implizite Konvertierung des Typs 'int' zu 'byte' nicht möglich!".

```
int i = 100;
byte b = i;               // Fehler!
```

Um den meckernden Compiler zu beschwichtigen, ist eine sogenannte *explizite Typkonvertierung* (auch *Typecasting* genannt) erforderlich.

Syntax:

```
neueVariable = (Neuer Datentyp)alteVariable;
```

Beispiel 3.32: Das Vorgängerbeispiel wird fehlerfrei ausgeführt.

```
C#
int i = 100;
byte b = (byte)i;       // explizite Typkonvertierung
```

Implizite Konvertierungen sind sicher, Datenverluste sind deshalb ausgeschlossen. Dabei kann stets nur der kleinere der beiden Datentypen direkt in einen größeren umgewandelt werden[9].

Explizite Typkonvertierungen sollten stets mit Vorsicht angewendet werden, wobei man sicher sein muss, dass die Wertebereiche zur Laufzeit nicht überschritten werden.

HINWEIS: Man muss sich immer bewusst sein, dass eine explizite Typkonvertierung dann zu Datenverlusten führen kann, wenn der Wertebereich durch die Konvertierung verkleinert wird.

Beispiel 3.33: Da der *byte*-Datentyp nur den Bereich 0 … 255 abdeckt, entsteht ein falsches Ergebnis.

```
C#
int i = 300;
byte b = (byte)i;
// liefert falsches Resultat (44)
```

Beispiel 3.34: Implizite und explizite Typkonvertierung *double* in *int* gegenübergestellt

```
C#
int i;
double d = 12.5;
i = d;               // implizite Konvertierung ergibt Fehler!
i = (int)d;          // explizite Konvertierung ergibt Datenverlust:
                     // i erhält den Wert 12
```

Beispiel 3.35: Implizite Typkonvertierung *char* in *int*

```
C#
char c = 'A';
int i = c;                // i erhält den Wert 65 (Zeichencode von 'A')
```

Beispiel 3.36: Explizite Typkonvertierung des Ergebnisses einer Division

```
C#
int i = 3;
double x = i/10;             // x erhält den Wert 0
double x = (double)i/10;     // x erhält den Wert 0,3
double x = i/10.0;     // x erhält den Wert 0,3
```

[9] Sie können sich das bildlich so vorstellen, dass jeder Datentyp einem Kochtopf mit unterschiedlichem Fassungsvermögen entspricht, und Sie dürfen immer nur etwas aus einem kleineren in einen größeren Topf füllen.

In diesem Beispiel haben wir bei der ersten Division zwei Ganzzahlen dividiert. Deshalb ist eine Ganzzahldivision durchgeführt worden. Sobald einer der beiden Werte (egal ob Zähler oder Nenner) aber eine Fließkommazahl ist, wird eine Fließkommazahlendivision durchgeführt.

as-Konvertierungsoperator

Eine Alternative zur expliziten Typumwandlung mittels *()*-Konvertierung ist der *as*-Operator, der allerdings nur auf Verweis- und nicht auf Wertetypen anwendbar ist. Auch alle Steuerelemente gehören zu den Verweistypen!

Beispiel 3.37: Konvertieren des *sender*-Parameters eines Eventhandlers

C#

```
Text = (sender as TextBox).Text;
```

3.3.2 Welcher Datentyp passt zu welchem?

Der folgenden Tabelle entnehmen Sie alle möglichen impliziten und expliziten Typkonvertierungen. Die impliziten Konvertierungen sind fett hervorgehoben.

Quell-Datentyp	Zieldatentypen
bool	*Object*
byte	*ushort, short, uint, int, ulong, long, float, double, decimal, object, sbyte, char*
sbyte	***short, int, long, float, double, decimal, object,*** *byte, ushort, uint, ulong, char*
short	***int, long, float, double, decimal, object,*** *sbyte, byte, ushort, uint, ulong, char*
ushort	***uint, int, ulong, long, float, double, decimal, object,*** *sbyte, byte, short, char*
char	***ushort, uint, int, ulong, long, float, double, decimal, object,*** *sbyte, byte, short*
int	***long, float, double, decimal, object****, sbyte, byte, short, ushort, uint, ulong, char*
uint	***long, float, double, decimal, object,*** *sbyte, byte, short, ushort, int, char*
long	***float, double, decimal, object,*** *sbyte, byte, short, ushort, int, uint, ulong, char*
ulong	***float, double, decimal, object,*** *sbyte, byte, short, ushort, int, uint, long, char*
float	***double, object,*** *sbyte, byte, short, ushort, int, uint, long, ulong, char, decimal*
double	***object,*** *sbyte, byte, short, ushort, int, uint, long, ulong, char, float, decimal*
decimal	***object,*** *sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double*
string	*Object*
object	alle Datentypen (Unboxing, siehe Abschnitt 3.3.6)

3.3.3 Konvertieren von string

Laut obiger Konvertierungstabelle lässt sich der *string*-Datentyp nur in den universellen *object*-Datentyp umwandeln und in umgekehrter Richtung scheint gar nichts zu gehen. Doch die Entwarnung folgt sogleich.

ToString-Methode

Der *object*-Datentyp – gewissermaßen die „Mutter aller Objekte" – vererbt an alle Nachkommen die *ToString()*-Methode, auf die Sie bereits hin und wieder in den bisherigen Beispielen gestoßen sind, nämlich dann, wenn es darum ging, Zahlenwerte zur Anzeige zu bringen.

HINWEIS: Jeder Datentyp kann mittels seiner *ToString()*-Methode in den Datentyp *string* umgewandelt werden!

Und noch ein Hinweis, der besonders für Umsteiger von Visual Basic wichtig ist:

HINWEIS: Vergessen Sie nicht die Klammern hinter *ToString()*!

Beispiel 3.38: Anzeige einer Gleitkommazahl

C#

```
double d = 12.75;
MessageBox.Show(d.ToString);      // Fehler
MessageBox.Show(d.ToString());    // ok
```

Beispiel 3.39: Konvertieren eines *bool* in *string*

C#

```
bool b = true;
string s = b.ToString();          // "True"
```

String in Zahl verwandeln

Zwar können wir mit der *ToString()*-Methode alle Datentypen in den *string*-Typ konvertieren, wie aber sieht es umgekehrt aus?

Für bestimmte andere Datentypen gibt es spezifische Lösungen, z. B. zum Umwandeln von *string* in *char*.

Beispiel 3.40: Einem *char* wird das zweite Zeichen eines *string* zugewiesen.

C#

```
string name = "Maria";
char c = name[1];
MessageBox.Show(c.ToString());      // zeigt "a"
```

Damit enden vorerst unsere Erfolgserlebnisse, denn das übliche Typecasting scheint bei den anderen Datentypen zu versagen.

Beispiel 3.41: Das geht leider nicht.

C#

```
string s = "5";
int i = (int)s;            // Fehler!
```

Rettung naht auch hier in Gestalt der *Convert*-Klasse (siehe auch den nächsten Abschnitt 3.3.4). Als Alternative zu den expliziten Typkonvertierungen bietet diese Klasse für nahezu jeden einfachen Datentyp eine spezielle (statische) Konvertierungsmethode, der man den zu konvertierenden Ausdruck als Argument übergibt.

Beispiel 3.42: Das Vorgängerbeispiel kann wie folgt gelöst werden.

C#

```
string s = "5 ";
int i = Convert.ToInt32(s);
MessageBox.Show(i.ToString());      // zeigt "5"
```

3.3.4 Die Convert-Klasse

Diese statische Klasse bietet für jeden einfachen Datentyp eine spezielle Konvertierungsmethode, der man den zu konvertierenden Ausdruck als Argument übergibt.

Syntax:

```
Convert.typMethode(object expr);
```

typeMethode = Konvertierungsmethode (*ToBoolean*, *ToByte*, *ToInt32*, *ToDouble* ...)

expr = zu konvertierender Ausdruck

Beispiel 3.43: *string* wird in *double* konvertiert.

C#

```
string s = "55,7";
double d = Convert.ToDouble(s);         // 55,7
```

Alternativ kann auch die *Parse*-Methode (siehe den nächsten Abschnitt 3.3.5) eingesetzt werden.

Beispiel 3.44: Konvertieren eines Stringliterals in eine Ganzzahl

C#

```
int nr = Int32.Parse("12");
```

Beispiel 3.45: *bool* wird in *int* und in *string* konvertiert.

```
C#
bool b = true;
int i = Convert.ToInt32(b);           // 1
b = false;
i = Convert.ToInt32(b);               // 0
string s = Convert.ToString(b);       // "False"
```

EVA-Prinzip

Auch für (fast) jedes Programm gilt nach wie vor das uralte EVA-Prinzip (Eingabe, Verarbeitung, Ausgabe). In diesem Zusammenhang sei nochmals auf die besondere Bedeutung der Typkonvertierung von und in den *string*-Datentyp hingewiesen. Da unter Windows sehr häufig die Übergabewerte als Zeichenketten vorliegen (*Text*-Eigenschaft der Ein- und Ausgabefelder), müssen sie zunächst in Zahlentypen umgewandelt werden, um dann nach ihrer Verarbeitung wieder in Zeichenketten rückverwandelt und (formatiert) zur Anzeige gebracht zu werden.

Beispiel 3.46: Ein Ausschnitt aus dem Einführungsbeispiel in Abschnitt 2.5.2

```
C#
euro = Convert.ToDouble(textBox1.Text);       // Eingabe: string => double
dollar = euro * kurs;                         // Verarbeitung
textBox2.Text = dollar.ToString("#,##0.00");  // Ausgabe: double => string
```

3.3.5 Die Parse-Methode

Die numerischen Typen *Byte* (*byte*), *Int16* (*short*), *Int32* (*int*), *Int64* (*long*), *Single* (*float*) und *Double* (*double*) verfügen u. a. über die (statische) *Parse*-Methode, welche die Stringdarstellung einer Zahl in den entsprechenden Typ konvertieren kann.

Beispiel 3.47: Der Inhalt einer *TextBox* wird in eine Gleitkommazahl konvertiert.

```
C#
double z = double.Parse(textBox1.Text);
```

HINWEIS: Die *Parse*-Methode hat den Vorteil, dass zusätzlich Kulturinformationen eines bestimmten Landes mit übergeben werden können.

3.3.6 Boxing und Unboxing

Die Begriffe *Boxing/Unboxing* gehören zu den häufig strapazierten .NET-Schlagwörtern, die manchem Einsteiger Ehrfurcht einflößen. Was verbirgt sich dahinter? Sie wissen bis jetzt, dass Sie dem universellen *object*-Datentyp jeden Wert direkt zuweisen können, d. h. durch implizite Typkonvertierung. Umgekehrt kann, falls es der *object*-Inhalt erlaubt, jeder Datentyp durch explizite Typkonvertierung (Typecasting) aus *object* wieder „herausgezogen" werden. Das direkte Zuweisen funktioniert in diesem Fall nicht.

Beispiel 3.48: Boxing und Unboxing

C#

Eine *bool*-Variable wird in ein *object* „verpackt" (Boxing) und dieses wird anschließend einer zweiten *bool*-Variablen zugewiesen (Unboxing).

```
bool b1 = true;
object o = b1;        // ok, implizite Konvertierung (Boxing)
bool b2 = o;          // Fehler, implizite Konvertierung
bool b2 = (bool)o;    // ok, explizite Konvertierung (Unboxing, b2 ist true)
```

Um den tieferen Sinn von Boxing/Unboxing zu verstehen, sollten Sie sich nochmals den Unterschied zwischen den beiden fundamentalen Arten von Datentypen vergegenwärtigen, d. h. zwischen den Wertetypen und den Verweis- bzw. Referenztypen (siehe Abschnitt 3.2.2).

Boxing

Es stellt sich nun die Frage, was denn passiert, wenn man einer *object*-Variablen, d. h. einem Verweistyp, einen Wertetyp zuweist, der naturgemäß im Stack gespeichert ist.

Beispiel 3.49: Ein Integer wird einem *object*-Datentyp zugewiesen.

C#

```
int i = 25;
object o = i;
```

Die genauere Fragestellung ist: Worauf zeigt die *object*-Variable *o*? Der Zeiger *o* darf doch keinesfalls auf den Stack verweisen (das würde die Stabilität des Programms massiv gefährden)!

Die Antwort: Es findet ein automatischer Kopiervorgang statt, d. h., eine Kopie der Variablen *i* wird auf dem Heap abgelegt, auf die dann die *object*-Variable *o* zeigt.

Unboxing

Wie greift man nun aber wieder auf den in der *object*-Variablen „eingepackten" Wert zu? Eine einfache (implizite) Zuweisung funktioniert nicht. Richtig ist eine explizite Typkonvertierung (Typecasting).

Beispiel 3.50: Das Vorgängerbeispiel wird fortgesetzt.

C#

```
int j = o;                    // Fehler!
int j = (int)o;               // ok
```

Allerdings funktioniert das Typecasting nur dann, wenn die Objektvariable tatsächlich auf den gewünschten Typ verweist, Trickserei – wie im folgenden Beispiel – nützt also nichts.

Beispiel 3.51: Das geht nicht!

C#

Die Hoffnung, bei der Umwandlung *string* nach *int* vielleicht ohne *Convert*-Klasse (siehe oben) auszukommen, geht leider nicht in Erfüllung.

```
string s = "5";
object o = s;
int i = (int)o;    // Fehler!
```

HINWEIS: Das Boxing ist mit ein wesentlicher Grund, warum in .NET „alles ein Objekt" ist, denn auch Wertetypen können damit quasi wie Objekte behandelt werden.

Beispiel 3.52: Ja, auch das funktioniert!

C#

```
int i = new int();
i = 12;
```

Der Wert von *i* nach der Instanziierung wäre 0.

3.4 Operatoren

Operatoren verknüpfen Variablen bzw. Operanden miteinander und führen Berechnungen durch. Wir unterscheiden zwischen

- arithmetischen Operatoren,
- Zuweisungsoperatoren,
- logischen Operatoren und
- Vergleichsoperatoren.

Die meisten Operatoren in C# benötigen zwei Operanden.

Beispiel 3.53: Operanden

C#

Im Ausdruck

```
i = 12;
```

ist der *Operator* das Gleichheitszeichen (=), die beiden *Operanden* sind die Variable *i* und die Literalkonstante *12*.

HINWEIS: C# erlaubt auch das Überladen von Operatoren, auf das wir aber erst an späterer Stelle eingehen wollen (siehe Kapitel 6).

3.4.1 Arithmetische Operatoren

In diesem Abschnitt wollen wir einige Operatoren betrachten.

Standard-Operatoren

Es gibt zunächst die üblichen Operatoren für die Grundrechenarten:

Operator	Beispielausdruck	Erklärung
+	x + y	Addition
-	x - y	Subtraktion
*	x * y	Multiplikation
/	x / y	Division
%	x % y	Modulo-Division (liefert Restwert)

Beispiel 3.54: Standard-Operatoren

C#

```
int i;
int j = 6;
i = 3 *(4 + 5) * j;     // 162
i = 7 % 3;              // 1 (Rest!)
```

HINWEIS: Achten Sie bei der Division von Literalen darauf, dass das Ergebnis abgerundet wird, wenn nicht mindestens einer der Operanden als Fließkommazahl gekennzeichnet ist.

Beispiel 3.55: Nur die beiden letzten Divisionen liefern das exakte Ergebnis.

```
C#
double d;
d = 7 / 3;              // 2   (Ergebnis wird abgerundet!)
d = 7D / 3;             // 2,3333333333333
d = 7.0 / 3;            //       dto.
```

Inkrement- und Dekrement-Operatoren

Mit den Kurz-Operatoren ++ und -- lässt sich das schrittweise Erhöhen (Inkrementieren) bzw. Erniedrigen (Dekrementieren) von Variablen vereinfachen.

Operator	Beispielausdruck	Erklärung
++	x++	Postfix-Inkrement
	++x	Präfix-Inkrement
--	x--	Postfix-Dekrement
	--x	Präfix-Dekrement

Wie Sie den Beispielen in obiger Tabelle entnehmen, können Sie die Kurz-Operatoren ++ und -- nicht nur hinter den Namen der Variablen (Postfix), sondern auch davor (Präfix) schreiben.

Beispiel 3.56: Postfix-Inkrement und Postfix-Dekrement

```
C#
int i = 10;
i++;                    // i erhält den Wert 11
double d = 2.5;
d--;                    // d erhält den Wert 1,5
```

Beispiel 3.57: Äquivalente Version des Vorgängerbeispiels mit Präfixoperationen

```
C#
int i = 10;
++i;                    // i erhält den Wert 11
double d = 2.5;
--d;                    // d erhält den Wert 1,5
```

Wie Sie sehen, haben beide Schreibweisen keinerlei Einfluss auf den Wert der Variablen. Diese wird in jedem Fall um 1 inkrementiert bzw. dekrementiert, wozu also sollen Postfix- und Präfix-Notationen dann gut sein?

Um den „feinen" Unterschied zu verstehen, muss man wissen, dass nicht nur die Variable (z. B. *i*) einem bestimmten Wert entspricht, sondern auch die mit dem Kurz-Operator verknüpfte Variable (z. B. *i*++). Letztere hat bei einer Postfix-Operation den Wert **vor** der Inkrementierung bzw. Dekrementierung, bei einer Präfix-Operation hingegen den Wert **nach** der Inkrementierung bzw. Dekrementierung.