

HANSER



Leseprobe

zu

C# und .NET 8

von Jürgen Kotz und Christian Wenz

Print-ISBN: 978-3-446-47982-1

E-Book-ISBN: 978-3-446-48060-5

E-Pub-ISBN: 978-3-446-48178-7

Weitere Informationen und Bestellungen unter

<https://www.hanser-kundencenter.de/fachbuch/artikel/9783446479821>

sowie im Buchhandel

© Carl Hanser Verlag, München

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	Werttypen 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 instanziiieren	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	Samlungsausdrücke	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 instanziiieren	604
12.2.2	Verwenden der Instanz im C#-Quellcode	605
12.2.3	Anforderungen an die Quell-Klasse	606
12.2.4	Instanziiieren 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 <code>IAsyncEnumerable<T></code>	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 <code>Parallel.Invoke</code>	734
15.2.1	Aufrufvarianten	734
15.2.2	Einschränkungen	736
15.3	Verwendung von <code>Parallel.For</code>	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	<code>Collections</code> mit <code>Parallel.ForEach</code> 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 <code>Collections</code>	762
15.7.2	Primitive für die Threadsynchronisation	762
15.8	Parallel LINQ (PLINQ)	763
15.9	Praxisbeispiele	763
15.9.1	<code>BlockingCollection</code>	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:

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.

7

Einführung in LINQ

LINQ (Language Integrated Query) ist ein C#-Sprachfeature, das mit C# 3.0 eingeführt wurde. Bei *LINQ to Objects* handelt es sich um die allgemeinste und grundlegendste LINQ-Implementierung, die auch die wichtigsten Bausteine für die übrigen LINQ-Implementierungen liefert. In einer SQL-ähnlichen Syntax können miteinander verknüpfte Collections/Auflistungen abgefragt werden, die die *IEnumerable*-Schnittstelle implementieren.

■ 7.1 LINQ-Grundlagen

Der wichtigste Vorteil von LINQ ist die standardisierte Möglichkeit, nicht nur Tabellen in einer relationalen Datenbank, sondern auch Textdateien, XML-Dateien und andere Datenquellen in einer SQL-ähnlichen Syntax abzufragen. Ein zweiter Vorteil ist die Fähigkeit, diese standardisierten Methoden von jeder .NET-konformen Sprache (wie zum Beispiel C# oder VB) aus aufrufen zu können.

7.1.1 Die LINQ-Architektur

Die folgende Abbildung soll die grundsätzliche Architektur von LINQ verdeutlichen.

Je nach Standort des Betrachters besteht LINQ einerseits aus einer Menge von Werkzeugen zur Arbeit mit Daten, was in den verschiedenen LINQ-Implementationen (LINQ to Objects, LINQ to DataSets, LINQ to Entities und LINQ to XML) zum Ausdruck kommt. Andererseits besteht LINQ aus einer Menge von Spracherweiterungen.

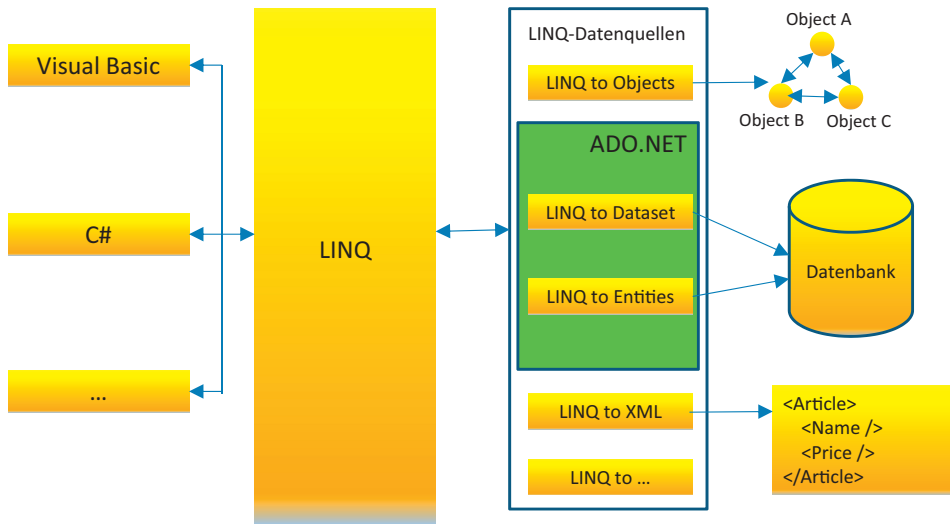


Bild 7.1 LINQ-Architektur

LINQ-Implementierungen

LINQ eröffnet zahlreiche Varianten für den Zugriff auf verschiedenste Arten von Daten. Diese sind in den verschiedenen LINQ-Implementierungen (auch als „LINQ Flavors“, d.h. „Geschmacksrichtungen“, bezeichnet) enthalten. Folgende LINQ-Provider wurden bereits als Bestandteile des .NET-Frameworks bereitgestellt (siehe Bild 7.1):

- LINQ to Objects (arbeitet mit Collections, die *IEnumerable<T>* implementieren),
- LINQ to XML (Zugriff auf XML-Strukturen),
- LINQ to DataSet (arbeitet auf Basis von DataSets) und
- LINQ to Entities (verwendet das Entity Framework (Core) als ORM). Entities sind vom Typ *IQueryable<T>* anstatt von *IEnumerable<T>* wie bei LINQ to Objects.

Diese LINQ-Provider/Implementierungen bilden eine Familie von Tools, die einzeln für bestimmte Aufgaben eingesetzt oder aber auch für leistungsfähige Lösungen mit einem Mix aus Objekten, XML und relationalen Daten kombiniert werden können.



HINWEIS: Wir werden uns in diesem Kapitel hauptsächlich auf **LINQ to Objects** beschränken, da dieser Provider die grundlegende Technologie bereitstellt und die übrigen Flavors mehr eine Angelegenheit der Datenbankliteratur sind.

Nochmals sei hier betont, dass LINQ eine offene Technologie ist, der jederzeit neue Provider hinzugefügt werden können. Die im .NET Framework enthaltenen Implementierungen bilden lediglich eine Basis, die eine Menge von Grundbausteinen (Abfrageoperatoren, Abfrageausdrücke, Abfragebäume) bereitstellt.

Die Einführung von LINQ machte mehrere Ergänzungen bei den .NET-Programmiersprachen erforderlich, von denen einige bereits in diesem Kapitel (Lambda-Ausdrücke) bzw. im

einführenden Sprachkapitel 3 (Typinferenz) bzw. im OOP-Kapitel 4 (Objektinitialisierer) besprochen wurden. Auf zwei weitere Sprachfeatures (anonyme Typen und Erweiterungsmethoden), ohne die LINQ nicht möglich wäre, wollen wir im Folgenden eingehen.

7.1.2 Anonyme Typen

Darunter verstehen wir einfache namenlose Klassen, die vom Compiler automatisch erzeugt werden und die nur über Eigenschaften und dazugehörige private Felder verfügen. „Namenlos“ bedeutet, dass uns der Name der Klasse nicht bekannt ist und man deshalb keinen direkten Zugriff auf die Klasse hat. Lediglich eine Instanz steht zur Verfügung, die man ausschließlich lokal, d. h. im Bereich der Deklaration, verwenden kann.

Die Deklaration anonymer Typen erfolgt mittels Typinferenz (Schlüsselwort *var*, siehe Abschnitt 3.2.10) und eines anonymen Objektinitialisierers. Das heißt, man lässt beim Initialisieren einfach den Klassennamen weg und schreibt stattdessen *var*. Der Compiler erzeugt die anonyme Klasse anhand der Eigenschaften im Objektinitialisierer und anhand des jeweiligen Typs der zugewiesenen Werte.



HINWEIS: In den meisten Beispielen, die Sie im Internet finden, wird fast nur noch *var* anstatt des korrekten Typnamens geschrieben, also auch bei einfachen Wertetypen. Dafür wurde das Schlüsselwort eigentlich nicht eingeführt, aber die Bequemlichkeit hat in den letzten Jahren wohl überhandgenommen. Bei sämtlichen Beispielen bis hierher haben wir auf *var* verzichtet, weil zumindest der Autor dieses Kapitels kein großer Fan davon ist. Ab jetzt werden wir aber *var* dort verwenden, wo wir es ohnehin brauchen, und auch an den Stellen, wo eindeutig erkennbar ist, für welchen Datentyp *var* steht.

Beispiel 7.1: Eine Objektvariable *person* wird aus einer anonymen Klasse instanziiert.

C#

```
var person = new { Vorname = "Jürgen", Nachname = "Kotz", Alter = 56};
```

Der Compiler generiert hierfür intern (in MSIL-Code) die folgende Klasse:

```
internal class ??????
{
    public string Vorname { get; set; }
    public string Nachname { get; set; }
    public int Alter { get; set; }
}
```



HINWEIS: Da der Typ der Eigenschaften aus der jeweiligen Klasse bzw. Struktur des im Objektinitialisierer zugewiesenen Werts abgeleitet wird, darf man hier nicht *null* zuweisen, denn der Compiler kann in diesem Fall den Datentyp der Eigenschaft nicht bestimmen.

Beispiel 7.2: Fehlerhafter Code, der Compiler kann den Typ der Eigenschaft *Alter* nicht ableiten.

C#

```
var person = new { Vorname = "Jürgen" , Nachname = "Kotz", Alter = null};
// Fehler!!!
```

Sobald eine weitere anonyme Klasse deklariert wird, bei der im Objektinitialisierer Eigenschaften mit gleichem Namen, Typ und in der gleichen Reihenfolge wie bei einer anderen bereits vorhandenen anonymen Klasse angegeben sind, verwendet der Compiler die gleiche anonyme Klasse und es sind untereinander Zuweisungen möglich.

Beispiel 7.3: Da Name, Typ und Reihenfolge der Eigenschaften im Objektinitialisierer bei *person* (siehe oben) und *kunde* identisch sind, ist eine direkte Zuweisung möglich.

C#

```
var kunde = new { Vorname = "Lennard", Nachname = "Kotz", Alter = 18};
person = kunde;
```



HINWEIS: Anonyme Typen dürfen nur lokal verwendet werden. Sie sind auch als Übergabeparameter nicht erlaubt.

7.1.3 Erweiterungsmethoden

Normalerweise erlaubt eine objektorientierte Programmiersprache das Erweitern von Klassen durch Vererbung. Bereits C# 3.0 führte aber eine neue Syntax ein, die das direkte Hinzufügen neuer Methoden zu einer vorhandenen Klasse erlaubt. Diese sogenannten *Erweiterungsmethoden* werden als statische Methoden in einer neuen statischen Klasse implementiert und können dann wie eine normale Methode (d.h. Instanzmethode) des erweiterten Datentyps aufgerufen werden. Um eine Methode als Erweiterungsmethode zu deklarieren, wird vor dem ersten Parameter das Schlüsselwort *this* angegeben. Der Argumenttyp des ersten Parameters bezieht sich auf die zu erweiternde Klasse bzw. Struktur. Wird die Erweiterungsmethode dann aufgerufen, übergibt der Compiler die Instanz des erweiterten Typs als erstes Argument an die Methode.



HINWEIS: Erweiterungsmethoden können auch für Interfaces definiert werden.

Beispiel 7.4: Zwei Erweiterungsmethoden (*Multiply* und *Abs*) für die Basisklasse *System.Int32*

C#

```
public static class IntExtension
{
    // 1. Erweiterungsmethode
    public static int Multiply(this int instanz, int faktor)
    {
```

```
        return instanz * faktor;
    }
    // 2. Erweiterungsmethode
    public static int Abs(this int instanz)
    {
        if (instanz < 0)
        {
            return -1 * instanz;
        }
        return instanz;
    }
}
```

Der Test:

```
int zahl = -95;
textBox1.Text = zahl.Multiply(7).ToString();    // -665
textBox2.Text = zahl.Abs().ToString();          // 95
```

In diesem Beispiel kann man nun die Erweiterungsmethoden *Multiply* und *Abs* für jede Integer-Variable so nutzen, als wären diese Methoden direkt in der Basisklasse *System.Int32* als Instanzenmethoden implementiert.



HINWEIS: Falls in *System.Int32* bereits eine *Abs*-Methode mit der gleichen Signatur wie die gleichnamige Erweiterungsmethode existieren würde, so hätte die in *System.Int32* bereits vorhandene Methode Vorrang vor der Erweiterungsmethode.



HINWEIS: Befindet sich die statische Klasse, in der die Extensionmethoden definiert sind, in einem anderen Namespace als die der aufrufenden Klasse, so muss unbedingt der Namespace der statischen Klasse mittels *using* bekannt gemacht werden. IntelliSense schlägt Ihnen die Extensionmethoden erst dann vor, wenn auch der Namespace bekannt ist.

■ 7.2 Abfragen mit LINQ to Objects

LINQ stellt bekanntlich grundlegende Abfragefunktionen in einer SQL-ähnlichen Syntax bereit. Dazu gehören zunächst als wichtigster Standard das Angeben einer Quelle (*from*), das Festlegen der zurückzugebenden Daten (*select*), das Filtern (*where*) und das Sortieren (*orderby*). Hinzu kommen eine Fülle weiterer Operatoren, wie z. B. für das Gruppieren, Verknüpfen und Sammeln von Datensätzen, auf die wir aber erst später eingehen wollen.

7.2.1 Grundlegendes zur LINQ-Syntax

Die LINQ-Abfrageoperatoren sind als Erweiterungsmethoden (siehe Abschnitt 7.1.3) definiert und in der Regel auf beliebige Objekte, die *IEnumerable<T>* implementieren, anwendbar.

Beispiel 7.5: Gegeben sei die folgende Auflistung.

C#

```
string[] monate = { "Januar", "Februar", "März", "April", "Mai", "Juni",
    "Juli", "August", "September", "Oktober", "November", "Dezember" };
```

Die folgende LINQ-Abfrage selektiert die Monatsnamen mit einer Länge von sechs Buchstaben, wandelt sie in Großbuchstaben um und ordnet sie alphabetisch.

```
var expr = from s in monate
    where s.Length == 6
    orderby s
    select s.ToUpper();
```

Die Ergebnisanzeige:

```
foreach (string item in expr)
{
    listBox1.Items.Add(item);
}
```

Das Resultat in einem Listenfeld:

```
AUGUST
JANUAR
```

Obiges Beispiel demonstriert das allgemeine Format einer LINQ-Abfrage:

Syntax:

```
from ... < where ... orderby ... > select ...
```

Eine LINQ-Abfrage muss immer mit *from* beginnen. Im Wesentlichen durchläuft *from* eine Liste von Daten. Dazu wird eine Variable benötigt, die jedem einzelnen Datenelement in der Quelle entspricht.



HINWEIS: Wer die Sprache SQL kennt, der wird zunächst davon irritiert sein, dass eine LINQ-Abfrage mit *from* und nicht mit *select* beginnt. Der Grund hierfür ist, dass nur so ein effektives Arbeiten mit der IntelliSense von Visual Studio möglich ist. Da zuerst die Datenquelle ausgewählt wird, kann die IntelliSense geeignete Typmitglieder für die Objekte der Auflistung anbieten. Vielleicht haben Sie diese Unterstützung im SQL Server Management Studio schon mal vermisst.

Weiterhin erkennen Sie, wie vom neuen Sprachfeature der lokalen Typinferenz (implizite Variablendeklaration) Gebrauch gemacht wird, denn die Anweisung

```
var expr = from s in monate ...
```

ist für den Compiler identisch mit

```
IEnumerable<string> expr = from s in monate ...
```

7.2.2 Zwei alternative Schreibweisen von LINQ-Abfragen

Grundsätzlich sind für LINQ-Abfragen zwei gleichberechtigte Schreibweisen möglich:

- Query-Expression-Syntax¹ (Abfrage-Syntax)
- Extension-Method-Syntax (Erweiterungsmethoden-Syntax)

Bis jetzt haben wir aber nur die Query-Expression-Syntax verwendet. Um die volle Leistungsfähigkeit von LINQ auszuschöpfen, sollten Sie aber beide Syntaxformen verstehen.

Beispiel 7.6: Die LINQ-Abfrage des obigen Beispiels in Extension-Method-Syntax

C#

```
var expr = monate
    .Where(s => s.Length == 6)
    .OrderBy(s => s)
    .Select(s => s.ToUpper());
```

Oder kompakt in einer Zeile:

```
var expr = monate.Where(s => s.Length == 6).OrderBy(s => s).
    Select(s => s.ToUpper());
```

Wie Sie sehen, verwenden wir bei dieser Notation Erweiterungsmethoden und Lambda-Ausdrücke. Aber auch eine Kombination von *Query-Expression-Syntax* mit *Extension-Method-Syntax* ist möglich.

Beispiel 7.7: Obiges Beispiel in gemischter Syntax

C#

```
var expr = (from s in monate where s.Length == 6 select s.ToUpper()).
    OrderBy(s => s);
```

Hier wurde ein Abfrageausdruck in runde Klammern eingeschlossen, gefolgt von der Erweiterungsmethode *OrderBy*. So lange, wie der Abfrageausdruck ein *IEnumerable* zurückgibt, kann darauf eine ganze Kette von Erweiterungsmethoden folgen.

Die Query-Expression-Syntax (Abfragesyntax) ermöglicht das Schreiben von Abfragen in einer SQL-ähnlichen Weise. Der Compiler kompiliert alle Queries in die Extension-Method-Syntax, die der objektorientierte Programmierung näher steht. Dabei wird z. B. die Filterbedingung *where* einfach in den Aufruf einer Erweiterungsmethode namens *Where* der *Enumerable*-Klasse übersetzt, die im Namespace *System.Linq* definiert ist.

¹ Die *Extension-Method-Syntax* wird auch als *Dot-Notation-Syntax* bezeichnet und entspricht der Objektorientierung etwas mehr.

Allerdings unterstützt die Query-Expression-Syntax nicht jeden standardmäßigen Abfrageoperator bzw. kann nicht jeden unterstützen, den Sie selbst hinzufügen. In einem solchen Fall sollten Sie direkt die Extension-Method-Syntax verwenden.

Abfrageausdrücke unterstützen eine Anzahl verschiedener „Klauseln“, z.B. *where*, *select*, *orderby*, *groupby* und *join*. Wie bereits erwähnt, lassen sich diese Klauseln in die gleichwertigen Operatoraufrufe übersetzen, die wiederum über Erweiterungsmethoden implementiert werden. Die enge Beziehung zwischen den Abfrageklauseln und den Erweiterungsmethoden, welche die Operatoren implementieren, erleichtert ihre Kombination, falls die Abfragesyntax keine direkte Klausel für einen erforderlichen Operator unterstützt.

7.2.3 Übersicht der wichtigsten Abfrageoperatoren

Die Klasse *Enumerable* im Namespace *System.Linq* stellt zahlreiche Abfrageoperatoren für LINQ to Objects bereit und definiert diese als Erweiterungsmethoden für Typen, die *IEnumerable<T>* implementieren.



HINWEIS: Kommen bei der Extension-Method-Syntax (Erweiterungsmethoden-Syntax) Abfrageoperatoren bzw. -methoden zur Anwendung, so sollten wir bei der Query-Expression-Syntax (Abfragesyntax) präziser von Abfrageklauseln bzw. -Statements sprechen.

Die folgende Tabelle zeigt die wichtigsten standardmäßigen Abfrageoperatoren von LINQ.

Bezeichnung der Gruppe	Operator
Beschränkungsoperatoren (Restriction)	<i>Where</i>
Projektionsoperatoren (Projection)	<i>Select</i> , <i>SelectMany</i>
Sortieroperatoren (Ordering)	<i>OrderBy</i> , <i>ThenBy</i>
Gruppierungsoperatoren (Grouping)	<i>GroupBy</i>
Quantifizierungsoperatoren (Quantifiers)	<i>Any</i> , <i>All</i> , <i>Contains</i>
Aufteilungsoperatoren (Partitioning)	<i>Take</i> , <i>Skip</i> , <i>TakeWhile</i> , <i>SkipWhile</i> , <i>Chunk</i>
Mengenoperatoren (Sets)	<i>Distinct</i> , <i>Union</i> , <i>Intersect</i> , <i>Except</i>
Elementoperatoren (Elements)	<i>First</i> , <i>FirstOrDefault</i> , <i>ElementAt</i>
Aggregatoperatoren (Aggregation)	<i>Count</i> , <i>Sum</i> , <i>Min</i> , <i>Max</i> , <i>Average</i>
Konvertierungsoperatoren (Conversion)	<i>ToArray</i> , <i>ToList</i> , <i>ToDictionary</i>
Typumwandlungsoperatoren (Casting)	<i>OfType<T></i>

Bild 7.2 illustriert an einem Beispiel, wie einige der bereits im Vorgängerabschnitt diskutierten neuen Sprachfeatures in LINQ-Konstrukten zur Anwendung kommen und wie die Abfragesyntax vom Compiler in die äquivalente Erweiterungsmethoden-Syntax umgesetzt wird.

Vergleich zwischen Abfragesyntax (oben) und Erweiterungsmethoden-Syntax (unten):

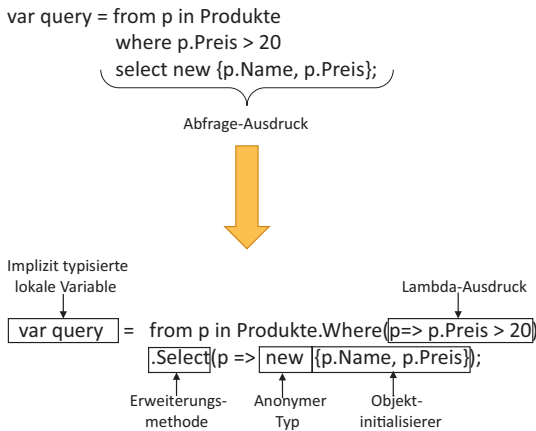


Bild 7.2
LINQ-Syntax

7.3 LINQ-Abfragen im Detail

Das Ziel der folgenden Beispiele ist nicht die vollständige Erläuterung aller in der obigen Tabelle aufgeführten Operatoren und deren Überladungen, sondern vielmehr eine Demonstration des prinzipiellen Aufbaus von Anweisungen zur Abfrage von Objektaufstellungen.

In der Regel werden beide Syntaxformen (Query-Expression-Syntax und Extension-Method-Syntax) gegenübergestellt, denn nur so erschließt sich am ehesten das allgemeine Verständnis für die auch für den SQL-Kundigen nicht immer leicht durchschaubare Logik der LINQ-Operatoren bzw. -Abfragen.

Für die Beispiele zu LINQ to Objects wird überwiegend auf eine Datenmenge zugegriffen, deren Struktur das folgende Diagramm zeigt.



HINWEIS: Die verwendeten Daten haben ihren Ursprung nicht in einer Datenbank, sondern werden per Code erzeugt (Listing siehe Beispieldaten zum Buch).

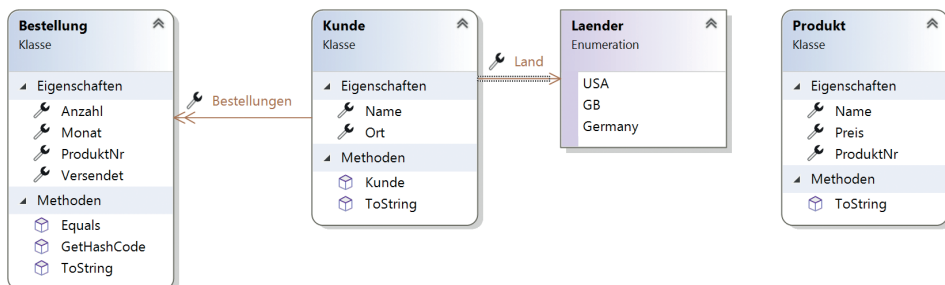


Bild 7.3 Datenmodell für unsere Beispiele

7.3.1 Die Projektionsoperatoren `Select` und `SelectMany`

Diese Operatoren „projizieren“ die Inhalte einer Quell-Auflistung in eine Ziel-Auflistung, die das Abfrageergebnis repräsentiert.

Select

Der Operator macht die Abfrageergebnisse über ein Objekt verfügbar, welches *IEnumerable<T>* implementiert.

Beispiel 7.8: Namen aller Produkte ausgegeben (Extension-Method-Syntax)

C#

```
var allProdukte = Produkte.Select(p => p.Name);
```

Alternativ die Query-Expression-Syntax:

```
var allProdukte = from p in Produkte select p.Name;
```

Die Ausgabe der Ergebnisliste:

```
listBox1.DataSource = allProdukte.ToList();
```

Ergebnis

Der Inhalt des Listenfelds sollte dann etwa folgenden Anblick bieten:

```
Marmelade
Quark
Mohrrüben
...
```

Beispiel 7.9: Das Abfrageergebnis wird auf einen anonymen Typ projiziert, der als Tupel definiert ist.

C#

```
var expr = Kunden.Select(k => new {k.Name, k.Ort});
```

Alternativ die Query-Expression-Syntax:

```
var expr = from k in Kunden
           select new { K.Name, k.Ort };
listBox1.DataSource = expr.ToList();
```

Ergebnis

```
{Name = Walter, Ort = Altenburg}
{Name = Thomas, Ort = Berlin}
```

SelectMany

Stände nur der *Select*-Operator zur Verfügung, so hätte man zum Beispiel bei der Abfrage der Bestellungen für alle Kunden eines bestimmten Landes das Problem, dass das Ergebnis

vom Typ *IEnumerable<Bestellung>* wäre, wobei es sich bei jedem Element um ein Array mit den Bestellungen eines einzelnen Kunden handeln würde. Um einen praktikableren, d. h. weniger tief geschachtelten, Ergebnistyp zu erhalten, wurde der Operator *SelectMany* eingeführt.

Beispiel 7.10: Die Bestellungen aller Kunden aus Deutschland sollen ermittelt werden.

C#

```
var bestellungen = Kunden
    .Where(k => k.Land == Länder.Germany)
    .SelectMany(k => k.Bestellungen);
```

Alternativ der Abfrageausdruck in Query-Expression-Syntax:

```
var bestellungen =
    from k in Kunden
    where k.Land == Länder.Germany
    from b in k.Bestellungen
    select b;
```

Das Auslesen des Ergebnisses der Abfrage:

```
listBox1.DataSource = bestellungen.ToList();
```

Die Ausgabe (Voraussetzung ist eine entsprechende Überschreibung der *ToString()*-Methode der Klasse *Bestellung*):

```
ProdNr: 2 , Anzahl: 4 , Monat: März, Versand: False
ProdNr: 1 , Anzahl: 11, Monat: Juni , Versand: True
...
```

7.3.2 Der Restriktionsoperator Where

Dieser Operator schränkt die Ergebnismenge anhand einer Bedingung ein. Sein prinzipieller Einsatz wurde bereits in den Vorgängerbeispielen hinreichend demonstriert. Außerdem können auch Indexparameter verwendet werden, um die Filterung auf bestimmte Indexpositionen zu begrenzen.

Beispiel 7.11: Die Kunden an den Positionen 2 und 3 der Kundenliste sollen angezeigt werden.

C#

```
var expr = Kunden
    .Where((k, index) => (index >= 2) && (index < 4))
    .Select(k => k.Name);
listBox1.DataSource = expr.ToList();
```

Ergebnis

```
Holger
Fernando
```

7.3.3 Die Sortieroperatoren `OrderBy` und `ThenBy`

Diese Operatoren bewirken ein Sortieren der Elemente innerhalb der Ergebnismenge.

`OrderBy/OrderByDescending`

Dieses nette Pärchen ermöglicht das Sortieren in auf- bzw. absteigender Reihenfolge.

Beispiel 7.12: Alle Produkte mit einem Preis kleiner gleich 20 sollen ermittelt und nach dem Preis sortiert ausgegeben werden (das teuerste zuerst).

C#

```
var prod = Produkte
    .Where(p => p.Preis <= 20)
    .OrderByDescending(p => p.Preis)
    .Select(p => new { p.Name, p.Preis });
```

Oder alternativ als Abfrageausdruck:

```
var prod = from p in Produkte
           where p.Preis <= 20
           orderby p.Preis descending
           select new { p.Name, p.Preis };
```

Die Ausgabe:

```
listBox1.DataSource = prod.ToList();
```

Ergebnis

```
{Name = Käse, Preis = 20}
{Name = Mohrrüben, Preis = 15}
...
```

`ThenBy/ThenByDescending`

Diese Operatoren verwendet man, wenn nacheinander nach mehreren Schlüsseln sortiert werden soll. Da `ThenBy` und `ThenByDescending` nicht auf den Typ `IEnumerable<T>`, sondern nur auf den Typ `IOrderedSequence<T>` anwendbar sind, können diese Operatoren nur im Anschluss an `OrderBy/OrderByDescending` eingesetzt werden.

Beispiel 7.13: Alle Kunden sollen zunächst nach ihrem Land und dann nach ihren Namen sortiert werden.

C#

```
var knd = Kunden
    .OrderBy(k => k.Land)
    .ThenBy(k => k.Name)
    .Select(k => new { k.Land, k.Name});
```

Der alternative Abfrageausdruck:

```
var knd = from k in Kunden
           orderby k.Land, k.Name
           select new { k.Land, k.Name };
```

Die Ausgabe ...

```
listBox1.DataSource = knd.ToList();
```

Ergebnis

... führt in beiden Fällen zu einem Ergebnis wie diesem:

```
{Land = USA, Name = Fernando}
{Land = USA, Name = Holger}
{Land = GB, Name = Alice}
{Land = Germany, Name = Thomas}
{Land = Germany, Name = Walter}
```

Reverse

Dieser Operator bietet eine einfache Möglichkeit, um die Reihenfolge der Elemente im Abfrageergebnis umzukehren. Der Operator ist als Abfrageausdruck nicht verfügbar.

Beispiel 7.14: Das Vorgängerbeispiel mit umgekehrter Reihenfolge der Ergebniselemente

C#

```
var knd = Kunden.OrderBy(k => k.Land).ThenBy(k => k.Name).
    Select(k => new { k.Land, k.Name }).Reverse();
```

Ergebnis

```
{Land = Germany, Name = Walter}
{Land = Germany, Name = Thomas}
...
```

7.3.4 Der Gruppierungsoperator GroupBy

Dieser Operator kommt dann zum Einsatz, wenn das Abfrageergebnis in gruppierter Form zur Verfügung stehen soll. *GroupBy* wählt die gewünschten Schlüssel-Elemente-Zuordnungen aus der abzufragenden Auflistung aus.

Beispiel 7.15: Alle Kunden nach Ländern gruppieren

C#

```
var knd = Kunden
    .GroupBy(k => k.Land);
```

Der alternative Abfrageausdruck:

```
var knd = from k in Kunden
    group k by k.Land;
```

Durchlaufen der Ergebnismenge:

```
foreach (IGrouping<Laender, Kunde> kdGroup in knd)
{
    listBox1.Items.Add(kdGroup.Key);
}
```

```

        foreach (var kd in kdGroup)
        {
            listBox1.Items.Add($" {kd}");
        }
    }
}

```

Ergebnis

Der Gruppenschlüssel (*kdGroup.Key*) ist hier das Land. Die Standardausgabe der Gruppenelemente erfolgt entsprechend der überschriebenen *ToString()*-Methode der Klasse *Kunden* (siehe Beispieldaten zum Buch):

```

Germany
    Walter – Altenburg – Germany
    Thomas – Berlin – Germany
USA
    Holger – Washington – USA
    Fernando – New York – USA
GB
    Alice – London – GB

```

Der *GroupBy*-Operator existiert in mehreren Überladungen, die alle den Typ *IEnumerable<IGrouping<K, T>>* liefern. Die generische Schnittstelle *IGrouping<K, T>* definiert einen spezifischen Schlüssel vom Typ *K* für die Gruppenelemente (Typ *T*).

Der Typ der äußeren Schleifenvariablen *IGrouping<Laender, Kunde>* kann auch implizit deklariert werden, sodass sich der Schleifenkopf im obigen Beispiel wie folgt vereinfachen lässt:

```

foreach(var kdGroup in knd)

```

Wenn nicht der standardmäßige, sondern ein benutzerdefinierter Elemente-Selektor zum Einsatz kommen soll, muss eine weitere Überladung von *GroupBy* verwendet werden (siehe folgendes Beispiel).

Beispiel 7.16: Das gleiche Problem wie im Vorgängerbeispiel wird gelöst, als Gruppenelemente werden allerdings nur die Namen der Kunden ausgegeben.

C#

```

var knd = Kunden
    .GroupBy(k => k.Land, k => k.Name);
foreach (var kdGroup in knd)
{
    listBox1.Items.Add(kdGroup.Key);
    foreach (var kd in kdGroup)
    {
        listBox1.Items.Add($" {kd}");
    }
}

```

Ergebnis

```

Germany
    Walter
    Thomas

```

```
USA
  Holger
  Fernando
GB
  Alice
```

Beispiel 7.17: Alle Produkte werden nach ihren Anfangsbuchstaben gruppiert.

C#

```
var prodGroups = Produkte
    .GroupBy(p => p.Name[0], p => p.Name);
foreach (var pGroup in prodGroups)
// var ersetzt IGrouping<char, string>
{
    listBox1.Items.Add(pGroup.Key);
    foreach (var p in pGroup)
    {
        listBox1.Items.Add($" {p}");
    }
}
```

Ergebnis

```
M
  Marmelade
  Mohrrüben
  Mehl
Q
  Quark
K
  Käse
H
  Honig
```

C#

Zum gleichen Ergebnis führt der folgende Code unter Verwendung eines Abfrageausdrucks:

```
var prodGroups = from p in Produkte
    group p by p.Name[0] into g
    select new { firstLetter = g.Key, prods = g };
foreach (var g in prodGroups)
{
    listBox1.Items.Add(g.firstLetter);
    foreach (var p in g.prods)
    {
        listBox1.Items.Add(p.Name);
    }
}
```

7.3.5 Verknüpfen mit Join

Mit diesem Operator definieren Sie Beziehungen zwischen verschiedenen Auflistungen. Im folgenden Beispiel werden Bestelldaten auf Produkte projiziert.

Beispiel 7.18: Die Bestellungen aller Kunden werden aufgelistet.

C#

```
var bestprod = Kunden.SelectMany(k => k.Bestellungen)
    .Join(Produkte, b => b.ProduktNr, p => p.ProduktNr,
        (b, p) => new {
            b.Monat, p.ProduktNr, p.Name, p.Preis, b.versendet });
```

Alternativ die Notation in Abfragesyntax:

```
var bestprod = from k in Kunden
                from b in k.Bestellungen
                join p in Produkte on b.ProduktNr equals p.ProduktNr
                select new { b.Monat, p.ProduktNr, p.Name, p.Preis, b.versendet };
```

Beim Vergleich (*equals*) ist zu beachten, dass zuerst der Schlüssel der äußeren Auflistung (*b.ProduktNr*) und dann der der inneren Auflistung (*p.ProduktNr*) angegeben werden muss. Ein Vergleich mit dem Vergleichsoperator `==` ist ebenfalls nicht zulässig.

Die Anzeigeroutine:

```
listBox1.DataSource = bestprod.ToList();
```

Ergebnis

Das Ergebnis liefert die Übersicht über alle Bestellungen:

```
{Monat = März, ProduktNr = 2, Name = Quark, Preis = 10, versendet = False}
{Monat = Juni, ProduktNr = 1, Name = Marmelade, Preis = 5,
    versendet = False}
{Monat = November, ProduktNr = 3, Name = Mohrrüben, Preis = 15,
    versendet = True}
{Monat = November, ProduktNr = 5, Name = Honig, Preis = 25,
    versendet = True}
{Monat = Juni, ProduktNr = 6, Name = Mehl, Preis = 30, versendet = False}
{Monat = Februar, ProduktNr = 4, Name = Käse, Preis = 20, versendet = True}
...
```

7.3.6 Aggregat-Operatoren

Zum Abschluss unserer Stippvisite bei den LINQ-Operatoren wollen wir noch einen kurzen Blick auf eine weitere wichtige Familie werfen. Diese Operatoren, zu denen *Count*, *Sum*, *Max*, *Min*, *Average* etc. gehören, setzen Sie ein, wenn Sie verschiedenste Berechnungen mit den Elementen der Datenquelle durchführen wollen.

Count

Die von diesem Operator durchzuführende Aufgabe ist sehr einfach, es wird die Anzahl der Elemente in der abzufragenden Auflistung ermittelt.

Beispiel 7.19: Alle Kunden sollen, zusammen mit der Anzahl der von ihnen aufgegebenen Bestellungen, angezeigt werden.

C#

```
var kdn = Kunden.Select(k => new {  
    k.Name, k.Ort, AnzahlBest = k.Bestellungen.Count() });
```

Oder das Gleiche in Abfragesyntax:

```
var kdn = from k in Kunden  
    select new { k.Name, k.Ort, AnzahlBest = k.Bestellungen.Count()};
```

Wir zeigen die Ergebnismenge in der ListBox an:

```
listBox1.DataSource = kdn.ToList();
```

Ergebnis

```
{Name = Walter, Ort = Altenburg, AnzahlBest = 1}  
{Name = Thomas, Ort = Berlin, AnzahlBest = 2}  
...
```

Wie Sie sehen, scheint die Anwendung dieser Operatoren einfach und leicht verständlich zu sein.

Sum

Wie es der Name schon vermuten lässt, können mit diesem Operator verschiedenste Summen aus den Elementen der Quell-Auflistung gebildet werden. Zunächst ein einfaches Beispiel.

Beispiel 7.20: Die Summe aller Preise der Produktliste

C#

```
var total = Produkte.Sum(p => p.Preis);
```

Die alternative Abfragesyntax (eigentlich gemischte Syntax):

```
var total = (from p in Produkte select p.Preis).Sum();
```

Die Ausgabe:

```
listBox1.Items.Add(total);  
// liefert mit den ursprünglichen Beispieldaten den Wert 105.
```

Das folgende Beispiel ist nicht mehr ganz so trivial, da sich hier der *Sum*-Operator innerhalb einer verschachtelten Abfrage versteckt.

Beispiel 7.21: Die Gesamtsumme aller angegebenen Bestellungen wird ermittelt.

C#

```
var expr = from k in Kunden
           join b in
             from k in Kunden
             from b in k.Bestellungen join p in Produkte
             on b.ProduktNr equals p.ProduktNr
           select new { k.Name, BestellBetrag = b.Anzahl * p.Preis }
           on k.Name equals b.Name into KundenMitBest
           select new { k.Name,
                      TotalBetrag = KundenMitBest.Sum(b => b.BestellBetrag)};
listBox1.DataSource = expr.ToList();
```

Ergebnis

```
{Name = Walter, TotalBetrag = 40}
{Name = Thomas, TotalBetrag = 340}
...
```

Das schaut jetzt schon nicht mehr ganz so trivial aus. Etwas übersichtlicher ist die Erweiterungsmethodensyntax:

```
var expr = Kunden.Select(k => new { k.Name,
                                   TotalerBetrag = k.Bestellungen.Sum(b => b.Anzahl *
                                   Produkte.First(p => p.ProduktNr == b.ProduktNr).Preis) });
```

Dadurch, dass bestimmte Operationen in der Abfragesyntax nicht möglich sind und wir deswegen sowieso die Erweiterungsmethodensyntax bzw. einen Mix verwenden müssen, ist der Favorit der Autoren eindeutig die Erweiterungsmethodensyntax. Deswegen werden wir in den folgenden Kapiteln die Abfragesyntax nicht mehr darstellen. Dem objektorientierten Programmierer, und das wollen Sie doch werden, dürfte die Punkt-Syntax sowieso besser gefallen.

7.3.7 Verzögertes Ausführen von LINQ-Abfragen

Normalerweise werden LINQ-Ausdrücke nicht bereits bei ihrer Definition, sondern erst bei Verwendung der Ergebnismenge ausgeführt (*deferred Execution*). Damit hat man die Möglichkeit, nachträglich Elemente zu der abzufragenden Auflistung hinzuzufügen bzw. zu ändern, ohne dazu die Abfrage nochmals neu erstellen zu müssen.

Beispiel 7.22: Alle Produkte, die mit dem Buchstaben „M“ beginnen, sollen ermittelt werden.

C#

```
var prods = Produkte.Where(p => p.Name[0] == 'M').Select(p => p.Name);
```

Oder alternativ in Abfragesyntax:

```
var prods = from p in Produkte where p.Name[0] == 'M' select p.Name;
```

Die Ergebnismenge wird das erste Mal durchlaufen und angezeigt:

```
foreach (var prod in prods)
{
    listBox1.Items.Add(prod);
}
listBox1.Items.Add("-----");
```

Anschließend ändern wir ein Element in der der Abfrage zugrunde liegenden Quelle

```
Produkte[0].Name = "Milch";
```

... und durchlaufen die Ergebnismenge ein zweites Mal:

```
foreach (var prod in prods)
{
    listBox1.Items.Add(prod);
}
```

Ergebnis

Die Ausgabe im Listenfeld zeigt, dass in der zweiten Ergebnismenge das geänderte Element erscheint:

```
Marmelade
Mohrrüben
Mehl
-----
Milch
Mohrrüben
Mehl
```

Wir sehen, dass die definierte Abfrage immer dann ausgeführt wird, wenn wir (wie hier in der *foreach*-Schleife) auf das Abfrageergebnis (*prods*) zugreifen.

Abfragen dieser Art bezeichnet man deshalb auch als „verzögerte Abfragen“.² Mitunter aber ist dieses Verhalten nicht erwünscht, d. h. man möchte das Abfrageergebnis nicht verzögert, sondern sofort nach Definition der Abfrage zur Verfügung haben. Abhilfe schafft hier die im nächsten Abschnitt beschriebene Anwendung von Konvertierungsmethoden.



HINWEIS: Wenn Sie mittels des LINQ-Operators *First* oder *FirstOrDefault* nur den ersten Satz einer Ergebnismenge lesen wollen (respektive *Last* oder *LastOrDefault*), dann wird die Abfrage auch sofort ausgeführt.

7.3.8 Konvertierungsmethoden

Zu dieser Gruppe gehören *ToArray*, *ToList*, *ToDictionary*, *AsEnumerable*, *Cast* und *ToLookup*. Sowohl die Methoden *ToArray* als auch *ToList* forcieren ein sofortiges Durchführen der Abfrage.

² *deferred query execution*

Beispiel 7.23: Das Vorgängerbeispiel wird wiederholt, diesmal aber wird das Abfrageergebnis in einer generischen List zwischengespeichert.

C#

```
var mProds = (Produkte.Where(p => p.Name[0] == 'M').
             Select(p => p.Name)).ToList();
```

bzw.:

```
var mProds = (from p in Produkte where p.Name[0] == 'M' select p.Name).ToList();
```

Ergebnis

Die Änderung der Quellfolge bleibt ohne Konsequenz für das Abfrageergebnis:

```
Produkte[0].Name = "Milch";
```

...

Ausgabe:

```
Marmelade
```

```
Mohrrüben
```

```
Mehl
```

```
-----
```

```
Marmelade
```

```
Mohrrüben
```

```
Mehl
```

7.3.9 Abfragen mit PLINQ

PLINQ ist eine parallele Implementierung von LINQ to Objects und kombiniert die Einfachheit und Lesbarkeit der LINQ-Syntax mit der Leistungsfähigkeit der parallelen Programmierung. PLINQ besitzt das komplette Angebot an Standard-Abfrageoperatoren und hat zusätzliche Operatoren für parallele Operationen.

Als Reaktion auf die zunehmende Verfügbarkeit von Mehrprozessorplattformen bietet PLINQ eine einfache Möglichkeit, die Vorteile paralleler Hardware einschließlich herkömmlicher Mehrprozessorcomputer und der neueren Generation von Mehrkernprozessoren zu nutzen.



HINWEIS: In vielen Szenarien kann PLINQ signifikant die Geschwindigkeit von LINQ-to-Objects-Abfragen steigern, da es alle verfügbaren Prozessoren des Computers nutzt.

Wer bereits mit LINQ vertraut ist, dem wird der Umstieg auf PLINQ kaum Sorgen bereiten. Die Verwendung von PLINQ entspricht meistens exakt der von LINQ to Objects und LINQ to XML. Sie können beliebige der bereits bekannten Operatoren nutzen, wie zum Beispiel *Join*, *Select*, *Where* usw.

Damit können Sie auch unter PLINQ Ihre bereits vorhandenen LINQ-Abfragen auf gewohnte Weise weiterverwenden, wenn Sie dabei einen wesentlichen Unterschied beachten:



HINWEIS: Parallelisieren Sie die Abfrage durch Aufruf der Erweiterungsmethode *AsParallel*!

Die Erweiterungsmethode *AsParallel* gehört zur *System.Linq.ParallelQuery*-Klasse. *AsParallel* kann auf jeder Datenmenge ausgeführt werden, die *IEnumerable<T>* implementiert.

Der Aufruf von *AsParallel* veranlasst den C#-Compiler, die parallele Version der Standard-Abfrageoperatoren zu binden. Damit übernimmt PLINQ die weitere Verarbeitung der Abfrage.

Beispiel 7.24: Eine einfache LINQ-Abfrage über eine Liste von Integer-Zahlen

C#

```
List<int> zahlen = new List<int>() { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Oder auch:

```
IEnumerable<int> zahlen = new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var q = from x in zahlen
        where x > 3
        orderby x descending
        select x;
```

Erst beim Iterieren über die Liste wird die Abfrage ausgeführt:

```
foreach (var z in q)
{
    listBox1.Items.Add(z.ToString());
    // 9, 8, 7, 6, 5, 4
}
```

Um dieselbe Abfrage mittels PLINQ auszuführen, ist lediglich *AsParallel* auf den Daten aufzurufen:

Beispiel 7.25: Die Abfrage im obigen Beispiel mit PLINQ

C#

```
...
var q = from x in zahlen.AsParallel()
        where x > 3
        orderby x descending
        select x;
...
```

Die Abfragen in den obigen Beispielen wurden in Query-Expression-Syntax geschrieben. Alternativ kann man natürlich auch die Extension-Method-Syntax³ verwenden.

³ Der Compiler konvertiert die Query-Expression-Syntax in die Extension-Method-Syntax, sodass letztendlich bei beiden Syntaxformen Erweiterungsmethoden aufgerufen werden.

Beispiel 7.26: Beide obigen Abfragen in Erweiterungsmethoden-Syntax

C#

Einfache LINQ-Version:

```
...
var q = zahlen
    .Where(x => x > 3)
    .OrderByDescending(x => x)
    .Select(x => x);
...

```

PLINQ-Version:

```
...
var q = zahlen.AsParallel()
    .Where(x => x > 3)
    .OrderByDescending(x => x)
    .Select(x => x);
...

```

Nach dem Aufruf der *AsParallel*-Methode führt PLINQ transparent die Erweiterungsmethoden (*Where*, *OrderBy*, *Select*, ...) auf allen verfügbaren Prozessoren aus. Genauso wie LINQ realisiert auch PLINQ eine verzögerte Ausführung von Abfragen, d.h. erst beim Durchlaufen der *foreach*-Schleife, beim Direktaufruf von *GetEnumerator* oder beim Eintragen der Ergebnisse in eine Liste (*ToList*, *ToDictionary*, ...) wird die Datenmenge abgefragt. Dann kümmert sich PLINQ darum, dass bestimmte Teile der Abfrage auf verschiedenen Prozessoren laufen, was mit versteckten multiplen Threads umgesetzt wird. Sie als Programmierer brauchen das nicht zu verstehen, Sie merken lediglich an der höheren Performance, dass die Prozessoren besser ausgelastet werden.

Probleme mit der Sortierfolge

Wie sollte es anders sein, bei genauerem Hinsehen werden Sie feststellen, dass es doch nicht ganz so unkompliziert ist, LINQ-Abfragen zu parallelisieren. Ganz abgesehen davon, dass die Parallelisierung nicht immer den erhofften Geschwindigkeitszuwachs bringt, haben wir es noch mit einem schwierigen und vor allem nicht gleich erkennbaren Problem zu tun: der Sortierfolge. Diese bereitet im Zusammenhang mit der parallelen Verarbeitung teilweise recht große Probleme, da auch bei einer geordneten Ausgangsmenge nicht eindeutig ist, in welcher Reihenfolge die Elemente durch PLINQ verarbeitet werden. Je nach LINQ-Operator kann es zu recht merkwürdigen Ergebnissen kommen.⁴

Aus diesem Grund wurde die Erweiterungsmethode *AsOrdered* eingeführt. Verwenden Sie diese im Zusammenhang mit *AsParallel*, wird die Sortierfolge der Ausgangsmenge in jedem Fall beibehalten.

⁴ Dies ist auch von der Anzahl der Prozessoren und der Größe der Datenmenge abhängig.

Beispiel 6.27: Verwendung von *AsOrdered***C#**

```
List<int> zahlen = new List<int>()  
    { 7, 4, 2, 3, 1, 6, 11, 5, 10, 8, 9, 13, 12 };  
var q = (from x in zahlen.AsParallel().AsOrdered()  
    where x > 3  
    select x).Take(5);
```

Das Ergebnis wird in jedem Fall

7, 4, 6, 11, 5

sein. Lassen Sie *AsOrdered* weg, sind weder die obige Reihenfolge noch die Zahlen eindeutig bestimmbar. Unter Umständen kann auch

13, 7, 11, 6, 5

ausgegeben werden.

Wie sich Sortierfolgen auf bestimmte Operatoren auswirken, beschreibt im Detail diese Webseite: [http://msdn.microsoft.com/de-de/library/dd460677\(v=vs.110\).aspx](http://msdn.microsoft.com/de-de/library/dd460677(v=vs.110).aspx).



HINWEIS: Grundsätzlich gilt jedoch: Vermeiden Sie im Zusammenhang mit PLINQ die Anwendung von Sortieroperationen. Diese machen die Vorteile von PLINQ durch erhöhten Verwaltungsaufwand meist wieder zunichte. Für Aufgaben, bei denen die Sortierung egal ist, erweist sich PLINQ jedoch als eine performancegewinnende Technologie.

7.4 Praxisbeispiele

7.4.1 Die Syntax von LINQ-Abfragen verstehen

In diesem Praxisbeispiel lernen Sie den prinzipiellen Aufbau von LINQ-Abfragen kennen. Im Zusammenhang damit kommen Sprachfeatures wie Typinferenz, Lambda-Ausdrücke und Erweiterungsmethoden zum Einsatz.

Prinzipiell gibt es zwei verschiedene Syntaxformen für LINQ-Abfragen:

- *Query-Expression-Syntax:* Hier werden Standard-Query-Operatoren verwendet.
- *Extension-Method-Syntax:* Hier kommen Erweiterungsmethoden zum Einsatz.

Im Folgenden werden wir beide Syntaxformen demonstrieren, um den Inhalt eines Integer-Arrays zu verarbeiten. Außerdem wird eine Mischform vorgeführt.

Oberfläche

Auf dem Startformular *Form1* befinden sich eine *ListBox* und drei *Buttons* (siehe Bild 7.4).

Index

Symbole

<ErrorBoundary> 1029
<PageTitle> 994
[ApiController] 950, 962
[Authorize] 930
.BAML 451
@bind 1001
[BindProperty] 889
@code 994
.cshtml 875, 898
.G.CS 451
[HttpGet] 951
[HttpPost] 951
[IInvokable] 1013
@model 886, 899
@rendermode 1015
.NET CLI 873, 894, 929
.NET Core 3
.NET-Framework 7
.NET MAUI 17, 649, 985
– Blazor 985
.NET Standard 16
??-Operator 69
.razor 993
@ (Razor-Syntax) 876
[Route] 950
[StreamRendering] 1016

A

Abbruchbedingung 779
Abhängige Eigenschaften 549
Abort 682

Abs 274
abstract 180f.
Abstraktion 5
AcceptReturn 495
Access-Key 491
Accessor 146
Action 320f.
ActionResult 956
Activated 478
ActualHeight 491
ActualWidth 491
AddCors 979
AddDays 269
AddDefaultPolicy 979
AddHours 269
AddMinutes 269
AddMonths 269
AddRange 304
AddYears 269
Ahead of Time 6
AllowsTransparency 481
Anfangswerte 64
Angehängte Eigenschaften 551
AngleX 588
AngleY 588
Animationen 590
Anonyme Methoden 316, 340
Anonyme Typen 347
AOT 943
API 939
App 444
– Klasse 475
App.config 787
App.Current 444

ApplicationCommands 565
 Application Programming Interface *siehe* API
 appsettings.json 915, 931, 960
 App.xaml.cs 444
 Array 121, 245, 257, 379, 384
 ArrayList 303, 310
 as-Operator 73
 ASP.NET Core 867

- MVC *siehe* ASP.NET Core MVC
- Projektvorlagen 870
- Rate-Limiting 935
- Razor Pages *siehe* Razor Pages

 ASP.NET Core Data Protection 1009
 ASP.NET Core Identity 928
 ASP.NET Core MVC 885, 893

- Action-Methoden 895
- CRUD 911
- Konzept 893
- Layout-Vorlagen 906
- Projektaufbau 894
- Routing 895
- Sessions *siehe* Sessions
- Zustandsmanagement 906

 ASP.NET Core Web API *siehe* ASP.NET Web API
 ASP.NET Web API 939

- Blazor 1003
- Controller 949
- CORS 978
- Daten aktualisieren 965
- Daten einfügen 964
- Daten lesen 949, 963
- Fehlerzustand 956
- HTTP-Statuscodes 956
- minimale API 968
- Paginierung 973
- Routing 950
- Vorlagen 943
- XML 975

 Assemblierung 11, 29, 46
 async 712
 Asynchrone Programmierentwurfsmuster 705
 Atn 274
 Attached Properties 456, 551
 Attribute 12

Auflistung 300
 Aufzählungszeichen 529
 Ausgabefenster 773
 Ausnahmen 797
 Ausnahmenfilter 405
 Authentifizierung 928
 AutoGenerateColumns 637
 AutoProperties 149, 404
 AutoProperty 142, 150
 Autorisierung 928
 AutoToolTipPlacement 512
 AutoToolTipPrecision 512
 await 712

B

Background 469, 489
 BandIndex 522
 Barrier 762
 base 169
 Befehlsfenster 770
 Begin 592
 BeginAnimation 592
 BeginInit 509
 BeginInvoke 700
 BeginningEdit 643
 Benannte Styles 568
 Bereichsoperator 424
 BigInteger 328
 Binary Application Markup Language 451
 BinaryFormatter 227
 BinarySearch 257
 Binding 597
 BindingBase.StringFormat 636
 BindingSource 228
 Bindungsarten 598
 BitmapFrame 509
 BitmapImage 509
 BlackoutDates 544
 Blazor 17, 981

- <ErrorBoundary> 1029
- APIs 1003
- Datenbindung 1000
- Event-Handling 997
- Fehlerbehandlung 1028
- File-Uploads 1026

- Grid 1033
- Hosting-Modelle 982
- JavaScript-Interoperabilität 1011
- Komponenten 993
- Layout-Template 990
- .NET MAUI 985
- Online-Status 1017
- Projektvorlagen 982, 985
- Protected Browser Storage 1009
- Render-Modus 1015
- Routing 994
- Seitentitel 994
- Tastaturereignisse 997
- Zustandsmanagement 1008
- Blazor Server 983, 1006
- Blazor WebAssembly 985 f., 1003
- BlockingCollection 762
- bool 62
- Boolesche Operatoren 83
- Border 527
- BorderBrush 489
- BorderThickness 527
- Boxing 77
- break 93, 120
- Breakpoints 776
- BringIntoView 540
- Bubbling Events 559
- BulletDecorator 529
- Button 492
- byte 61

C

- Calendar 542
- Callback 708
- CallerFilePath 789
- Caller Information 788
- CallerLineNumber 789
- CallerMemberName 789
- CamelCase-Notation 132
- CancellationToken 753
- CancellationTokenSource 753
- CanExecute 567
- Canvas 453, 458
- CaretBrush 497
- CaretIndex 496

- case 87, 120
- C#-Compiler 23
- CenterOwner 480
- CenterScreen 480
- CenterX 587
- CenterY 587
- ChangeTracker 833
- char 62
- CheckAccess 702
- CheckBox 498
- class 28, 122, 131
- ClassLoader 10
- Clear 257
- Client-Server-Prinzip 868
- Clone 256, 265 f.
- CLR 8, 10
- CLR-Threadpool 733
- CLS 7
- Codefenster 39
- Code Manager 10
- Collapsed 490
- Collection 300, 386, 609
- CollectionView 620
- CollectionViewSource 612
- ColumnDefinitions 465
- ColumnSpan 469
- ComboBox 505
- COM-Komponenten 11
- Command 561 f.
- CommandTarget 564
- COM-Marshallen 10
- Common Language Runtime 4, 10
- Common Language Specification 7 f.
- Common Type System 7, 9
- Completed 593
- Complex 330
- ComponentCommands 565
- ConcurrentBag 762
- ConcurrentDictionary 762
- ConcurrentQueue 762
- ConcurrentStack 762
- const 68
- Constraint 312
- Content Negotiation 975
- ContentPage 662
- ContentPresenter 579

ContentView 662
 ContextMenu 520
 continue 90
 Controller 893, 895, 944, 949
 ControllerBase 950
 ControlTemplate 579
 Convert 75, 634
 ConvertBack 634
 Cookies 907
 CopyTo 256, 265f.
 CornerRadius 527
 CORS 978, 1003
 Cos 274
 CountdownEvent 762
 Cross-Origin Resource Sharing *siehe* CORS
 Cross-Site Request Forgery 920
 CRUD 911, 924
 C#-Source-Datei 26
 CSRF *siehe* Cross-Site Request Forgery
 CTS 7
 Current 301
 CurrentItem 612, 620
 CurrentPosition 620
 Cursor 489

D

DataGrid 541, 637
 DataGridCheckBoxColumn 639
 DataGridTextColumn 639
 DataGridView 228
 DataTemplate 614
 Datenbankkontext 960
 Datenstrukturen 762
 Daten-Trigger 577
 Datentypen 61, 118
 Datenzugriff 96
 DateOnly 273
 DatePicker 542
 DateTime 268
 Datumsformatierung 279
 Datumsfunktionen 267
 Day 268
 DayOfWeek 268
 DayOfYear 268
 DaysInMonth 270
 Deactivated 478
 Deadlocks 677
 Debug 782
 - Write 783
 - Writelf 783
 - WriteLineIf 783
 Debugger 769
 decimal 62
 DecodePixelWidth 509
 default 90
 default! 410
 DefaultView 611
 deferred execution 394
 Dekrement 80
 Delay 601
 delegate 123, 155
 Delegate 313, 340
 - instanziiieren 314
 DELETE 942, 966
 Dependency Injection 184
 DependencyObject 550
 Dependency Properties 549
 Designer 38
 Destruktor 161, 165
 Diagnostics 680
 Dictionary 311
 Dimensionsgrenzen 251
 DirectX 440
 Direkte Events 561
 DispatcherUnhandledException 478
 DisplayDate 543
 DisplayMemberPath 614
 DisplayMode 542
 Dispose 167
 Distinct 377
 do 91f.
 DockPanel 453, 461
 DockPanel.Dock 461
 Document Object Model *siehe* DOM
 DOM 997
 double 61
 Duplikate 377
 Dynamische Programmierung 323

E

EditingCommands 565
 EF Core Provider 806
 Eigenschaften 31, 141
 Eigenschaften-Fenster 38
 Eigenschaften-Trigger 574
 Eigenschaftsmethoden 237
 Einzelschrittmodus 780
 Ellipse 547
 else 119
 else if 86
 Emscripten 985
 Endeoperator 424
 EndInvoke 700
 EndsWith 260
 Enter 690
 Entity Framework Core 914, 929
 Entwicklermodus 657
 Entwicklungsumgebung 33
 enum 94, 121
 Enumerable 379, 384
 Enumerationen 121
 Ereignis 31, 123, 154f.
 – auslösen 157
 Ereignismethoden 565
 Ereignis-Trigger 576
 Erweiterungsmethoden 348, 367
 event 123, 155
 EventLog 787
 EventLogTraceListener 787
 Events 31
 Exception 798
 ExceptionManager 10
 Exit 478, 690
 Exp 274
 ExpandDirection 531
 Expander 531
 Exponentialfunktion 275
 eXtensible Application Markup Language 442
 Extension-Method-Syntax 351, 367

F

Fehlerbehandlung 790
 Fehlerklassen 792, 799

Fill 470
 Filter 622
 FindResource 603
 float 61
 Fluent-API 826
 FontFamily 489
 FontSize 489
 FontStyle 489
 FontWeight 489
 for 91, 120
 foreach 121, 178, 249, 311
 Foreground 490
 Form1.cs 38
 Format 279, 634
 Formulare 31
 FromCurrentSynchronizationContext 758
 Func 320f.
 Funktionen 122

G

Garbage Collector 165
 Generics 307f.
 get 146
 GET 942
 GetDefaultView 611
 GetEnumerator 301, 310
 GetLength 256, 266
 Getter-only Auto-Property 150
 global usings 297
 goto 90
 Grafikausgabe 440
 Grafikskalierung 511
 Grid 453, 465
 Grid.Column 467
 Grid.Row 467
 GridSplitter 469
 GroupBox 528
 GroupName 500

H

Haltepunkte 778
 Hardwarebeschleunigung 440
 Hashtable 304
 HasValue 69

- Header 531
 - Hidden 490
 - HorizontalAlignment 458, 468, 490
 - HorizontalContentAlignment 490
 - HorizontalOffset 535
 - HorizontalScrollBarVisibility 514
 - Hot Reload 17, 874, 991
 - Hour 268
 - HSTS *siehe* HTTP Strict Transport Security
 - HTML-Formulare 886
 - HTML Helper 890
 - HTTP 868
 - Accept-Header 975
 - Content Negotiation 975
 - Methoden 869, 942
 - Statuscode 870, 956
 - HttpClient 1004
 - HttpContext 887, 908
 - HttpDelete 904
 - HttpPost 904
 - HttpPut 904
 - HTTP Strict Transport Security 873
- I**
- IActionResult 897
 - IAsyncEnumerable<T> 719
 - IAsyncResult 707f., 710
 - ICollection 302
 - IComparable 222
 - IComparer 222
 - Icon 480
 - IEnumerable 301, 379
 - IEnumerator 302
 - if 86, 119
 - IIS Express 872, 875
 - IJSRuntime 1011
 - Image 508
 - immutable 187, 284
 - IndentLevel 784
 - IndentSize 784
 - Index 245
 - Indexer 237, 299, 307, 338
 - IndexOf 257, 260
 - Indexprüfung 248
 - Initialisierer 379
 - Initialisierung 134
 - Initialize 256, 266
 - InitializeComponent 444
 - Init-only Setter 426
 - Inkrement 80
 - INotifyCollectionChanged 609
 - INotifyPropertyChanged 606
 - InputGestureText 517
 - Insert 260
 - Instanz 128
 - Instanzieren 133
 - int 61
 - Int16 61
 - Int32 61
 - Int64 61
 - IntelliSense 138
 - internal 130
 - internal protected 130
 - Interrupt 681
 - InvalidOperationException 613
 - Invoke 699, 710
 - InvokeRequired 701
 - IsAlive 683
 - IsBackground 683
 - IsCheckable 519
 - IsChecked 499f., 519
 - IsCompleted 707, 740
 - IsCurrentAfterLast 611, 620
 - IsCurrentBeforeFirst 612, 620
 - IsDirectionReversed 512
 - IsEditable 505
 - IsExpanded 533, 540
 - IsIndeterminate 526
 - IsLeapYear 270
 - IsLocked 522
 - IsOpen 535
 - IsReadOnly 495
 - IsSelected 540
 - IsSelectionRangeEnabled 513
 - IsSnapToTickEnabled 513
 - IsSynchronizedWithCurrentItem 613
 - IsThreeState 499
 - Iterator 310, 741
 - IValueConverter 634

J

JavaScript Object Notation *siehe* JSON
 JIT-Compiler 5
 Join 682
 JSON 903, 911, 947, 963
 JSON Columns 835
 JsonResult 903

K

k6 937
 Kapselung 5, 128
 Kartenspiel 232
 Kartesische Koordinaten 214
 Kestrel 872, 875
 Kind-Elemente 467
 Klasse 128
 – statische 183
 Klassendefinition 122
 Klassenmethode 153
 Kommentare 60
 Komplexe Zahlen 214
 Konsolenanwendung 106
 Konstante Felder 148
 Konstanten 61, 68
 Konstruktor 161
 – überladen 237
 Kontravarianz 327
 Kovarianz 327
 Kurz-Operatoren 81
 Kurzschlussauswertung 84

L

Label 491
 Lambda-Ausdruck 317, 367, 373
 Lambda Expression 340
 Language Integrated Query 345
 LastChildFill 462
 Lasttest 937
 launchSettings.json 874, 897
 Layout 453
 LazyLoading 632
 Leerzeichen 474
 Length 256, 260, 265f.

Line 548
 LineBreak 474
 LINQ 345, 373, 375, 384, 386
 – Abfrageoperatoren 352
 – Aggregat-Operatoren 360
 – AsEnumerable 363
 – Count 361
 – GroupBy 357
 – Grundlagen 345
 – Gruppierungsoperator 357
 – Join 360
 – Konvertierungsmethoden 363
 – OrderBy 356
 – OrderByDescending 356
 – Projektionsoperatoren 354
 – Restriktionsoperator 355
 – Reverse 357
 – Select 354
 – SelectMany 354
 – Sortierungsoperatoren 356
 – Sum 361
 – ThenBy 356
 – ToArray 363
 – ToDictionary 363
 – ToList 363
 – ToLookup 363
 – Where 355
 LINQ-Abfrageoperatoren 350
 LINQ-Architektur 345
 LINQ-Provider 346
 LINQ-Syntax 350
 LINQ to Objects 345
 List 308, 310
 ListBox 502
 List-Klasse 310
 ListView 541, 617
 Live Shaping 623
 Live Share 415
 Local Storage 1008
 lock 687
 Log 274
 Log10 274
 Logarithmus 275
 Logische Operatoren 82
 Lokale Variablen 70
 Lokal-Fenster 771

long 61
 LongRunning 758
 Long Term Support 3
 LowestBreakIteration 740

M

MainWindow.xaml 445
 MainWindow.xaml.cs 445
 ManualResetEventSlim 762
 Margin 457
 Mass Assignment 919, 963
 Matrix 237
 MatrixTransform 586
 Matrizen 242
 Max 274
 MaxHeight 457, 490
 MaxLength 496
 MaxLines 496
 MaxWidth 457, 490
 Media 680
 MediaCommands 565
 Menu 516
 Menü
 – Grafiken 518
 – Tastenkürzel 517
 MenuItem 516
 Menüleiste 515
 Messwertliste 370
 Metadaten 12
 Methoden 31, 97, 122, 151
 – generische 309
 – statische 153
 – überladen 114, 237
 – überladene 152
 Methodenzeiger 313
 MethodImpl 695
 Methods 31
 Microsoft Intermediate Language Code 5
 Min 274
 MinHeight 457, 490
 MinLines 496
 Minute 268
 MinWidth 457, 490
 Modale Dialoge 493
 Model 893, 899

Model Binding 889, 954
 Monitor 690
 Mono Runtime 651
 Month 268
 MoveCurrentTo 621
 MoveCurrentToFirst 612, 621
 MoveCurrentToLast 611, 621
 MoveCurrentToNext 611, 621
 MoveCurrentToPosition 621
 MoveCurrentToPrevious 612, 621
 MoveNext 301
 MSIL-Code 5
 Multi-Platform App UI 649
 MultipleRange 543
 Multitasking 676
 Multithreading 14, 676
 Mutex 694
 MVC *siehe* ASP.NET Core MVC
 MVVM 440

N

Name-Attribut 447
 Namespace 10, 133
 NavigationCommands 565
 new 123, 161, 246
 Next 381
 non-destructive mutation 192
 Now 270
 null 68, 135
 Nullable Type 68
 null-coalescing operator 69
 Null-Coalescing-Zuweisungsoperator
 422
 NULL-Zusammenführungsoperator 69
 Nutzeradministration 932

O

object 62
 – Datentyp 67
 Objekt 128
 Objektbaum 227
 Objekte 123
 Objektinitialisierer 164, 347f.
 ObservableCollection 609

- ODER 84
 - OnAfterRenderAsync 1009, 1019
 - OneTime 598
 - OneWay 598
 - OneWayToSource 598
 - OnExplicitShutdown 478
 - OnGet 886, 889
 - OnInitializedAsync 1006, 1011
 - OnLastWindowClose 478
 - Online-Status 1017
 - OnMainWindowClose 478
 - OnPost 887, 889
 - OnStartup 477
 - OOP 232
 - Opacity 481, 490
 - OpenAPI 944, 972
 - OpenFileDialog 510
 - Open Source 3
 - Operatoren 78, 119
 - arithmetische 79
 - boolesche 83
 - logische 82
 - Operatorenüberladung 214
 - Optionale Parameter 326
 - orderby 385
 - Orientation 459, 512, 514
 - out 102
 - OverflowMode 523
 - Overposting *siehe* Mass Assignment
 - override 169
- P**
- Padding 457, 490
 - PadLeft 260
 - PadRight 260
 - Paginator 1033
 - Paket-Manager-Konsole 627
 - PAP 106
 - Parallel.For 737
 - Parallel.ForEach 741
 - Parallel.Invoke 734
 - Parallel LINQ 763
 - ParallelLoopResult 740
 - Parallel-Programmierung 729
 - Parameterübergabe 101f.
 - Parent 490
 - Parse 75, 270
 - Pascal-Notation 132
 - PasswordBox 495, 497
 - PATCH 942
 - Pattern Matching 408
 - Pause 592
 - Pi 274
 - Placement 535
 - PlacementRectangle 535
 - PlacementTarget 535
 - PLINQ 364, 763
 - Polarkoordinaten 214
 - Polling 706
 - Polymorphes Verhalten 176
 - Polymorphie 5, 129, 168, 178
 - Popup 534
 - Portieren 117
 - POST 942, 963
 - Postman 948
 - Potenz 275
 - Pow 274
 - PreferFairness 758
 - Priority 683
 - private 130
 - private protected 414
 - Procedure-Step 776
 - Process 722, 726
 - Process.Start 727
 - ProcessThread 722
 - Program.cs 872, 945, 960, 969, 986
 - Programm starten 725
 - ProgressBar 526
 - Progressive Web Application *siehe* PWA
 - Projektmappen-Explorer 37
 - Projekttyp 36
 - Properties 31, 194
 - Property-Accessoren 146
 - PropertyChanged 600
 - protected 130
 - Prozeduren 122
 - Prozedurschritt 781
 - Prozesse 722
 - public 130
 - Pulse 690f.
 - PulseAll 690f.

PUT 942, 965

PWA 986

Q

Query-Expression-Syntax 351, 367

Queue 308, 311

QueueUserWorkItem 684

QuickGrid 1033

R

Racing 678

RadioButton 500

Rahmenbreite 527

Random 234, 276 f., 381

Range 380, 424

Rank 256, 265 f.

Rate-Limiting *siehe* ASP.NET Core, Rate Limiting

Razor-Komponenten *siehe* Blazor, Komponenten

Razor Pages 871, 875

– anlegen 875

– CRUD 924

– foreach 878

– Formulare 886

– Layout-Vorlagen 881

– Modelle 884

Razor-Syntax 876, 995

ReadLine 29

Records 187, 425

Rectangle 547

ref 101

Referenzieren 133

Referenztyp 67, 259

Reflexion 12

ReleaseMutex 694

Remove 260

RenderBody 882

RenderSectionAsync 883

Repeat 381

RepeatButton 492

Replace 260

Representational State Transfer *siehe* REST

Reset 301

Resources 490

Ressourcen 551

REST 940

– HTTP-Methoden 942

– Prinzipien 940

– URIs 941

Resume 592

return 90, 122, 310

Roslyn 24

RotateTransform 586

Round 274

Routed Events 558

RowDefinitions 465

RowDetailsTemplate 641

RowDetailsVisibilityMode 641, 643

RowSpan 469

Rückrufmethode 706

Rücksprung 781

S

Same-Origin Policy 978

ScaleTransform 586

ScaleX 587

ScaleY 587

Schaltjahr 270

Schleifen 120

Schleifenabbruch 739

Schleifenanweisungen 91

Schlüsselwörter 59, 296

ScrollBar 514

ScrollViewer 514

SDK-Style Projekte 55

sealed 181

Second 268

Security Engine 10

Seek 592

select 385

SelectedDate 543

SelectedItem 540

SelectedItemChanged 540

SelectedItems 503

SelectionBrush 497

SelectionMode 502

Semaphore 696

SemaphoreSlim 762

- Separator 516
 - Serialisierung 13
 - Serializable 228
 - SessionEnding 478
 - Sessions 907
 - JSON 911
 - lesen 908
 - schreiben 908
 - Session Storage 1008, 1011
 - set 142, 146
 - Shared-Methoden 237
 - short 61
 - Show 481
 - ShowDialog 481
 - Shutdown 478
 - Sign 274
 - SignalR 984
 - Sin 274
 - Single-Page Application 869, 939
 - SingleRange 543
 - Single-Step 776
 - Skalieren mit ScaleTransform 587
 - SkewTransform 586
 - Sleep 682
 - Slider 512
 - Sort 222, 257
 - SortDescriptions 622
 - SortedList 311
 - SortedSet 332
 - Sortieren 384
 - Source 508
 - SPA *siehe* Single-Page Application
 - SpellCheck.IsEnabled 496
 - Sperrmechanismen 685
 - SpinLock 762
 - SpinWait 762
 - Split 260
 - Spread-Operator 435
 - Sqrt 274
 - Stack 308
 - StackPanel 453, 459
 - StartInfo 726
 - StartsWidth 260
 - Startup 477f.
 - StartupEventArgs 477
 - StartupUri 444, 475
 - StateHasChanged 1009, 1019
 - static 122, 183
 - StaticResource 554, 571
 - Statische Klassen 183
 - Statische Methode 153
 - Statischer Konstruktor 164
 - StatusBar 524
 - StatusBarItem 524
 - Steuerelemente 31
 - Stop 592
 - Storyboard 590
 - Stretch 511
 - StretchDirection 511
 - string 62
 - String 259
 - Stringaddition 284
 - struct 95, 121
 - Strukturen 121
 - Strukturvariable 97
 - Style 490, 570
 - Style anpassen 571, 573
 - Style ersetzen 571
 - Styles vererben 572
 - Subklassen 169f.
 - SubString 260
 - Swagger UI 946, 972
 - switch 87, 120
 - System 61
 - System.Collections.Concurrent 762
 - System.Nullable 69
 - System.Object 179
 - Systemressourcen 557
 - System.Threading 679, 732
 - System.Threading.Tasks 732
- ## T
- TabControl 533
 - TabIndex 490
 - TabPanel 453
 - Tag 490
 - Tag Helper 892
 - Tan 274
 - Target 491
 - Task
 - Canceled 757

- ContinueWith 749, 758
- Created 757
- Datenübergabe 745
- Faulted 757
- Fehlerbehandlung 756
- IsCanceled 757
- IsCompleted 757
- IsFaulted 757
- Klasse 742
- RanToCompletion 757
- Result 749
- return 752
- Rückgabewerte 748
- Running 757
- starten 743
- Status 757
- TaskCreationOptions 758
- Task-Ende 758
- Task-Id 757
- User Interface 758
- Verarbeitung abbrechen 751
- Wait 747
- WaitAll 748
- WaitingForActivation 757
- WaitingForChildrenToComplete 757
- WaitingToRun 757
- weitere Eigenschaften 757
- Task<> 716
- Task.Factory.StartNew 742
- Task Parallel Library 729
- TaskScheduler 758
- Tastaturereignisse 997
- Template 578
- Textausrichtung 475
- TextBlock 471
- TextBox 495
- Textformatierungen 472
- TextFormattingMode 484
- TextWriterTraceListener 786
- Thin Client 198
- Thread 679, 681
- ThreadInterruptedException 681
- ThreadPool 684
- Threads 722
- Thread Service 10
- Threadsicher 698
- Threadsichere Collections 762
- ThreadState 683
- Throw 793, 799
- ThrowIfCancellationRequested 753
- TickFrequency 513
- TickPlacement 513
- TimeOnly 273
- Timer-Threads 704
- TimeSpan-Klasse 284
- Title 480
- ToArray 377, 385
- ToCharArray 260
- Today 270
- ToggleButton 492
- ToLongDateStrng 269
- ToLongTimeString 269
- ToLower 260
- ToolBar 521
- ToolBarTray 521 f.
- Toolbox 38
- ToolTip 490
- ToShortDateStrng 269
- ToShortTimeString 269
- ToStrng 74, 278
- ToUpper 260
- Trace 782, 786
- TraceListener 786
- TrackBar 284
- Transformationen 585
- TransformGroup 588
- TranslateTransform 586
- Transparenz 481
- TreeView 537
- Trefferanzahl 780
- Trigger 573
- Trim 260
- try 120
- try-catch 791
- TryEnter 690, 693
- try-finally 795
- Tunneling Events 558
- Tuple 331
- TwoWay 598
- Typdiskriminator 848
- Typecasting 336
- Typen 848

Typinferenz 70, 347, 367
 Typ-Styles 570
 Typsuffixe 64

U

Überladene Methoden 152
 Überwachungsfenster 772
 Uhr anzeigen 271
 UI-Virtualisierung 639
 Unboxing 77
 UND 84
 Unicode 65
 Uniform 470
 UniformGrid 453, 464
 UniformToFill 471
 UnIndent 784
 UpdateSourceTrigger 600
 Uri 556
 URI 941
 UseCors 979
 using 28, 133, 296

V

Value 512
 var 70
 Variablen 61
 Variablentypen 61
 VB 117
 Verarbeitungsstatus 740
 Vererbung 129
 Verformen mit SkewTransform 587
 Vergleichsoperatoren 82
 Verschieben mit TranslateTransform 588
 VerticalAlignment 458, 468, 490
 VerticalContentAlignment 490
 VerticalOffset 535
 VerticalScrollBarVisibility 514
 Verweistypen 62
 Verzweigungen 119
 View 893, 898
 ViewBox 453, 470
 ViewData 882, 906, 909
 ViewResult 902
 VirtualizingStackPanel 639

Visibility 490
 Visual Studio
 – ASP.NET Core 870
 Visual Studio 22
 Visual Studio Enterprise 22
 Visual Studio Professional 22
 void 99

W

Wait 690 f.
 WaitOne 694, 696
 WASM *siehe* WebAssembly
 Web API *siehe* API
 WebAssembly 984
 WebSockets 984
 Werkzeugkasten 38
 Wertetypen 62
 where 312
 while 91 f., 120
 Wiederholmuster 383
 Wiederverwendbarkeit 129
 Windows Presentation Foundation 439
 WindowStartupLocation 480
 WindowStyle 480 f.
 Winkel 275
 with 193
 work stealing 733
 WPF 439, 702
 – Anwendung beenden 478
 – Applikationstypen 449
 – Eigenschaften 489
 – Ereignishandler 446
 – Ereignismodell 557
 – Height 456
 – Kommandozeilenparameter 477
 – Left 456
 – Maßangaben 456
 – Startobjekt festlegen 475
 – Style-System 567
 – Top 456
 – Width 456
 – Window-Klasse 479
 – Zielplattformen 449
 WPF-Programm 475
 WPF-Wertkonvertierer 634

Wrap 472
WrapPanel 453, 463
WrapWithOverflow 472
Writeln 782
WriteLine 29, 782
Wurzel 275

X

Xamarin 17
XAML 442
x:Class 445
XML 975
xml:space 474
XOR 84

Y

Year 268
yield 310, 339

Z

Zahlenformatierung 278
Zeilenumbrüche 474
Zeitfunktionen 267
Zeitmessung 285
Zufallszahlen 276 f., 381
Zustandsmanagement 1008
Zuweisungsoperatoren 81