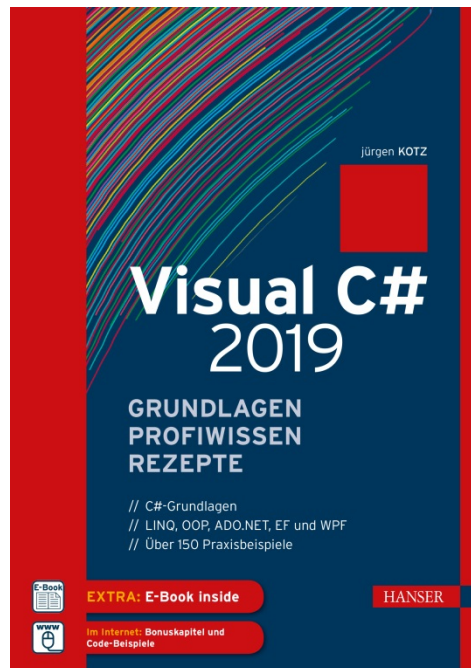


HANSER



Leseprobe

zu

„Visual C# 2019“

von Jürgen Kotz

Print-ISBN: 978-3-446-45802-4
E-Book-ISBN: 978-3-446-46099-7
E-Pub-ISBN: 978-3-446-46253-3

Weitere Informationen und Bestellungen unter
<http://www.hanser-fachbuch.de/978-3-446-45802-4>

sowie im Buchhandel

© Carl Hanser Verlag, München

Inhalt

Vorwort	XXIII
Teil I: Grundlagen	1
1 Einstieg in Visual Studio 2019	3
1.1 Die Installation von Visual Studio 2019	3
1.1.1 Überblick über die Produktpalette	3
1.1.2 Anforderungen an Hard- und Software	4
1.2 Unser allererstes C#-Programm	5
1.2.1 Vorbereitungen	5
1.2.2 Quellcode schreiben	7
1.2.3 Programm kompilieren und testen	8
1.2.4 Einige Erläuterungen zum Quellcode	9
1.2.5 Konsolenanwendungen sind out	10
1.3 Die Windows-Philosophie	10
1.3.1 Mensch-Rechner-Dialog	10
1.3.2 Objekt- und ereignisorientierte Programmierung	11
1.3.3 Programmieren mit Visual Studio 2019	12
1.4 Die Entwicklungsumgebung Visual Studio 2019	13
1.4.1 Neues Projekt	14
1.4.2 Die wichtigsten Fenster	16
1.5 Microsofts .NET-Technologie	20
1.5.1 Zur Geschichte von .NET	20
1.5.2 .NET-Features und Begriffe	22
1.6 Praxisbeispiele	29
1.6.1 Unsere erste Windows-Forms-Anwendung	30
1.6.2 Umrechnung Euro-Dollar	35
2 Grundlagen der Sprache C#	45
2.1 Grundbegriffe	45
2.1.1 Anweisungen	45
2.1.2 Bezeichner	46

2.1.3	Schlüsselwörter	47
2.1.4	Kommentare	48
2.2	Datentypen, Variablen und Konstanten	49
2.2.1	Fundamentale Typen	49
2.2.2	Werttypen versus Verweistypen	50
2.2.3	Benennung von Variablen	51
2.2.4	Deklaration von Variablen	51
2.2.5	Typsuffixe	52
2.2.6	Zeichen und Zeichenketten	53
2.2.7	object-Datentyp	55
2.2.8	Konstanten deklarieren	56
2.2.9	Nullable Types	56
2.2.10	Typinferenz	58
2.2.11	Gültigkeitsbereiche und Sichtbarkeit	59
2.3	Konvertieren von Datentypen	59
2.3.1	Implizite und explizite Konvertierung	59
2.3.2	Welcher Datentyp passt zu welchem?	62
2.3.3	Konvertieren von string	62
2.3.4	Die Convert-Klasse	64
2.3.5	Die Parse-Methode	65
2.3.6	Boxing und Unboxing	65
2.4	Operatoren	67
2.4.1	Arithmetische Operatoren	68
2.4.2	Zuweisungsoperatoren	70
2.4.3	Logische Operatoren	71
2.4.4	Rangfolge der Operatoren	73
2.5	Kontrollstrukturen	74
2.5.1	Verzweigungsbefehle	74
2.5.2	Schleifenanweisungen	79
2.6	Benutzerdefinierte Datentypen	82
2.6.1	Enumerationen	82
2.6.2	Strukturen	83
2.7	Nutzerdefinierte Methoden	86
2.7.1	Methoden mit Rückgabewert	86
2.7.2	Methoden ohne Rückgabewert	88
2.7.3	Parameterübergabe mit ref	89
2.7.4	Parameterübergabe mit out	90
2.7.5	Methodenüberladung	91
2.7.6	Optionale Parameter	92
2.7.7	Benannte Parameter	93
2.8	Praxisbeispiele	94
2.8.1	Vom PAP zur Konsolenanwendung	94
2.8.2	Ein Konsolen- in ein Windows-Programm verwandeln	97
2.8.3	Schleifenanweisungen verstehen	99

2.8.4	Benutzerdefinierte Methoden überladen	101
2.8.5	Anwendungen von Visual Basic nach C# portieren	104
3	OOP-Konzepte	113
3.1	Kleine Einführung in die OOP	113
3.1.1	Historische Entwicklung	114
3.1.2	Grundbegriffe der OOP	115
3.1.3	Sichtbarkeit von Klassen und ihren Mitgliedern	117
3.1.4	Allgemeiner Aufbau einer Klasse	118
3.1.5	Das Erzeugen eines Objekts	120
3.1.6	Einführungsbeispiel	123
3.2	Eigenschaften	128
3.2.1	Eigenschaften mit Zugriffsmethoden kapseln	128
3.2.2	Berechnete Eigenschaften	130
3.2.3	Lese-/Schreibschutz	132
3.2.4	Property-Accessoren	133
3.2.5	Statische Felder/Eigenschaften	134
3.2.6	Einfache Eigenschaften automatisch implementieren	136
3.3	Methoden	138
3.3.1	Öffentliche und private Methoden	138
3.3.2	Überladene Methoden	139
3.3.3	Statische Methoden	140
3.4	Ereignisse	142
3.4.1	Ereignis hinzufügen	142
3.4.2	Ereignis verwenden	145
3.5	Arbeiten mit Konstruktor und Destruktor	148
3.5.1	Konstruktor und Objektinitialisierer	149
3.5.2	Destruktor und Garbage Collector	152
3.5.3	Mit using den Lebenszyklus des Objekts kapseln	154
3.6	Vererbung und Polymorphie	155
3.6.1	Method-Overriding	155
3.6.2	Klassen implementieren	156
3.6.3	Implementieren der Objekte	159
3.6.4	Ausblenden von Mitgliedern durch Vererbung	160
3.6.5	Allgemeine Hinweise und Regeln zur Vererbung	162
3.6.6	Polymorphes Verhalten	163
3.6.7	Die Rolle von System.Object	166
3.7	Spezielle Klassen	167
3.7.1	Abstrakte Klassen	167
3.7.2	Versiegelte Klassen	168
3.7.3	Partielle Klassen	169
3.7.4	Statische Klassen	170
3.8	Schnittstellen (Interfaces)	171
3.8.1	Definition einer Schnittstelle	171

3.8.2	Implementieren einer Schnittstelle	172
3.8.3	Abfragen, ob Schnittstelle vorhanden ist	173
3.8.4	Mehrere Schnittstellen implementieren	173
3.8.5	Schnittstellenprogrammierung ist ein weites Feld	174
3.9	Praxisbeispiele	174
3.9.1	Eigenschaften sinnvoll kapseln	174
3.9.2	Eine statische Klasse anwenden	177
3.9.3	Vom fetten zum schlanken Client	179
3.9.4	Schnittstellenvererbung verstehen	190
3.9.5	Rechner für komplexe Zahlen	194
3.9.6	Sortieren mit Comparable/Comparer	203
3.9.7	Einen Objektbaum in generischen Listen abspeichern	208
3.9.8	OOP beim Kartenspiel erlernen	213
3.9.9	Eine Klasse zur Matrizenrechnung entwickeln	218
4	Arrays, Strings, Funktionen	225
4.1	Datenfelder (Arrays)	225
4.1.1	Array deklarieren	226
4.1.2	Array instanzieren	226
4.1.3	Array initialisieren	227
4.1.4	Zugriff auf Array-Elemente	228
4.1.5	Zugriff mittels Schleife	229
4.1.6	Mehrdimensionale Arrays	230
4.1.7	Zuweisen von Arrays	232
4.1.8	Arrays aus Strukturvariablen	233
4.1.9	Löschen und Umdimensionieren von Arrays	234
4.1.10	Eigenschaften und Methoden von Arrays	236
4.1.11	Übergabe von Arrays	237
4.2	Verarbeiten von Zeichenketten	239
4.2.1	Zuweisen von Strings	239
4.2.2	Eigenschaften und Methoden von String-Variablen	240
4.2.3	Wichtige Methoden der String-Klasse	242
4.2.4	Die StringBuilder-Klasse	244
4.3	Reguläre Ausdrücke	247
4.3.1	Wozu werden reguläre Ausdrücke verwendet?	247
4.3.2	Eine kleine Einführung	248
4.3.3	Wichtige Methoden/Eigenschaften der Klasse Regexp	248
4.3.4	Kompilierte reguläre Ausdrücke	250
4.3.5	RegexOptions-Enumeration	251
4.3.6	Metazeichen (Escape-Zeichen)	252
4.3.7	Zeichenmengen (Character Sets)	253
4.3.8	Quantifizierer	254
4.3.9	Zero-Width Assertions	256
4.3.10	Gruppen	259

4.3.11	Text ersetzen	260
4.3.12	Text splitten	261
4.4	Datums- und Zeitberechnungen	262
4.4.1	Die DateTime-Struktur	262
4.4.2	Wichtige Eigenschaften von DateTime-Variablen	263
4.4.3	Wichtige Methoden von DateTime-Variablen	264
4.4.4	Wichtige Mitglieder der DateTime-Struktur	265
4.4.5	Konvertieren von Datumstrings in DateTime-Werte	265
4.4.6	Die TimeSpan-Struktur	266
4.5	Mathematische Funktionen	268
4.5.1	Überblick	268
4.5.2	Zahlen runden	268
4.5.3	Winkel umrechnen	269
4.5.4	Potenz- und Wurzeloperationen	269
4.5.5	Logarithmus und Exponentialfunktionen	269
4.5.6	Zufallszahlen erzeugen	270
4.5.7	Kreisberechnung	271
4.6	Zahlen- und Datumsformatierungen	271
4.6.1	Anwenden der ToString-Methode	272
4.6.2	Anwenden der Format-Methode	273
4.6.3	Stringinterpolation	274
4.7	Praxisbeispiele	275
4.7.1	Zeichenketten verarbeiten	275
4.7.2	Zeichenketten mit StringBuilder addieren	278
4.7.3	Reguläre Ausdrücke testen	281
4.7.4	Methodenaufrufe mit Array-Parametern	283
5	Weitere Sprachfeatures	287
5.1	Namespaces (Namensräume)	287
5.1.1	Ein kleiner Überblick	287
5.1.2	Einen eigenen Namespace einrichten	288
5.1.3	Die using-Anweisung	289
5.1.4	Namespace Alias	290
5.2	Operatorenüberladung	291
5.2.1	Syntaxregeln	291
5.2.2	Praktische Anwendung	291
5.3	Collections (Auflistungen)	293
5.3.1	Die Schnittstelle IEnumerable	293
5.3.2	ArrayList	295
5.3.3	Hashtable	297
5.3.4	Indexer	297
5.4	Generics	300
5.4.1	Generics bieten Typsicherheit	300

5.4.2	Generische Methoden	301
5.4.3	Iteratoren	302
5.5	Generische Collections	303
5.5.1	List-Collection statt ArrayList	303
5.5.2	Vorteile generischer Collections	304
5.5.3	Constraints	304
5.6	Das Prinzip der Delegates	305
5.6.1	Delegates sind Methodenzeiger	305
5.6.2	Einen Delegate-Typ deklarieren	305
5.6.3	Ein Delegate-Objekt erzeugen	306
5.6.4	Delegates vereinfacht instanziiieren	308
5.6.5	Anonyme Methoden	308
5.6.6	Lambda-Ausdrücke	310
5.6.7	Lambda-Ausdrücke in der Task Parallel Library	312
5.7	Dynamische Programmierung	313
5.7.1	Wozu dynamische Programmierung?	314
5.7.2	Das Prinzip der dynamischen Programmierung	314
5.7.3	Optionale Parameter sind hilfreich	317
5.7.4	Kovarianz und Kontravarianz	317
5.8	Weitere Datentypen	318
5.8.1	BigInteger	318
5.8.2	Complex	321
5.8.3	Tuple<>	321
5.8.4	SortedSet<>	322
5.9	Praxisbeispiele	324
5.9.1	ArrayList versus generische List	324
5.9.2	Generische IEnumerable-Interfaces implementieren	327
5.9.3	Delegates, anonyme Methoden, Lambda Expressions	330
5.9.4	Dynamischer Zugriff auf COM Interop	334
6	Einführung in LINQ	339
6.1	LINQ-Grundlagen	339
6.1.1	Die LINQ-Architektur	339
6.1.2	Anonyme Typen	341
6.1.3	Erweiterungsmethoden	342
6.2	Abfragen mit LINQ to Objects	343
6.2.1	Grundlegendes zur LINQ-Syntax	344
6.2.2	Zwei alternative Schreibweisen von LINQ-Abfragen	345
6.2.3	Übersicht der wichtigsten Abfrageoperatoren	346
6.3	LINQ-Abfragen im Detail	347
6.3.1	Die Projektionsoperatoren Select und SelectMany	348
6.3.2	Der Restriktionsoperator Where	350
6.3.3	Die Sortierungsoperatoren OrderBy und ThenBy	350
6.3.4	Der Gruppierungsoperator GroupBy	352

6.3.5	Verknüpfen mit Join	354
6.3.6	Aggregat-Operatoren	355
6.3.7	Verzögertes Ausführen von LINQ-Abfragen	357
6.3.8	Konvertierungsmethoden	358
6.3.9	Abfragen mit PLINQ	359
6.4	Praxisbeispiele	362
6.4.1	Die Syntax von LINQ-Abfragen verstehen	362
6.4.2	Aggregat-Abfragen mit LINQ	365
6.4.3	LINQ im Schnelldurchgang erlernen	367
6.4.4	Strings mit LINQ abfragen und filtern	370
6.4.5	Duplikate aus einer Liste oder einem Array entfernen	371
6.4.6	Arrays mit LINQ initialisieren	374
6.4.7	Arrays per LINQ mit Zufallszahlen füllen	376
6.4.8	Einen String mit Wiederholmuster erzeugen	378
6.4.9	Mit LINQ Zahlen und Strings sortieren	379
6.4.10	Mit LINQ Collections von Objekten sortieren	380
6.4.11	Ergebnisse von LINQ-Abfragen in ein Array kopieren	383
7	Neuerungen von C# im Überblick	385
7.1	C# 4.0 – Visual Studio 2010	385
7.1.1	Datentyp dynamic	385
7.1.2	Benannte und optionale Parameter	386
7.1.3	Kovarianz und Kontravarianz	387
7.2	C# 5.0 – Visual Studio 2012	388
7.2.1	Async und Await	388
7.2.2	CallerInfo	389
7.3	Visual Studio 2013	390
7.4	C# 6.0 – Visual Studio 2015	390
7.4.1	String Interpolation	390
7.4.2	Schreibgeschützte AutoProperties	390
7.4.3	Initialisierer für AutoProperties	391
7.4.4	Expression Body Funktionsmember	391
7.4.5	using static	392
7.4.6	Bedingter Nulloperator	392
7.4.7	Ausnahmenfilter	393
7.4.8	nameof-Ausdrücke	393
7.4.9	await in catch und finally	394
7.4.10	Indexinitialisierer	394
7.5	C# 7.0 – Visual Studio 2017	394
7.5.1	out-Variablen	394
7.5.2	Tupel	395
7.5.3	Mustervergleich	396
7.5.4	Discards	398
7.5.5	Lokale ref-Variablen und Rückgabetypen	398

7.5.6	Lokale Funktionen	398
7.5.7	Mehr Expression-Bodied Member	399
7.5.8	throw-Ausdrücke	399
7.5.9	Verbesserung der numerischen literalen Syntax	399
7.6	C# 7.1 bis 7.3 – Visual Studio 2019	400
7.6.1	C# 7.1	400
7.6.2	C# 7.2	401
7.6.3	C# 7.3	402
7.6.4	Visual Studio 2019 – Live Share	403
Teil II: WPF-Anwendungen		407
8	Einführung in WPF	409
8.1	Einführung	409
8.1.1	Was kann eine WPF-Anwendung?	410
8.1.2	Die eXtensible Application Markup Language	411
8.1.3	Verbinden von XAML und C#-Code	416
8.1.4	Zielplattformen	421
8.1.5	Applikationstypen	422
8.1.6	Vor- und Nachteile von WPF-Anwendungen	423
8.1.7	Weitere Dateien im Überblick	424
8.2	Alles beginnt mit dem Layout	426
8.2.1	Allgemeines zum Layout	426
8.2.2	Positionieren von Steuerelementen	428
8.2.3	Canvas	432
8.2.4	StackPanel	433
8.2.5	DockPanel	435
8.2.6	WrapPanel	437
8.2.7	UniformGrid	438
8.2.8	Grid	439
8.2.9	ViewBox	444
8.2.10	TextBlock	446
8.3	Das WPF-Programm	449
8.3.1	Die App-Klasse	450
8.3.2	Das Startobjekt festlegen	450
8.3.3	Kommandozeilenparameter verarbeiten	452
8.3.4	Die Anwendung beenden	453
8.3.5	Auswerten von Anwendungsereignissen	453
8.4	Die Window-Klasse	454
8.4.1	Position und Größe festlegen	454
8.4.2	Rahmen und Beschriftung	455
8.4.3	Das Fenster-Icon ändern	456
8.4.4	Anzeige weiterer Fenster	456
8.4.5	Transparenz	456

8.4.6	Abstand zum Inhalt festlegen	457
8.4.7	Fenster ohne Fokus anzeigen	458
8.4.8	Ereignisfolge bei Fenstern	458
8.4.9	Ein paar Worte zur Schriftdarstellung	459
8.4.10	Ein paar Worte zur Darstellung von Controls	462
8.4.11	Wird mein Fenster komplett mit WPF gerendert?	463
9	Übersicht WPF-Controls	465
9.1	Allgemeingültige Eigenschaften	465
9.2	Label	467
9.3	Button, RepeatButton, ToggleButton	468
9.3.1	Schaltflächen für modale Dialoge	468
9.3.2	Schaltflächen mit Grafik	470
9.4	TextBox, PasswordBox	471
9.4.1	TextBox	471
9.4.2	PasswordBox	473
9.5	CheckBox	474
9.6	RadioButton	476
9.7	ListBox, ComboBox	477
9.7.1	ListBox	477
9.7.2	ComboBox	480
9.7.3	Den Content formatieren	482
9.8	Image	484
9.8.1	Grafik per XAML zuweisen	484
9.8.2	Grafik zur Laufzeit zuweisen	484
9.8.3	Bild aus Datei laden	486
9.8.4	Die Grafikskalierung beeinflussen	487
9.9	MediaElement	488
9.10	Slider, ScrollBar	490
9.10.1	Slider	490
9.10.2	ScrollBar	492
9.11	ScrollViewer	492
9.12	Menu, ContextMenu	493
9.12.1	Menu	494
9.12.2	Tastenkürzel	495
9.12.3	Grafiken	496
9.12.4	Weitere Möglichkeiten	498
9.12.5	ContextMenu	498
9.13	ToolBar	499
9.14	StatusBar, ProgressBar	502
9.14.1	StatusBar	502
9.14.2	ProgressBar	504

9.15	Border, GroupBox, BulletDecorator	505
9.15.1	Border	505
9.15.2	GroupBox	506
9.15.3	BulletDecorator	507
9.16	RichTextBox	509
9.16.1	Verwendung und Anzeige von vordefiniertem Text	510
9.16.2	Neues Dokument zur Laufzeit erzeugen	511
9.16.3	Sichern von Dokumenten	512
9.16.4	Laden von Dokumenten	513
9.16.5	Texte per Code einfügen/modifizieren	514
9.16.6	Texte formatieren	515
9.16.7	EditingCommands	517
9.16.8	Grafiken/Objekte einfügen	518
9.16.9	Rechtschreibkontrolle	519
9.17	FlowDocumentPageViewer & Co.	520
9.17.1	FlowDocumentPageViewer	520
9.17.2	FlowDocumentReader	520
9.17.3	FlowDocumentScrollViewer	521
9.18	FlowDocument	521
9.18.1	FlowDocument per XAML beschreiben	522
9.18.2	FlowDocument per Code erstellen	524
9.19	Expander, TabControl	526
9.19.1	Expander	526
9.19.2	TabControl	527
9.20	Popup	529
9.21	TreeView	531
9.22	ListView	535
9.23	DataGrid	535
9.24	Calendar/DatePicker	536
9.25	Ellipse, Rectangle, Line und Co.	540
9.25.1	Ellipse	541
9.25.2	Rectangle	541
9.25.3	Line	542
10	Wichtige WPF-Techniken	543
10.1	Eigenschaften	543
10.1.1	Abhängige Eigenschaften (Dependency Properties)	543
10.1.2	Angehängte Eigenschaften (Attached Properties)	545
10.2	Einsatz von Ressourcen	545
10.2.1	Was sind eigentlich Ressourcen?	545
10.2.2	Wo können Ressourcen gespeichert werden?	546
10.2.3	Wie definiere ich eine Ressource?	547
10.2.4	Statische und dynamische Ressourcen	548

10.2.5	Wie werden Ressourcen adressiert?	550
10.2.6	Systemressourcen einbinden	550
10.3	Das WPF-Ereignismodell	551
10.3.1	Einführung	551
10.3.2	Routed Events	552
10.3.3	Direkte Events	554
10.4	Verwendung von Commands	554
10.4.1	Einführung zu Commands	555
10.4.2	Verwendung vordefinierter Commands	555
10.4.3	Das Ziel des Commands	557
10.4.4	Vordefinierte Commands	558
10.4.5	Commands an Ereignismethoden binden	558
10.4.6	Wie kann ich ein Command per Code auslösen?	560
10.4.7	Command-Ausführung verhindern	561
10.5	Das WPF-Style-System	561
10.5.1	Übersicht	561
10.5.2	Benannte Styles	562
10.5.3	Typ-Styles	564
10.5.4	Styles anpassen und vererben	565
10.6	Verwenden von Triggern	568
10.6.1	Eigenschaften-Trigger (Property Triggers)	568
10.6.2	Ereignis-Trigger	570
10.6.3	Daten-Trigger	571
10.7	Einsatz von Templates	572
10.7.1	Neues Template erstellen	572
10.7.2	Template abrufen und verändern	576
10.8	Transformationen, Animationen, StoryBoards	579
10.8.1	Transformationen	579
10.8.2	Animationen mit dem StoryBoard realisieren	584
11	WPF-Datenbindung	589
11.1	Grundprinzip	589
11.1.1	Bindungsarten	590
11.1.2	Wann eigentlich wird die Quelle aktualisiert?	592
11.1.3	Geht es auch etwas langsamer?	593
11.1.4	Bindung zur Laufzeit realisieren	594
11.2	Binden an Objekte	595
11.2.1	Objekte im XAML-Code instanziiieren	596
11.2.2	Verwenden der Instanz im C#-Quellcode	597
11.2.3	Anforderungen an die Quell-Klasse	598
11.2.4	Instanziiieren von Objekten per C#-Code	599
11.3	Binden von Collections	601
11.3.1	Anforderung an die Collection	601
11.3.2	Einfache Anzeige	602

11.3.3	Navigieren zwischen den Objekten	603
11.3.4	Einfache Anzeige in einer ListBox	605
11.3.5	DataTemplates zur Anzeigeformatierung	606
11.3.6	Mehr zu List- und ComboBox	607
11.3.7	Verwendung der ListView	609
11.4	Noch einmal zurück zu den Details	612
11.4.1	Navigieren in den Daten	612
11.4.2	Sortieren	614
11.4.3	Filtern	614
11.4.4	Live Shaping	615
11.5	Anzeige von Datenbankinhalten	617
11.5.1	Datenmodell per EF-Designer erzeugen	617
11.5.2	Die Programmoberfläche	621
11.5.3	Der Zugriff auf die Daten	622
11.6	Formatieren von Werten	624
11.6.1	IValueConverter	625
11.6.2	BindingBase.StringFormat-Eigenschaft	627
11.7	Das DataGrid als Universalwerkzeug	627
11.7.1	Grundlagen der Anzeige	627
11.7.2	UI-Virtualisierung	629
11.7.3	Spalten selbst definieren	629
11.7.4	Zusatzinformationen in den Zeilen anzeigen	631
11.7.5	Vom Betrachten zum Editieren	632
11.8	Praxisbeispiel – Collections in Hintergrundthreads füllen	633
Teil III: Technologien		637
12	Zugriff auf das Dateisystem	639
12.1	Grundlagen	639
12.1.1	Klassen für den Zugriff auf das Dateisystem	640
12.1.2	Statische versus Instanzen-Klasse	640
12.2	Übersichten	641
12.2.1	Methoden der Directory-Klasse	642
12.2.2	Methoden eines DirectoryInfo-Objekts	642
12.2.3	Eigenschaften eines DirectoryInfo-Objekts	642
12.2.4	Methoden der File-Klasse	643
12.2.5	Methoden eines FileInfo-Objekts	644
12.2.6	Eigenschaften eines FileInfo-Objekts	644
12.3	Operationen auf Verzeichnisebene	645
12.3.1	Existenz eines Verzeichnisses/einer Datei feststellen	645
12.3.2	Verzeichnisse erzeugen und löschen	646
12.3.3	Verzeichnisse verschieben und umbenennen	646
12.3.4	Aktuelles Verzeichnis bestimmen	647
12.3.5	Unterverzeichnisse ermitteln	647

12.3.6	Alle Laufwerke ermitteln	648
12.3.7	Dateien kopieren und verschieben	649
12.3.8	Dateien umbenennen	649
12.3.9	Dateiattribute feststellen	650
12.3.10	Verzeichnis einer Datei ermitteln	651
12.3.11	Alle im Verzeichnis enthaltenen Dateien ermitteln	652
12.3.12	Dateien und Unterverzeichnisse ermitteln	652
12.4	Weitere wichtige Klassen	653
12.4.1	Die Path-Klasse	653
12.4.2	Die Klasse FileSystemWatcher	654
12.4.3	Die Klasse ZipArchive	656
12.5	Datei- und Verzeichnisdialoge	657
12.5.1	OpenFileDialog	658
12.5.2	SaveFileDialog	660
12.6	Praxisbeispiele	661
12.6.1	Infos über Verzeichnisse und Dateien gewinnen	661
12.6.2	Eine Verzeichnisstruktur in die TreeView einlesen	666
13	Dateien lesen und schreiben	671
13.1	Grundprinzip der Datenpersistenz	671
13.1.1	Dateien und Streams	671
13.1.2	Die wichtigsten Klassen	672
13.1.3	Erzeugen eines Streams	673
13.2	Dateiparameter	673
13.2.1	FileAccess	673
13.2.2	FileMode	674
13.2.3	FileShare	674
13.3	Textdateien	675
13.3.1	Eine Textdatei beschreiben bzw. neu anlegen	675
13.3.2	Eine Textdatei lesen	676
13.4	Binärdateien	678
13.4.1	Lese-/Schreibzugriff	678
13.4.2	Die Methoden ReadAllBytes und WriteAllBytes	679
13.4.3	Erzeugen von BinaryReader/BinaryWriter	679
13.5	Sequenzielle Dateien	680
13.5.1	Lesen und Schreiben von strukturierten Daten	680
13.5.2	Serialisieren von Objekten	681
13.6	Dateien verschlüsseln	682
13.6.1	Das Methodenpärchen Encrypt/Decrypt	683
13.6.2	Verschlüsseln mit der CryptoStream-Klasse	683
13.7	Praxisbeispiele	684
13.7.1	Auf eine Textdatei zugreifen	684
13.7.2	Einen Objektbaum persistent speichern	688

13.7.3	Eine Datei verschlüsseln	693
13.7.4	PDFs erstellen/exportieren	698
13.7.5	Eine CSV-Datei erstellen	702
13.7.6	Eine CSV-Datei mit LINQ lesen und auswerten	703
13.7.7	Einen korrekten Dateinamen erzeugen	706
14	Asynchrone Programmierung	707
14.1	Übersicht	708
14.1.1	Multitasking versus Multithreading	708
14.1.2	Deadlocks	709
14.1.3	Racing	710
14.2	Programmieren mit Threads	711
14.2.1	Einführungsbeispiel	711
14.2.2	Wichtige Thread-Methoden	713
14.2.3	Wichtige Thread-Eigenschaften	715
14.2.4	Einsatz der ThreadPool-Klasse	716
14.3	Sperrmechanismen	717
14.3.1	Threading ohne lock	718
14.3.2	Threading mit lock	719
14.3.3	Die Monitor-Klasse	722
14.3.4	Mutex	726
14.3.5	Methoden für die parallele Ausführung sperren	727
14.3.6	Semaphore	728
14.4	Interaktion mit der Programmoberfläche	730
14.4.1	Die Werkzeuge	731
14.4.2	Einzelne Steuerelemente mit Invoke aktualisieren	731
14.4.3	Mehrere Steuerelemente aktualisieren	732
14.4.4	Ist ein Invoke-Aufruf nötig?	733
14.4.5	Und was ist mit WPF?	733
14.5	Timer-Threads	735
14.6	Asynchrone Programmierentwurfsmuster	736
14.6.1	Kurzübersicht	736
14.6.2	Polling	738
14.6.3	Callback verwenden	739
14.6.4	Callback mit Parameterübergabe verwenden	740
14.6.5	Callback mit Zugriff auf die Programmoberfläche	741
14.7	Asynchroner Aufruf beliebiger Methoden	743
14.7.1	Die Beispielklasse	743
14.7.2	Asynchroner Aufruf ohne Callback	744
14.7.3	Asynchroner Aufruf mit Callback und Anzeigefunktion	745
14.7.4	Aufruf mit Rückgabewerten (per Eigenschaft)	746
14.7.5	Aufruf mit Rückgabewerten (per EndInvoke)	747
14.8	Es geht auch einfacher - async und await	748
14.8.1	Der Weg von synchron zu asynchron	748

14.8.2	Achtung: Fehlerquellen!	751
14.8.3	Eigene asynchrone Methoden entwickeln	753
14.9	Praxisbeispiele	755
14.9.1	Prozess- und Thread-Informationen gewinnen	755
14.9.2	Ein externes Programm starten	758
15	Die Task Parallel Library	761
15.1	Überblick	761
15.1.1	Parallel-Programmierung	761
15.1.2	Möglichkeiten der TPL	764
15.1.3	Der CLR-Threadpool	764
15.2	Parallele Verarbeitung mit Parallel.Invoke	765
15.2.1	Aufrufvarianten	766
15.2.2	Einschränkungen	767
15.3	Verwendung von Parallel.For	767
15.3.1	Abbrechen der Verarbeitung	769
15.3.2	Auswerten des Verarbeitungsstatus	770
15.3.3	Und was ist mit anderen Iterator-Schrittweiten?	771
15.4	Collections mit Parallel.ForEach verarbeiten	772
15.5	Die Task-Klasse	773
15.5.1	Einen Task erzeugen	773
15.5.2	Den Task starten	774
15.5.3	Datenübergabe an den Task	775
15.5.4	Wie warte ich auf das Ende des Tasks?	777
15.5.5	Tasks mit Rückgabewerten	778
15.5.6	Die Verarbeitung abbrechen	781
15.5.7	Fehlerbehandlung	785
15.5.8	Weitere Eigenschaften	786
15.6	Zugriff auf das User Interface	788
15.6.1	Task-Ende und Zugriff auf die Oberfläche	788
15.6.2	Zugriff auf das UI aus dem Task heraus	789
15.7	Weitere Datenstrukturen im Überblick	791
15.7.1	Thread sichere Collections	791
15.7.2	Primitive für die Threadsynchroisation	792
15.8	Parallel LINQ (PLINQ)	792
15.9	Praxisbeispiele	792
15.9.1	BlockingCollection	792
15.9.2	PLINQ	796
16	Debugging, Fehlersuche und Fehlerbehandlung	797
16.1	Der Debugger	797
16.1.1	Allgemeine Beschreibung	797
16.1.2	Die wichtigsten Fenster	798

16.1.3	Debugging-Optionen	802
16.1.4	Praktisches Debugging am Beispiel	804
16.2	Arbeiten mit Debug und Trace	808
16.2.1	Wichtige Methoden von Debug und Trace	808
16.2.2	Besonderheiten der Trace-Klasse	812
16.2.3	TraceListener-Objekte	813
16.3	Caller Information	816
16.3.1	Attribute	816
16.3.2	Anwendung	816
16.4	Fehlerbehandlung	817
16.4.1	Anweisungen zur Fehlerbehandlung	817
16.4.2	try-catch	818
16.4.3	try-finally	822
16.4.4	Das Standardverhalten bei Ausnahmen festlegen	825
16.4.5	Die Exception-Klasse	826
16.4.6	Fehler/Ausnahmen auslösen	827
16.4.7	Eigene Fehlerklassen	827
16.4.8	Exceptionhandling zur Debugzeit	829
16.4.9	Code Contracts	829
17	JSON und XML in Theorie und Praxis	831
17.1	JSON – JavaScriptObjectNotation	831
17.1.1	Grundlagen	831
17.1.2	De-/Serialisierung mit JSON	832
17.2	XML – etwas Theorie	835
17.2.1	Übersicht	836
17.2.2	Der XML-Grundaufbau	837
17.2.3	Wohlgeformte Dokumente	838
17.2.4	Processing Instructions (PI)	840
17.2.5	Elemente und Attribute	841
17.3	XSD-Schemas	842
17.3.1	XML-Schemas in Visual Studio analysieren	843
17.3.2	XML-Datei mit XSD-Schema erzeugen	846
17.3.3	XSD-Schema aus einer XML-Datei erzeugen	847
17.4	Verwendung des DOM unter .NET	848
17.4.1	Übersicht	848
17.4.2	DOM-Integration in C#	849
17.4.3	Laden von Dokumenten	850
17.4.4	Erzeugen von XML-Dokumenten	851
17.4.5	Auslesen von XML-Dateien	853
17.4.6	Direktzugriff auf einzelne Elemente	854
17.4.7	Einfügen von Informationen	855
17.4.8	Suchen in den Baumzweigen	858

17.5	XML-Verarbeitung mit LINQ to XML	861
17.5.1	Die LINQ to XML-API	861
17.5.2	Neue XML-Dokumente erzeugen	863
17.5.3	Laden und Sichern von XML-Dokumenten	864
17.5.4	Navigieren in XML-Daten	866
17.5.5	Auswählen und Filtern	868
17.5.6	Manipulieren der XML-Daten	869
17.5.7	XML-Dokumente transformieren	870
17.6	Weitere Möglichkeiten der XML-Verarbeitung	874
17.6.1	Schnelles Suchen in XML-Daten mit XPathNavigator	874
17.6.2	Schnelles Auslesen von XML-Daten mit XmlReader	876
17.6.3	Erzeugen von XML-Daten mit XmlWriter	878
17.6.4	XML transformieren mit XSLT	880
17.7	Praxisbeispiele	882
17.7.1	Mit dem DOM in XML-Dokumenten navigieren	882
17.7.2	XML-Daten in eine TreeView einlesen	887
17.7.3	In Dokumenten mit dem XPathNavigator navigieren	891
18	Einführung in ADO.NET und Entity Framework	899
18.1	Eine kleine Übersicht	899
18.1.1	Die ADO.NET-Klassenhierarchie	899
18.1.2	Die Klassen der Datenprovider	900
18.1.3	Das Zusammenspiel der ADO.NET-Klassen	903
18.2	Das Connection-Objekt	904
18.2.1	Allgemeiner Aufbau	904
18.2.2	SqlConnection	904
18.2.3	Schließen einer Verbindung	905
18.2.4	Eigenschaften des Connection-Objekts	906
18.2.5	Methoden des Connection-Objekts	908
18.2.6	Der SqlConnectionStringBuilder	909
18.2.7	ConnectionString in der Konfigurationsdatei	910
18.3	Das Command-Objekt	911
18.3.1	Erzeugen und Anwenden eines Command-Objekts	911
18.3.2	Erzeugen mittels CreateCommand-Methode	912
18.3.3	Eigenschaften des Command-Objekts	912
18.3.4	Methoden des Command-Objekts	915
18.3.5	Freigabe von Connection- und Command-Objekten	917
18.4	Parameter-Objekte	919
18.4.1	Erzeugen und Anwenden eines Parameter-Objekts	919
18.4.2	Eigenschaften des Parameter-Objekts	920
18.5	Das SqlCommandBuilder-Objekt	921
18.5.1	Erzeugen	921
18.5.2	Anwenden	922

18.6	Das DataReader-Objekt	922
18.6.1	DataReader erzeugen	923
18.6.2	Daten lesen	923
18.6.3	Eigenschaften des DataReaders	924
18.6.4	Methoden des DataReaders	925
18.7	Das DataAdapter-Objekt	925
18.7.1	DataAdapter erzeugen	926
18.7.2	Command-Eigenschaften	927
18.7.3	Fill-Methode	928
18.7.4	Update-Methode	929
18.7.5	DataSet	930
18.8	Entity Framework	933
18.8.1	Überblick	933
18.8.2	CodeFirst	934
18.8.3	CodeFirst aus Datenbank	940
18.9	Praxisbeispiele	948
18.9.1	Wichtige ADO.NET-Objekte im Einsatz	948
18.9.2	Eine Aktionsabfrage ausführen	951
18.9.3	Eine StoredProcedure aufrufen	955
18.9.4	Daten mit Entity Framework laden und als JSON speichern	958
19	Weitere Techniken	969
19.1	Zugriff auf die Zwischenablage	969
19.1.1	Das Clipboard-Objekt	969
19.1.2	Zwischenablage-Funktionen für Textboxen	971
19.2	.NET-Reflection	971
19.2.1	Übersicht	972
19.2.2	Assembly laden	972
19.2.3	Mittels GetType und Type Informationen sammeln	973
19.2.4	Dynamisches Laden von Assemblies	975
19.3	Praxisbeispiele	977
19.3.1	Nutzer und Gruppen des aktuellen Systems ermitteln	977
19.3.2	Testen, ob Nutzer in einer Gruppe enthalten ist	980
19.3.3	Die IP-Adressen des Computers ermitteln	981
19.3.4	Diverse Systeminformationen ermitteln	983
Anhang A: Glossar	991	
Anhang B: Wichtige Dateiextensions	995	
Index	997	

Vorwort

C# ist die momentan 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.

Damit ist C# das ideale Werkzeug zum Programmieren beliebiger Komponenten für das Microsoft .NET Framework, beginnend bei Windows Forms- über WPF-, ASP.NET- und mobilen Anwendungen (mittlerweile auch für Android und iOS) bis hin zu systemnahen Applikationen.

Das vorliegende Buch ist ein Angebot für künftige 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

Wie Sie bereits dem Buchtitel entnehmen können, wagt das vorliegende Werk den Spagat zwischen einem Grundlagen- und einem Profibuch. Sinn eines solchen Buches kann es nicht sein, eine umfassende Schritt-für-Schritt-Einführung in C# 7.3 zu liefern oder all die Informationen noch einmal zur Verfügung zu stellen, die Sie in der Produktdokumentation (MSDN) ohnehin schon finden und von denen Sie in der Regel nur ein Mausklick oder die F1-Taste trennt.

- Für den *Einsteiger* möchte ich 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* möchte ich 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 und die Onlinekapitel sind in fünf Themenkomplexe gruppiert:

1. Grundlagen der Programmierung mit C# (Buch)
2. WPF-Anwendungen (Buch)
3. Technologien (Buch)
4. Windows Forms-Anwendungen (online)
5. Bonuskapitel (online)

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

Zu den Codebeispielen

Auf vielfachen Wunsch habe ich mich dazu entschlossen, ab dieser Auflage des Buches die Beispiele auch mit WPF zu erstellen. Der Einfachheit halber werden im ersten Teil die Beispiele weiter mit Windows Forms erstellt. Der zweite Teil des Buches behandelt dann ausführlich WPF und im dritten Teil „Technologien“ werden die Beispiele dann mit WPF dargestellt. Alle Beispieldaten dieses Buches und das Bonuskapitel zu Windows Forms können Sie unter der folgenden Adresse herunterladen:

www.hanserfachbuch.de

Geben Sie bitte im Suchfeld „Visual C# 2019“ ein und klicken Sie auf der Buchdetailseite auf die Registerkarte *Extras*.

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 **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.
- Beachten Sie die zu einigen Beispielen beigelegten *Liesmich.txt*-Dateien, die Sie auf besondere Probleme hinweisen.

Nobody is perfect

Sie werden in diesem Buch nicht alles finden, was C# bzw. das .NET Framework 4.7.2 zu bieten haben. Manches ist sicher in einem anderen Spezialtitel noch besser oder 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 mich gerne kontaktieren:

juergen.kotz@primetime-software.de

Ich hoffe, dass ich Ihnen mit diesem Buch einen nützlichen Begleiter bei der .NET-Programmierung zur Seite gestellt habe, der es verdient, seinen Platz nicht im Regal, sondern griffbereit neben dem Computer einzunehmen.

Jürgen Kotz

München, im Sommer 2019

Nachdem wir schon an der einen oder anderen Stelle auf Datenbindung zurückgegriffen haben, wollen wir uns jetzt direkt mit dieser Thematik beschäftigen.

Im Unterschied zu den Windows-Forms-Anwendungen sind Sie bei der Datenbindung nicht auf spezielle Controls oder Eigenschaften angewiesen. In einer WPF-Anwendung kann fast jede Eigenschaft (Abhängigkeitseigenschaft) an andere Eigenschaften gebunden werden.

Als Datenquelle können Sie beispielsweise

- Eigenschaften anderer WPF-Controls (Elemente),
- Ressourcen,
- XML-Elemente oder
- beliebige Objekte (auch ADO.NET-Objekte, z. B. *DataTable*)

verwenden.

■ 11.1 Grundprinzip

Zunächst wollen wir Ihnen das Grundprinzip der Datenbindung in WPF an einem recht einfachen Beispiel demonstrieren.

Beispiel 11.1: Datenbindung zwischen *Slider* und *ProgressBar*

XAML

Fügen Sie in ein *Window*, eine *ProgressBar* und einen *Slider* ein. Mit dem *Slider* soll der aktuelle Wert der *ProgressBar* direkt und ohne zusätzlichen Quellcode verändert werden.

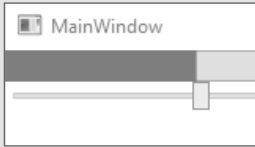
```
<StackPanel>
```

Hier sehen Sie auch schon den Ablauf: Das Ziel (*ProgressBar*) bindet seine Eigenschaft *Value* an die Quelle (*Slider*) mit deren Eigenschaft *Value*.

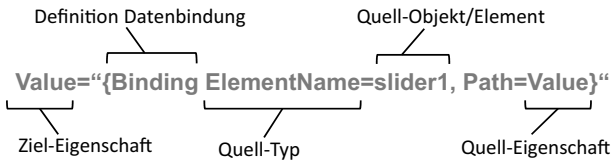
```
    <ProgressBar Height="20" Name="progressBar1" Maximum="100"
                Value="{Binding ElementName=slider1, Path=Value}"/>
    Separator Height="10"/>
    <Slider Name="slider1" Maximum="100" />
</StackPanel>
```

Ergebnis

Zur Laufzeit können Sie den *Slider* beliebig verändern, die *ProgressBar* passt sofort ihren Wert an:



Sehen wir uns noch einmal die Syntax im Detail an:



HINWEIS: Kann die Quelleigenschaft nicht automatisch in den Datentyp der Zieleigenschaft konvertiert werden, können Sie zusätzlich einen Typkonverter angeben (siehe dazu Abschnitt 11.6.1, „IValueConverter“).

11.1.1 Bindungsarten

Das vorhergehende Beispiel zeigte bereits recht eindrucksvoll, wie einfach sich Eigenschaften verschiedener Objekte miteinander verknüpfen lassen. Doch das ist noch nicht alles. Über ein zusätzliches Attribut *Mode* lässt sich auch bestimmen, in welche Richtungen die Bindung aktiv ist, d. h. ob die Werte nur von der Quelle zum Ziel oder auch umgekehrt übertragen werden. Die folgende Tabelle zeigt die möglichen Varianten:

Typ	Beschreibung
<i>OneTime</i>	Mit der Initialisierung wird der Wert einmalig von der Quelle zum Ziel kopiert. Danach wird die Bindung aufgehoben.
<i>OneWay</i>	Der Wert wird nur von der Quelle zum Ziel übertragen (readonly). Ändert sich der Wert des Ziels, wird die Bindung aufgehoben.
<i>OneWayToSource</i>	Der Wert wird vom Ziel zur Quelle übertragen (writeonly). Ändert sich der Wert der Quelle, bleibt die Bindung erhalten, eine Wertübertragung findet jedoch nicht statt.
<i>TwoWay</i>	(meist Defaultwert!) Werte werden zwischen Quelle und Ziel in beiden Richtungen übertragen.

¹ Bei Bindung an eine *ItemsSource* wird per Default *OneWay*-Binding verwendet.

Beispiel 11.2: Testen der verschiedenen Bindungsarten**XAML**

```

...
<Grid Margin="10">
  <Grid.ColumnDefinitions>
    <ColumnDefinition></ColumnDefinition>
    <ColumnDefinition></ColumnDefinition>
    <ColumnDefinition></ColumnDefinition>
  </Grid.ColumnDefinitions>
  <StackPanel>
    <Label Height="30">One Time</Label>
    <Label Height="30">One Way</Label>
    <Label Height="30">One Way To Source</Label>
    <Label Height="30">Two Way</Label>
  </StackPanel>
  <StackPanel Grid.Column="1">
    <Slider Name="s11" Maximum="100" Height="30"/>
    <Slider Name="s13" Maximum="100" Height="30"/>
    <Slider Name="s15" Maximum="100" Height="30"/>
    <Slider Name="s17" Maximum="100" Height="30"/>
  </StackPanel>
  <StackPanel Grid.Column="2">
    <Slider Name="s12" Maximum="100" Height="30"
      Value="{Binding ElementName=s11, Path=Value, Mode=OneTime}"/>
    <Slider Name="s14" Maximum="100" Height="30"
      Value="{Binding ElementName=s13, Path=Value, Mode=OneWay}"/>
    <Slider Name="s16" Maximum="100" Height="30"
      Value="{Binding ElementName=s15, Path=Value, Mode=OneWayToSource}"/>
    <Slider Name="s18" Maximum="100" Height="30"
      Value="{Binding ElementName=s17, Path=Value, Mode=TwoWay}"/>
  </StackPanel>
</Grid>

```

Ergebnis

Verschieben Sie ruhig einmal die *Slider* im Testprogramm. Jeweils der linke und der rechte *Slider* bilden eine Datenbindung und sollten auch das entsprechende Verhalten zeigen:



11.1.2 Wann eigentlich wird die Quelle aktualisiert?

Im obigen Beispiel scheint alles ganz einfach zu sein, Sie ziehen an einem Schieberegler und der andere bewegt sich mit (oder auch nicht, wie bei *OneTime*). Doch was ist, wenn Sie beispielsweise eine *TextBox* in einer Datenbindung verwenden? Hier stellt sich die Frage, **wann** der „gewünschte“ Wert wirklich in der *TextBox* steht.

Eine eingegebene Ziffer ist vielleicht nicht der richtige Wert, sie kann aber schon als gültiger Inhalt interpretiert werden. Nicht in jedem Fall möchte man deshalb sofort einen Datenaustausch zwischen Ziel und Quelle zulassen (bei *TwoWay* oder *OneWayToSource*).

Über das optionale Attribut *UpdateSourceTrigger* haben Sie direkten Einfluss darauf, wann die Aktualisierung **der Quelle** durchgeführt wird. Vier Varianten bieten sich dabei an:

- *Default*
Meist wird das *PropertyChanged*-Ereignis für die Datenübernahme genutzt, bei einigen Controls kann es auch *LostFocus* sein.
- *Explicit*
Die Datenübernahme muss „manuell“ per *UpdateSource*-Methode ausgelöst werden.
- *LostFocus*
Die Datenübernahme erfolgt bei Fokusverlust des Ziels.
- *PropertyChanged*
Die Datenübernahme erfolgt mit jeder Werteänderung. Dies kann bei komplexeren Abläufen zu Problemen führen, da der Abgleich, z. B. bei einem Schieberegler/einer Scrollbar, recht häufig vorgenommen wird.

Beispiel 11.3: Explizite Datenübernahme nur per **Enter**-Taste

XAML

```
<StackPanel>
  <TextBox Name="txt1">Hallo</TextBox>
  <TextBox Name="txt2"
    Text="{Binding ElementName=txt1, Path=Text,
      UpdateSourceTrigger=Explicit}"
    KeyDown="TextBox_KeyDown"/>
</StackPanel>
```

C#

```
private void TextBox_KeyDown(object sender, KeyEventArgs e)
{
    if (e.Key == Key.Enter)
    {
        txt2.GetBindingExpression(TextBox.TextProperty).UpdateSource();
    }
}
```



HINWEIS: Da die Bindung im XAML-Code vorgenommen wurde, müssen wir im C#-Code erst mit *GetBindingExpression* das *BindingExpression*-Objekt abrufen, um die *UpdateSource*-Methode aufzurufen.

11.1.3 Geht es auch etwas langsamer?

Am obigen Beispiel konnten Sie es ja schon beobachten: Sie verschieben den *Slider* und der zweite *Slider* reagiert sofort. So weit, so gut. Was aber, wenn Sie erst nach einiger Zeit auf die Veränderung reagieren wollen?

Hier hilft die mit WPF 4.5 eingeführte Eigenschaft *Delay* weiter. Diese verzögert die Datenübergabe um den angegebenen Wert (Millisekunden). Das heißt, erst wenn die Zeit nach einer Änderung verstrichen ist, wird der Wert weitergegeben. Jede Änderung in dieser Zeitspanne setzt den internen Timer zurück und lässt die Zeit erneut laufen. Bewegen Sie also den *Slider* dauernd hin und her, passiert nichts, erst nach der letzten Bewegung und dem Ablaufen der Zeit wird auch die Änderung berücksichtigt.

Beispiel 11.4: Zeitverzögerung bei Datenbindung

XAML

```
...
    <Slider Name="s10" Maximum="100" Height="30"
        Value="{Binding ElementName=s19,
        Path=Value, Mode=TwoWay, Delay=500}"/>
...

```

Was dem einen oder anderen als Spielerei vorkommen mag, ist ein fast unverzichtbares Feature im Zusammenhang mit größeren Datenmengen oder langsamen Datenverbindungen. Folgende Szenarien sind denkbar:

▪ 1:n-Beziehung

Änderungen in einer *ListBox* sollen sich nicht sofort auf die Detaildaten auswirken, sondern erst nachdem sich der Anwender für einen Datensatz final entschieden hat (Scrollen per Tastatur durch die Liste). Andernfalls kann es schnell zum Ruckeln oder Springen zwischen den Datensätzen kommen.

▪ Texteingaben

Nutzen Sie laufende Eingaben als Filter oder Suchwert, kann gerade bei großen Ergebnismengen eine Verzögerung bei der Eingabe auftreten (ein kurzer Filter mit Platzhalter hat meist große Ergebnismengen zur Folge).

▪ Anzeige großer Datenmengen

Mit der Auswahl in einer Liste soll eine größere Grafik angezeigt werden. Jede Änderung, z. B. beim Scrollen, führt im Normalfall zum Laden der Grafik. Hier ist eine entsprechende Verzögerung sinnvoll.

Alle obigen Fälle lassen sich natürlich auch mit einem eigenen *Timer* realisieren, aber warum kompliziert, wenn es jetzt auch wesentlich einfacher geht?

Eine Einschränkung sollten Sie allerdings beachten, auch wenn sie meist nicht von Bedeutung ist:



HINWEIS: Die Verzögerung gilt nur für eine Richtung der Datenbindung, d. h. nur für das Control, dem auch die Verzögerung zugeordnet ist. Ziehen Sie deswegen im obigen Beispiel auch einmal den rechten Slider, der linke reagiert sofort darauf.

11.1.4 Bindung zur Laufzeit realisieren

Nicht immer werden Sie mit den schon zur Entwurfszeit definierten Datenbindungen auskommen. Es ist aber auch kein Problem, die Datenbindung erst zur Laufzeit per C#-Code zu realisieren. Alles, was Sie dazu benötigen, ist ein *Binding*-Objekt, dessen Konstruktor Sie bereits den *BindingPath* zuweisen können. Legen Sie anschließend noch die *BindingSource* sowie gegebenenfalls den *Mode* (z. B. *OneWay*) fest. Letzter Schritt ist das eigentliche Binden mit der *SetBinding*-Methode des jeweiligen Controls.

Beispiel 11.5: Bindung zur Laufzeit realisieren

XAML

Unsere Testoberfläche:

```
<Window x:Class="DatenbindungStart.MainWindow"
...
    Title="MainWindow" Height="300" Width="500" Loaded="Window_Loaded">
...
    <StackPanel>
        <Label Name="Label1"></Label>
        <Button "Button1" Click="Button_Click">Test</Button>
    </StackPanel>
</Window>
```

C#

Mit dem Laden des Fensters erzeugen wir die Bindung wie oben beschrieben:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
```

Wir binden an einen Button:

```
    Binding binding = new Binding("Content");
    binding.Source = button1;
    binding.Mode = BindingMode.OneWay;
```

Binden an den Content:

```
    label1.SetBinding(Label.ContentProperty, binding);
}
```

Und hier verändern wir die Beschriftung des Buttons:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    button1.Content = "Ein neuer Text";
}
}
```

Test

Nach dem Start dürfte im Label zunächst „Test“ stehen, die ursprüngliche Button-Beschriftung. Nach einem Klick auf die Schaltfläche ändern sich sowohl die Button-Beschriftung als auch die Label-Beschriftung.

Ein neuer Text

Ein neuer Text

Die Bindung selbst können Sie recht einfach wieder aufheben, indem Sie der Ziel-Eigenschaft der Bindung einen neuen Wert zuweisen.

Beispiel 11.6: Bindung zur Laufzeit aufheben

C#

Entweder so:

```
label1.Content = "Bindung beendet";
```

Oder so:

```
label1.ClearValue(Label.ContentProperty);
```

■ 11.2 Binden an Objekte

Nachdem wir uns bereits mit dem Binden an Oberflächenelemente vertraut gemacht haben, wollen wir jetzt den Schritt hin zu selbstdefinierten Objekten gehen.

Prinzipiell bieten sich zwei Varianten der Instanziierung von Objekten an:

- Sie instanziierten die Objekte in XAML (in einem Resource-Abschnitt).
- Sie instanziierten wie bisher die Objekte im Quellcode.



HINWEIS: Von der Möglichkeit, Objekte im XAML-Code zu instanziierten, ist abzuraten. Einerseits wird mit Klassen gearbeitet, die per Code definiert und verarbeitet werden, andererseits wird die Instanz in der Oberfläche, d. h. im XAML-Code, erzeugt. Das ist sicher nicht der Weisheit letzter Schluss. Gerade die üble Vermischung von Code und Oberfläche sollte eigentlich vermieden werden.

Fragwürdig werden Beispielprogramme dann, wenn im C#-Quellcode das zunächst in XAML erzeugte Objekt per *FindResource* gesucht wird (siehe folgender Abschnitt). Das pervertiert doch jede Form der sauberen Programmierung.

Wohlgermerkt wollen wir nicht die komplette Datenbindung im Code realisieren. Das ist sicher zu aufwendig und auch nicht notwendig. Doch aus Sicht des Entwicklers sollte nicht die Oberfläche (XAML), sondern der Code im Mittelpunkt des Programms stehen.

11.2.1 Objekte im XAML-Code instanziiieren

Erster Schritt nach der Definition der Klasse ist das Importieren des entsprechenden Namespace in die XAML-Datei, andernfalls können Sie auch nicht darauf Bezug nehmen.

Beispiel 11.7: Import des aktuellen Namespace *Datenbindung* in die XAML-Datei

XAML

```
<Window x:Class="ObjectBinding.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="clr-namespace:ObjectBinding"
  ...
```

Nachdem in XAML die entsprechende Klasse bekannt ist, kann diese auch verwendet werden, um eine eigene Instanz zu erzeugen.

Beispiel 11.8: Erzeugen der Instanz im XAML-Code (wir nutzen eine Klasse *Person*)

C#

```
public class Person
{
    public string Nachname { get; set; }
    public string Vorname { get; set; }
}
```

XAML

```
...
<Window.Resources>
  <local:Person x:Key="pers" Nachname="Mooshammer" Vorname="Elisabeth" />
</Window.Resources>
```

Die Werte im Einzelnen:

- *local*: Der Bezug auf den Namespace-Alias
- *Person*: Der Klassenname
- *x:Key*: Der Schlüssel, unter dem die Instanz verwendet werden kann
- *Nachname*, *Vorname*: Das Setzen einzelner Eigenschaften für die Instanz von *Person*

Letzter Schritt: Wir nutzen die Möglichkeiten der Datenbindung und binden zwei *TextBox*en an die Eigenschaften *Nachname* und *Vorname*.

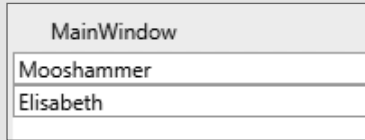
Beispiel 11.9: Bindung an das neue Objekt erzeugen

XAML

```
<StackPanel>
  <TextBox Text="{Binding Source={StaticResource pers}, Path=Nachname}" />
  <TextBox Text="{Binding Source={StaticResource pers}, Path=Vorname}" />
</StackPanel>
```

Ergebnis

Schon zur Entwurfszeit dürfte in den beiden *TextBox*en der gewünschte Inhalt auftauchen:



Wem das zu viel Schreibarbeit ist, der kann mit dem *DataContext* auch eine alternative Variante der Zuweisung nutzen. Diese Eigenschaft bietet zunächst eine Alternative zur Zuweisung von *Source*, hat jedoch zusätzlich die Fähigkeit, von übergeordneten auf untergeordnete Elemente vererbt zu werden. Damit können Sie beispielsweise einem *Panel* oder sogar dem gesamten *Window* einen *DataContext* zuweisen und diesen in allen enthaltenen Elementen nutzen.

Beispiel 11.10: Vereinfachung durch Verwendung eines *DataContext*

XAML

```
<StackPanel DataContext="{StaticResource pers}">
  <TextBox Text="{Binding Path=Nachname}" />
  <TextBox Text="{Binding Path=Vorname}" />
  ...
```

Sie sparen sich so die Angabe von *Source* bei jedem einzelnen Element.

11.2.2 Verwenden der Instanz im C#-Quellcode

Sicher nicht ganz abwegig ist der Wunsch, zur Laufzeit per C#-Code auch mit dem Objekt zu arbeiten, um z. B. die Werte mit einer *MessageBox* anzuzeigen.

Hier wird die Programmierung dann schon recht windig, müssen Sie doch zunächst die entsprechende Ressource des *Window* suchen und typisieren; möglichen Fehlern ist Tür und Tor geöffnet.

Beispiel 11.11: Anzeige der Werte eines per XAML instanziierten Objekts

C#

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Person pers = (Person)FindResource("pers");
    MessageBox.Show($"{pers.Nachname}, {pers.Vorname}");
}
```

Aus Sicht eines Programmierers sieht das doch ziemlich merkwürdig aus, auch wenn sich hier der XAML-Profi freut, dass er sogar eine Instanz plus Wertzuweisung per XAML-Code realisiert hat.

Doch was passiert eigentlich mit der Datenbindung, wenn wir der Instanz ein paar neue Werte zuweisen? Ein Test ist schnell realisiert:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Person pers = (Person)FindResource("pers");
    pers.Nachname = "Eberhofer";
}
```

Der nachfolgende Blick auf die Oberfläche dürfte in den meisten Fällen für Ernüchterung sorgen: Haben Sie Ihre .NET-Klasse (in diesem Fall *Person*) nicht entsprechend angepasst, passiert überhaupt nichts und in den *TextBoxen* stehen nach wie vor die alten Werte.

11.2.3 Anforderungen an die Quell-Klasse

Was ist hier schiefgelaufen? Eigentlich nichts, die neuen Werte stehen wirklich im Objekt, sie werden aber nicht angezeigt, weil die darstellenden Elemente von einer Wertänderung nichts mitbekommen haben. Wir müssen diese quasi „wecken“, und was eignet sich dafür besser als ein Ereignis?

Auch hier gibt es bereits eine fertige Lösung:



HINWEIS: Implementieren Sie in Ihrer Klasse das Interface *INotifyPropertyChanged* (Namespace *System.ComponentModel*).

Beispiel 11.12: Unsere Klasse *Schüler* mit implementiertem *NotifyPropertyChanged*-Ereignis

C#

```
using System.ComponentModel;

namespace ObjectBinding
{
    public class Person : INotifyPropertyChanged
    {
        string nachname;
        string vorname;

        public event PropertyChangedEventHandler PropertyChanged;

        public string Vorname
        {
            get { return vorname; }
            set
            {
                vorname = value;
                NotifyPropertyChanged(nameof(Vorname));
            }
        }
    }
}
```



```
public string Nachname
{
    get { return nachname; }
    set
    {
        nachname = value;
        NotifyPropertyChanged(nameof(Nachname));
    }
}

public override string ToString()
{
    return $"{Nachname}, {Vorname}";
}

private void NotifyPropertyChanged(string info)
{
    if (PropertyChanged != null)
    {
        PropertyChanged(this, new PropertyChangedEventArgs(info));
    }
}
}
```



HINWEIS: Alternativ können Sie natürlich auch Abhängigkeitseigenschaften definieren. Diese verfügen „ab Werk“ über die erforderliche Benachrichtigung an die gebundenen Elemente, sie erfordern aber einen höheren Programmieraufwand.



HINWEIS: Damit die Klasse auch im XAML-Code instanziiert werden kann, muss diese über einen parameterlosen Konstruktor verfügen.

11.2.4 Instanzieren von Objekten per C#-Code

Eigentlich könnten wir Ihnen an dieser Stelle noch weitere Möglichkeiten zeigen, wie Sie in XAML Objekte erzeugen bzw. zuweisen können, aber dies ist weder sinnvoll noch besonders übersichtlich. Wir wollen uns stattdessen mit der Vorgehensweise bei vorhandenen, d. h. per Code erzeugten, .NET-Objekten beschäftigen.

Zunächst bleiben wir bei unserem einfachen Beispiel mit der Instanz der Klasse *Person*.

Beispiel 11.13: Verwendung von instanziierten Objekten in XAML**C#**

Zunächst die Instanziierung:

```
..
public partial class MainWindow : Window
{
    public Person Person { get; set;}

    public MainWindow()
    {
        InitializeComponent();
    }
}
```

Instanz erzeugen und Werte zuweisen:

```
Person = new Person()
    { Nachname = "Birkenberger", Vorname = "Rudi"};
```

Hier legen wir per C#-Code den *DataContext* fest:

```
stackPanel1.DataContext = Person;
}
```

Die spätere Abfrage des Objekts stellt jetzt überhaupt kein Problem dar, die Instanz liegt ja bereits vor:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show($"{Person.Nachname}, {Person.Vorname}");
}
...

```

XAML

Der XAML-Code:

```
..
<StackPanel Name="stackPanel1">
    <TextBox Text="{Binding Path=Nachname}" />
    <TextBox Text="{Binding Path=Vorname}" />
    <Button Click="Button_Click">Name anzeigen</Button>
</StackPanel>
</Window>
```

Der Vorteil dieser Vorgehensweise: Sie entscheiden, wie und wann die Instanz erzeugt wird, können vorher noch diverse Methoden aufrufen, profitieren von der Syntaxprüfung und haben einen lesbaren Code.

Der einzige Nachteil: Sie haben keine Wertanzeige zur Entwurfszeit. Im XAML-Code ist es nicht sofort erkennbar, welches Objekt zugeordnet wird. Dies ist allerdings auch gleich wieder der Vorteil, mit einem Klick können Sie einen neuen *DataContext* zuweisen und eine andere Instanz bearbeiten.

■ 11.3 Binden von Collections

Die bisherigen Ausführungen dürften zwar schon das Potenzial der Datenbindung demonstriert haben, doch nach der Pflicht kommt jetzt die Kür, d. h. die Arbeit mit einer Reihe von Objekten (Collections). Diese sind vor allem dann interessant, wenn Sie Objekte von Datenbanken abrufen, um sie in Eingabedialogen oder gleich in Listenelementen darzustellen. Ausgangspunkt können hier Geschäftsobjekte, LINQ-Abfragen etc. sein.



HINWEIS: Im vorliegenden Abschnitt werden wir uns zunächst auf eine „selbstgestrickte“ Collection beziehen (wir verwenden das *Person*-Objekt aus dem vorhergehenden Abschnitt). Ab Abschnitt 11.5 geht es dann mit Datenbindung in Verbindung mit *LINQ to SQL*-Abfragen weiter.

11.3.1 Anforderung an die Collection

Wie auch bei der Klassendefinition für das einzelne Objekt, werden auch an die Collection einige Anforderungen gestellt. Zwar können die WPF-Elemente durch die Verwendung der *INotifyPropertyChanged*-Schnittstelle auf Änderungen einzelner Objekteigenschaften reagieren, das Hinzufügen oder Löschen von ganzen Objekten ist davon aber nicht betroffen. Aus diesem Grund bietet WPF auch hier ein genormtes Interface für die Rückmeldung an: *INotifyCollectionChanged*.



HINWEIS: Grundvoraussetzung für die Anzeige von Listen ist die Verwendung des *IEnumerable*-Interfaces.

Wollen Sie es sich leicht machen, können Sie direkt Objekte der Klasse *ObservableCollection* (Namespace *System.Collections.ObjectModel*) erzeugen.

Beispiel 11.14: (*Fortsetzung*) Erzeugen und Verwenden einer geeigneten Klasse für die Datenbindung von Collections

C#

```
using System.Collections.ObjectModel;
...
public partial class MainWindow : Window
{
```

Eine Collection von Personen:

```
public ObservableCollection<Person> Abteilung { get; set; }
```

Im Konstruktor des *Window* erzeugen wir eine Instanz und füllen diese mit einigen Datensätzen:

```
public MainWindow()
{
    Abteilung = new ObservableCollection<Person>();
    Abteilung.Add(new Person()
        { Nachname = "Eberhofer", Vorname = "Franz" });
    Abteilung.Add(new Person()
        { Nachname = "Birkenberger", Vorname= "Rudi" });
    Abteilung.Add(new Person()
        { Nachname = "Simmerl", Vorname ="Max" });
    InitializeComponent();
}
```

Hier dürften Sie die Verbindung zur bisherigen Vorgehensweise sehen, die Collection wird als *DataContext* für das Fenster und damit für alle untergeordneten Elemente ausgewählt:

```
    DataContext = Abteilung;
}
```

11.3.2 Einfache Anzeige

Damit können wir uns zunächst der einfachen Anzeige, z. B. in *TextBoxen*, widmen.

Beispiel 11.15: (*Fortsetzung*) Binden von *TextBoxen* an die Collection

XAML

```
<StackPanel Background="Aqua">
    <Label Content="Nachname:" />
```

Hier werden die *TextBoxen* an die Eigenschaften der Collection bzw. an das aktive Objekt der Collection gebunden (dies ist durch eine View bestimmt, siehe ab Abschnitt 11.4):

```
<TextBox Text="{Binding Path=Nachname}" />
<Label Content="Vorname:" />
```

Beachten Sie auch diese mögliche Kurzsyntax, die auf die Angabe von *Path* verzichtet:

```
<TextBox Text="{Binding Vorname}" />
```

Einige Schaltflächen definieren:

```
<StackPanel Orientation="Horizontal">
    <Button Content=" <" Click="ButtonPrevious_Click" />
    <Button Content=" >" Click="ButtonNext_Click"/>
    <Button Content=" Neu " Click="ButtonNeu_Click"/>
    <Button Content=" Löschen " Click="ButtonLoeschen_Click"/>
</StackPanel>
</StackPanel>
```

Ergebnis

Das erzeugte Formular:

Nach dem Start dürfte schon etwas in den Textfeldern angezeigt werden, ein Navigieren zwischen den einzelnen Datensätzen (Objekten) ist allerdings noch nicht möglich.

11.3.3 Navigieren zwischen den Objekten



HINWEIS: An dieser Stelle müssen wir etwas vorgeifen, Abschnitt 11.4 geht auf dieses Thema im Detail ein.

Navigation zwischen Datensätzen bedeutet, dass auch irgendwo ein aktueller Datensatz gespeichert wird und entsprechende Navigationsmethoden zur Verfügung stehen. Auch bei intensiver Suche werden Sie aber derartige Eigenschaften zunächst nicht finden.

WPF erzeugt beim Binden von Collections automatisch eine Sicht auf die eigentliche Collection. Diese Sicht verwaltet den aktuellen Datensatz, bietet Navigationsmethoden an und ermöglicht das Filtern und Sortieren der Daten².

Diese automatisch erzeugte Sicht können Sie mit der Methode `CollectionViewSource.GetDefaultView` für eine spezifische Collection abrufen.

Beispiel 11.16: (Fortsetzung) Abrufen und Verwenden der `DefaultView` für unsere Collection

C#

Wir erweitern die Liste der lokalen Variablen, um die Sicht zu speichern:

```
private ICollectionView view;
...
public MainWindow()
{
    Abteilung = new ObservableCollection<Person>();
...

```

² Derartige Sichten können Sie auch selbst erstellen und quasi als Schicht zwischen Daten und `DataContext` schieben.

Vergessen Sie nicht, einen Verweis auf den Namespace *System.ComponentModel* zu setzen.

Im Konstruktor rufen wir die Sicht ab:

```
        view = CollectionViewSource.DefaultView(Abteilung);  
    }
```

Jetzt können wir mit dieser Sicht auch die Navigation zwischen den einzelnen Elementen der Collection realisieren.

Nächstes Objekt:

```
private void ButtonNext_Click(object sender, RoutedEventArgs e)  
{  
    view.MoveCurrentToNext();  
    if (view.IsCurrentAfterLast)  
    {  
        view.MoveCurrentToLast();  
    }  
}
```

Vorhergehendes Objekt:

```
private void ButtonPrevious_Click(object sender, RoutedEventArgs e)  
{  
    view.MoveCurrentToPrevious();  
    if (view.IsCurrentBeforeFirst)  
    {  
        view.MoveCurrentToFirst();  
    }  
}
```

Wir fügen zum Testen ein neues Objekt zur Laufzeit in die Collection ein:

```
private void Button_Neu(object sender, RoutedEventArgs e)  
{  
    Abteilung.Add(  
        new Person() { Nachname = "Moratschek", Vorname = "Willi" });  
}
```

Auch das Löschen von Objekten ist auf diesem Wege möglich:

```
private void ButtonDelete_Click(object sender, RoutedEventArgs e)  
{  
    Abteilung.Remove(view.CurrentItem as Person);  
}
```

Nach dem Start des Beispiels können Sie zwischen den Objekten „navigieren“, Objekte hinzufügen und diese auch wieder löschen. Das Ganze kommt Ihnen sicherlich unter dem Stichwort „Datenbanknavigator“ bekannt vor.

11.3.4 Einfache Anzeige in einer ListBox

Das Anzeigen von Einzeldatensätzen ist ja schon ganz gut, wie aber steht es mit dem Füllen von ganzen Listenfeldern?

Auch hier können Sie, dank Datenbindung, schnell zu brauchbaren Ergebnissen kommen.

Beispiel 11.17: (Fortsetzung) Anbinden einer *ListBox* an unsere Collection

C#

Es genügt zunächst die einfache Zuweisung von "{Binding}" an die *ItemsSource*:

```
<ListBox Height="100" IsSynchronizedWithCurrentItem="True"
        Name="listBox1" ItemsSource="{Binding}"/>
```

Der Hintergrund: Da die Collection bereits direkt an das Formular gebunden ist, brauchen wir hier nicht weitere Eigenschaften zu spezifizieren. Alternativ könnten Sie hier auch die Collection per *DataContext* zuweisen.

Und wofür ist das Attribut *IsSynchronizedWithCurrentItem* verantwortlich? Hier sollten Sie sich an unsere Sicht erinnern, die auch den aktuellen „Satzzeiger“ verwaltet. Nur wenn Sie das Attribut auf *True* setzen, wird das aktuelle Item mit dem „Satzzeiger“ synchronisiert (dies gilt für beide Richtungen).

Ergebnis

Die angezeigte *ListBox* zur Laufzeit:

Eberhofer, Franz
Birkenberger, Rudi
Simmerl, Max



HINWEIS: Die *ItemsSource*-Eigenschaft kann nur verwendet werden, wenn die *Items*-Collection eines *ItemsControl* leer ist. Falls nicht, wird Ihre Anwendung eine *InvalidOperationException* auslösen.

Doch woher „weiß“ die *ListBox* eigentlich, welche Eigenschaften des *Schüler*-Objekts in der Liste darzustellen sind? Antwort: Sie weiß es nicht und verwendet in diesem Fall einfach die *ToString*-Methode des betreffenden Objekts. Wenn Sie jetzt mal kurz in Abschnitt 11.2.3, „Anforderungen an die Quell-Klasse“, nachschlagen, werden Sie feststellen, dass wir in unserer Vorahnung bereits die *ToString*-Methode überschrieben haben und damit eine Kombination aus *Nachname* und *Vorname* zurückgeben (siehe oben).

Verwendung von *DisplayMemberPath*

Natürlich ist das Überschreiben der *ToString*-Methode nicht der Weisheit letzter Schluss und so ist es sicher sinnvoll, noch einen anderen Weg zur Auswahl des anzuzeigenden Members

zu unterstützen. Genau für diesen Zweck wird die *DisplayMemberPath*-Eigenschaft angeboten. Diese bestimmt, welcher Member für den Text des Listeneintrags verwendet wird.

Beispiel 11.18: Verwendung von *DisplayMemberPath* für die Auswahl der anzuzeigenden Eigenschaft

XAML

```
<ListBox Height="100" IsSynchronizedWithCurrentItem="True"
        ItemsSource="{Binding}" DisplayMemberPath="Nachname" />
```

Leider genügt jedoch auch diese Version der Anzeigeformatierung nicht immer und so landen wir unweigerlich bei den *DataTemplates*.

11.3.5 DataTemplates zur Anzeigeformatierung

Obige Art der Datenbindung dürfte in vielen Fällen wohl kaum genügen. Die WPF-Entwickler haben aber auch für diesen Fall vorgesorgt und mit dem *DataTemplate* ein mächtiges Werkzeug geschaffen.

Das Prinzip: Jeder *ListBox/ComboBox* können Sie ein *DataTemplate* zuweisen, das dafür verantwortlich ist, wie das einzelne Item aufgebaut ist (quasi eine Schablone in die die Daten eingefügt werden). Und da WPF im Content eines Items fast jede Zusammenstellung von Elementen akzeptiert, können Sie hier Formatierungen beliebiger Art erzeugen (natürlich im Rahmen der XAML-Vorgaben).

Beispiel 11.19: (*Fortsetzung*) Wir wollen in der *ListBox* eine zweispaltige Anzeige realisieren (links der Nachname, rechts der Nachname und der Vorname).

XAML

In den Ressourcen (z. B. Window) erzeugen Sie das erforderliche *DataTemplate*:

```
<Window.Resources>
  <DataTemplate x:Key="AbteilungsListTemplate">
```

Das Layout bestimmen Sie:

```
  <StackPanel Orientation="Horizontal">
```

Bei der Zuweisung von Inhalten können Sie jetzt direkt auf die Eigenschaften zugreifen:

```
    <TextBlock VerticalAlignment="Top" Width="100"
              Text="{Binding Path=Nachname}" />
    <StackPanel>
      <TextBlock Text="{Binding Path=Nachname}" />
      <TextBlock Text="{Binding Path=Vorname}" />
    </StackPanel>
  </StackPanel>
</DataTemplate>
</Window.Resources>
```

...

Last, but not least, müssen Sie der *ListBox* auch noch das Template zuweisen:

```
<ListBox Height="120" IsSynchronizedWithCurrentItem="True"
Name="listBox1" ItemsSource="{Binding}"
ItemTemplate="{StaticResource AbteilungsListTemplate}"/>
```

Ergebnis

Die erzeugte *ListBox*:

Eberhofer	Eberhofer Franz
Birkenberger	Birkenberger Rudi
Simmerl	Simmerl Max

Dass Sie hier auch mit Grafiken, optischen Effekten, Kontextmenüs etc. arbeiten können, sollte nach den Darstellungen der vorhergehenden Kapitel klar sein.

11.3.6 Mehr zu List- und ComboBox

An dieser Stelle wollen wir uns noch einige spezielle Eigenschaften von *List*- und *ComboBox* ansehen, die in der täglichen Programmierpraxis von Bedeutung sind.

SelectedItem

Möchten Sie Einträge in der *ListBox* auswählen bzw. bestimmen, der wievielte Eintrag (Index) in der Liste markiert ist, können Sie die *SelectedItem*-Eigenschaft verwenden.

Beispiel 11.20: Auswahl des zweiten Eintrags

```
C#
private void ButtonZweitenSatzAuswaehlen_Click(
    object sender, RoutedEventArgs e)
{
    listBox1.SelectedItem = 1;
}
```

SelectedItem/SelectedItems

Möchten Sie das markierte Listenelement selbst abrufen bzw. das damit verbundene Objekt, verwenden Sie die *SelectedItem*-Eigenschaft. Alternativ können Sie auch eine Liste der markierten Einträge mit *SelectedItems* abrufen.



HINWEIS: Die Collection *SelectedItem* steht Ihnen nur zur Verfügung, wenn Sie *SelectionMode* auf *Multiple* festgelegt haben.

Beispiel 11.21: Verwendung *SelectedItem* / *SelectedItems*

C#

Wir nutzen unsere überschriebene *ToString*-Methode:

```
MessageBox.Show(listBox1.SelectedItem.ToString());
```

Wir greifen direkt auf einen Member (typisieren nicht vergessen) zu:

```
MessageBox.Show((listBox1.SelectedItem as Person)?.Nachname);
```

Wir zeigen alle markierten Einträge an:

```
foreach (Person p in listBox1.SelectedItems)
{
    MessageBox.Show(p.Nachname);
}
```

SelectedValuePath und SelectedValue

Mit *SelectedValuePath* können Sie festlegen, welcher Member von der Eigenschaft *SelectedValue* zurückgegeben wird. Dies ist im Zusammenhang mit Datenbanken meist der Primärindex der Tabelle, mit dem Sie einen Datensatz eindeutig identifizieren können.



HINWEIS: Ist *SelectedValuePath* nicht festgelegt, gibt *SelectedValue* das komplette Objekt zurück (entspricht *SelectedItem*).

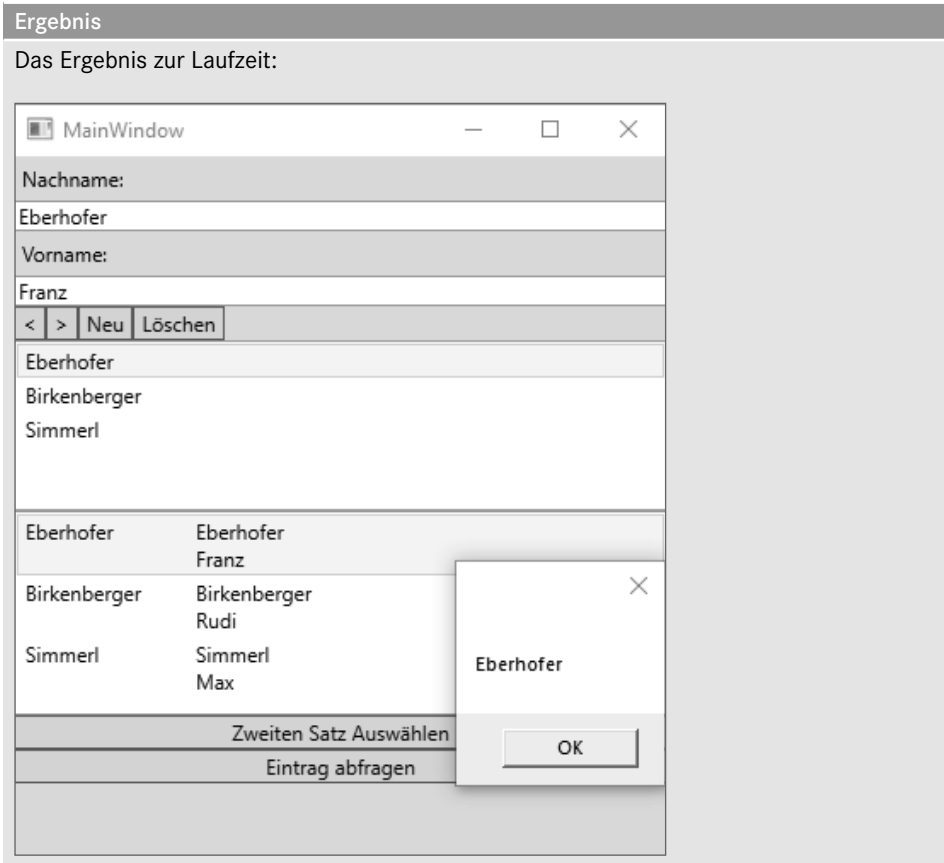
Beispiel 11.22: Verwendung *SelectedValuePath* und *SelectedValue*

XAML

```
<ListBox IsSynchronizedWithCurrentItem="True" Name="listBox1"
ItemsSource="{Binding}" DisplayMemberPath="Nachname"
SelectedValuePath="Nachname"/>
```

C#

```
private void Button2_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show(listBox1.SelectedValue.ToString());
}
```



11.3.7 Verwendung der ListView

Im vorhergehenden Kapitel hatten wir die *ListView* ja bereits kurz gestreift (Trockenschwimmen), an dieser Stelle zeigen wir Ihnen die *ListView* „in Action“.

Einfache Bindung

Prinzipiell ist die *ListView* der *ListBox* recht ähnlich. Die Anbindung der Einträge erfolgt ebenfalls per *ItemsSource*, die Auswahl bzw. Bestimmung (*SelectedItem*, *SelectedValue* etc.) der markierten Einträge ist analog realisiert.

Neu ist, dass die *ListView* über Spaltenköpfe verfügt, die Sie getrennt konfigurieren können (*GridViewColumnHeader*) und gegebenenfalls auch für das Sortieren (siehe 2. Beispiel) verwenden können.

Ein weiterer Unterschied ist die Unterstützung von verschiedenen Ansichten, von denen jedoch nur die *GridView* vordefiniert ist. Im Weiteren werden wir uns auch nur auf diese Ansicht beschränken.

Beispiel 11.23: (Fortsetzung) Anzeige der Collection-Daten in einer *ListView*

XAML

Zuweisen der Datenquelle (Übernahme von *Window.DataContext*):

```
<ListView Name="listView1" Height="100" IsSynchronizedWithCurrentItem="True"
          ItemsSource="{Binding}">
  <ListView.View>
```

Hier wird die *GridView* definiert:

```
<GridView>
```

Die einzelnen Spalten definieren:

```
<GridView.Columns>
```

Und jetzt wird es einfach, binden Sie lediglich die gewünschten Eigenschaften an die einzelnen Spalten der *GridView*:

```
<GridViewColumn Header="Name"
                 DisplayMemberBinding="{Binding Path=Nachname}" />
<GridViewColumn Header="Vorname"
                 DisplayMemberBinding="{Binding Path=Vorname}" />
</GridView.Columns>
</GridView>
</ListView.View>
</ListView>
```

Ergebnis

Das Endergebnis zur Laufzeit:

Name	Vorname
Eberhofer	Franz
Birkenberger	Rudi
Simmerl	Max

Sortieren der Einträge

Wie schon erwähnt, können Sie die Spaltenköpfe auch für das Sortieren der Einträge nutzen. Ein einfaches Beispiel zeigt die Vorgehensweise:

Beispiel 11.24: Sortieren nach Klick auf den jeweiligen Spaltenkopf

XAML

Unsere Änderung in der Seitenbeschreibung:

```
<ListView Name="listView1" IsSynchronizedWithCurrentItem="True"
          ItemsSource="{Binding}">
  <ListView.View>
    <GridView>
```

```

<GridView.Columns>
  <GridViewColumn DisplayMemberBinding="{Binding Path=Nachname}" >
    <GridViewColumnHeader Click="SortClick" Content="Nachname" />
  </GridViewColumn>
  <GridViewColumn DisplayMemberBinding="{Binding Path=Vorname}" >
    <GridViewColumnHeader Click="SortClick" Content="Vorname" />
  </GridViewColumn>
</GridView.Columns>
</GridView>
</ListView.View>
</ListView>

```

C#

Der Quellcode fällt recht kurz aus:

```

using System.ComponentModel;
using System.Collections.ObjectModel;
...
private void SortClick(object sender, RoutedEventArgs e)
{

```

Zunächst die betreffende Spalte bestimmen:

```

    GridViewColumnHeader spalte = sender as GridViewColumnHeader;

```

Die Default-View bestimmen:

```

    ICollectionView view =
        CollectionViewSource.GetDefaultView(listView1.ItemsSource);

```

Eine neue Sortierfolge festlegen:

```

    view.SortDescriptions.Clear();
    view.SortDescriptions.Add(
        new SortDescription(spalte.Content.ToString(),
            ListSortDirection.Ascending));

```

Und aktualisieren:

```

    view.Refresh();
}

```

Auf weitere Experimente mit der *ListView* verzichten wir an dieser Stelle, mit dem *DataGrid* steht uns ein wesentlich mächtigeres Control zur Verfügung. Mehr dazu in Abschnitt 11.7. Wie Sie auch größere Collections bändigen, zeigt das Praxisbeispiel in Abschnitt 11.8.

■ 11.4 Noch einmal zurück zu den Details

Nachdem wir in den bisherigen Abschnitten schon mehrfach vorgeifen mussten, wollen wir an dieser Stelle noch einmal kurz auf einige Details der Datenbindung eingehen.

Interessant für den Datenbankprogrammierer ist vor allem eine Zwischenschicht, die vom WPF quasi zwischen die Daten (Collections) und die reinen Anzeige-Controls (z. B. *ListView*) geschoben wird, um einige datenbanktypische Operationen zu ermöglichen:

- Verwaltung des aktuellen Satzzeigers
- Navigation zwischen den Datensätzen
- Sortierfunktion
- Filterfunktion

Die Rede ist von der Klasse *CollectionView*, um deren Erzeugung Sie sich nicht selbst kümmern müssen, da Sie diese automatisch erstellte View recht einfach abrufen können.

Beispiel 11.25: Abrufen der *CollectionView*

```
C#
...
    NwDataContext db = new NwDataContext();
    ICollectionView view;
...
    private void Window_Loaded(object sender, RoutedEventArgs e)
    {
        lvOrder.DataContext = db.Orders;
        view = CollectionViewSource.GetDefaultView(lvOrder.DataContext);
    }
```

Mit dieser *CollectionView* stellt es jetzt kein Problem mehr dar, die oben gewünschten Datenbankfunktionen zu implementieren.

11.4.1 Navigieren in den Daten

Wie schon in den vorhergehenden Abschnitten gezeigt, ist eine der Hauptaufgaben der *CollectionView* die Verwaltung des „Satzzeigers“. Dazu steht Ihnen zunächst die Eigenschaft *CurrentItem* zur Verfügung, die das aktuell ausgewählte Element der gebundenen Collection zurückgibt.

Weitere interessante Eigenschaften:

Eigenschaft	Beschreibung
<i>CurrentItem</i>	Aktuelles Element der Auflistung.
<i>CurrentPosition</i>	Ordinalposition des aktuellen Elements in der Auflistung.
<i>IsCurrentAfterLast</i>	Befindet sich der „Satzzeiger“ hinter dem Ende der Auflistung?
<i>IsCurrentBeforeFirst</i>	Befindet sich der „Satzzeiger“ vor dem Beginn der Auflistung?

Die eigentliche Navigation realisieren Sie mit den folgenden Methoden:

Methode	Beschreibung
<i>MoveCurrentTo</i>	Das übergebene Element wird als <i>CurrentItem</i> festgelegt.
<i>MoveCurrentToFirst</i>	„Satzzeiger“ auf das erste Element verschieben.
<i>MoveCurrentToLast</i>	„Satzzeiger“ auf das letzte Element verschieben.
<i>MoveCurrentToNext</i>	„Satzzeiger“ auf das folgende Element verschieben.
<i>MoveCurrentToPosition</i>	„Satzzeiger“ auf den angegebenen Index verschieben.
<i>MoveCurrentToPrevious</i>	„Satzzeiger“ auf das vorhergehende Element verschieben.

Beispiel 11.26: Navigationstasten für „Vor“ und „Zurück“

C#

Der folgende Aufwand ist nötig, um nicht hinter bzw. vor dem letzten bzw. ersten Datensatz zu landen:

```
private void ButtonNext_Click(object sender, RoutedEventArgs e)
{
    view.MoveCurrentToNext();
    if (view.IsCurrentAfterLast)
    {
        view.MoveCurrentToLast();
    }
}

private void ButtonPrevious_Click(object sender, RoutedEventArgs e)
{
    view.MoveCurrentToPrevious();
    if (view.IsCurrentBeforeFirst)
    {
        view.MoveCurrentToFirst();
    }
}
```

Beispiel 11.27: Verwendung von *CurrentItem*

C#

Löschen eines Listeneintrags per *CurrentItem* und Typisierung:

```
private void ButtonLoeschen_Click(object sender, RoutedEventArgs e)
{
    Abteilung.Remove(view.CurrentItem as Person);
}
```

11.4.2 Sortieren

Dass sich die *CollectionView* auch zum Sortieren eignet, haben wir ja bereits am Beispiel der *ListView* gezeigt, wo durch Klicken auf den Spaltenkopf die Collection nach der jeweiligen Spalte sortiert wurde.

Zum Einsatz kommt die Collection *SortDescriptions*, die neben den Membernamen auch die Sortierfolge enthält. Da es sich um eine Collection handelt, können Sie auch mehrere Elemente angeben:

Beispiel 11.28: Sortieren einer Collection

C#

```
private void SortClick(object sender, RoutedEventArgs e)
{
    GridViewColumnHeader spalte = sender as GridViewColumnHeader;

    CollectionView abrufen:

    ICollectionView view =
        CollectionViewSource.GetDefaultView(listView1.ItemsSource);

    Bisherige Sortiervorgaben löschen:

    view.SortDescriptions.Clear();

    Eine neue Sortierfolge (Spaltenname, aufsteigend) festlegen:

    view.SortDescriptions.Add(new SortDescription(
        spalte.Content.ToString(),
        ListSortDirection.Ascending));

    Ansicht aktualisieren:

    view.Refresh();
}
```

11.4.3 Filtern

Auch wenn Sie mit dieser Variante vorsichtig sein sollten (Daten werden eigentlich vor der Anzeige gefiltert, um unnötigen Traffic zu vermeiden), so besteht doch die Möglichkeit, zur Laufzeit gezielt Daten aus der gebundenen Collection herauszufiltern. Nutzen Sie dazu die *Filter*-Eigenschaft, der Sie eine selbst zu definierende Methode zuweisen.

Beispiel 11.29: Filter festlegen

C#

Zunächst unsere Filterfunktion (alle Vornamen die mit „R“ beginnen):

```
protected bool FilterVorname(object value)
{
    Person p = value as Person;
    return p.Vorname.StartsWith("R");
}
```

Und hier wird der Filter zugewiesen (ein Aktualisieren ist nicht nötig):

```
private void ButtonFilter_Click(object sender, RoutedEventArgs e)
{
    ICollectionView view =
        CollectionViewSource.GetDefaultView(listBox1.ItemsSource);
    view.Filter += FilterVorname;
}
```



HINWEIS: Möchten Sie den Filter wieder löschen, weisen Sie der Eigenschaft einfach *null* zu.

11.4.4 Live Shaping

Die vorhergehenden Funktionen zum Sortieren, Filtern und auch Gruppieren funktionieren recht gut, auch wenn Sie zum Beispiel neue Einträge zur Collection hinzufügen. Alternativ können Sie auch die *Refresh*-Methode der *CollectionView* aufrufen.

Doch was ist, wenn Sie lediglich ein Objekt der Collection bearbeiten und sich so z.B. die Filterbedingung für dieses Objekt ändert? In diesem Fall werden Sie schnell feststellen, dass Anzeige und Inhalt der Collection nicht mehr übereinstimmt.

Beispiel 11.30: (Fortsetzung) Fehlende Aktualisierung

C#

...

Filtern Sie die Daten und rufen Sie folgende Methode auf, passiert nichts:

```
private void ButtonFilterChange_Click(
    object sender, RoutedEventArgs e)
{
    Person person = Abteilung.FirstOrDefault(
        p => p.Vorname == "Rudi");
    p.Vorname = "Rudolf";
}
```

Erst nach einem Refresh sind die gefilterten Daten auch aktuell:

```
//ICollectionView view = CollectionViewSource.DefaultView(
    listBox1.ItemsSource);
//view.Refresh();
}
...
```

Hier hilft Ihnen Live Shaping weiter. Über das Interface *ICollectionViewLiveShaping* können Sie bestimmte Spalten zur Überwachung anmelden.

Jeweils drei neue Member sind für die weitere Arbeit interessant:

- Die Eigenschaft *CanChangeLiveFiltering* (... *Sorting*, ... *Grouping*) bestimmt, ob die Überwachung möglich ist.
- Die Eigenschaft *IsLiveFiltering* (... *Sorting*, ... *Grouping*) bestimmt, ob die Überwachung eingeschaltet ist.
- Die Collection *LiveFilteringProperties* (... *Sorting*..., ... *Grouping*...) enthält die Namen der zu überwachenden Eigenschaften.

Beispiel 11.31: Filtern mit Live Shaping

C#

```
...
private void ButtonLiveShaping_Click(
    object sender, RoutedEventArgs e)
{
```

Filter wie bekannt festlegen:

```
view.Filter += FilterVorname;
```

Wir rufen das neue Interface ab:

```
ICollectionViewLiveShaping viewls =
    view as ICollectionViewLiveShaping;
```

Test auf Interface:

```
if (viewls == null)
{
    MessageBox.Show("Nicht unterstützt!");
}
```

Überprüfen ob eine Überwachung möglich ist?

```
if (viewls.CanChangeLiveFiltering)
{
```

Feld *Vorname* soll überwacht werden:

```
views.LiveFilteringProperties.Add(nameof(Person.Vorname));
views.IsLiveFiltering = true;
}
...
}
```



HINWEIS: Da diese Form der Überwachung recht ressourcenintensiv ist, sollten Sie davon nur Gebrauch machen, wenn es unbedingt nötig ist.

■ 11.5 Anzeige von Datenbankinhalten

Anzeige eigener Collections gut und schön, aber wir wollen auch noch kurz einen Blick aufs große Ganze werfen und damit sind wir schon bei der „Königsdisziplin“, den Datenbanken, angelangt.



HINWEIS: Wer jetzt Berge von ADO.NET-Quellcode erwartet, den werden wir enttäuschen. Für den Zugriff auf unsere *Northwind*-Beispieldatenbank werden wir das Entity Framework verwenden.



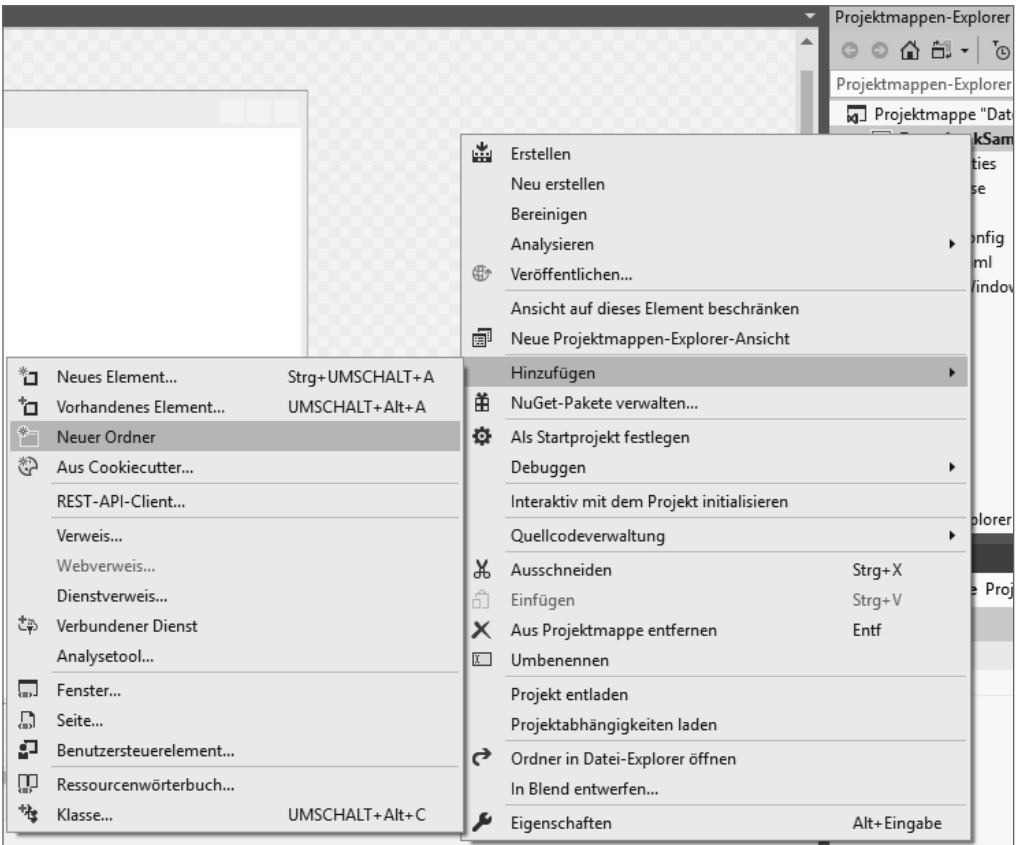
HINWEIS: Wenn Sie bislang noch nicht mit Datenbanken gearbeitet haben, sollten Sie das Kapitel 18 durcharbeiten, bevor Sie mit diesem Abschnitt fortfahren.

11.5.1 Datenmodell per EF-Designer erzeugen

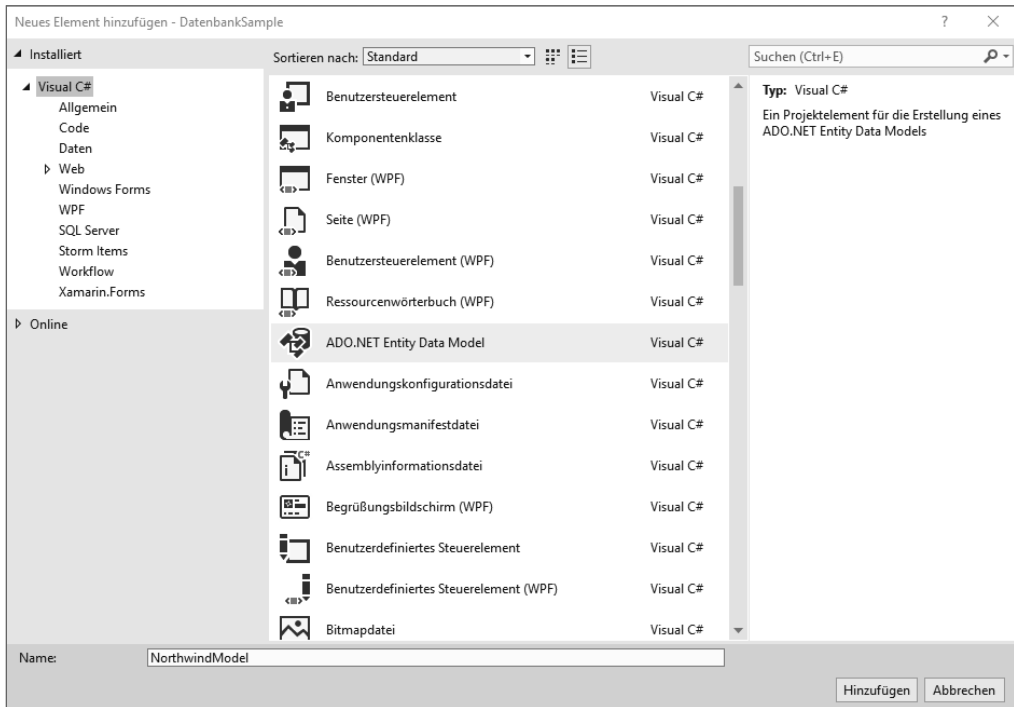
Fügen Sie Ihrem WPF-Projekt zuerst einen neuen Ordner hinzu und nennen diesen *Model*.



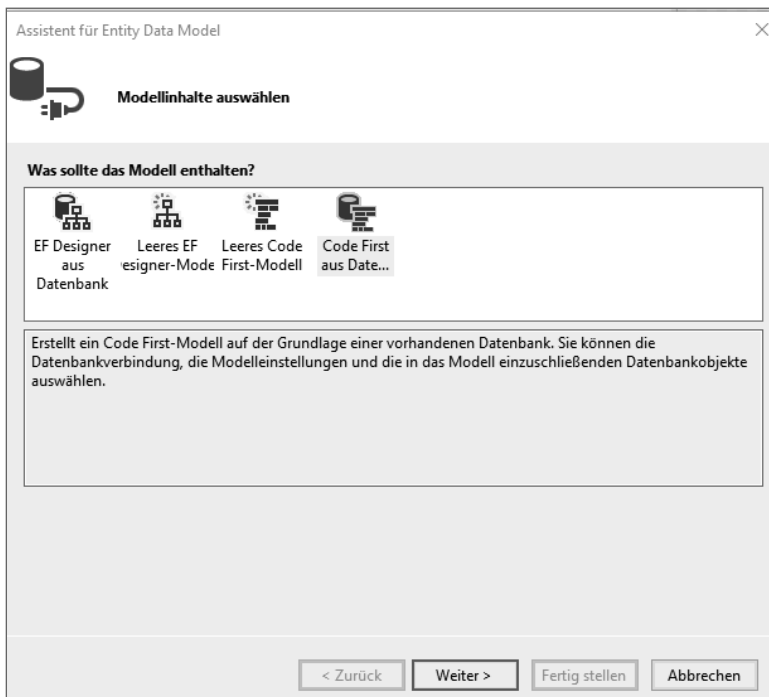
HINWEIS: Sollten Sie die *Northwind*-Beispieldatenbank nicht installiert haben, können Sie unter <https://github.com/Microsoft/sql-server-samples/tree/master/samples/databases/northwind-pubs> das Installationskript herunterladen.



Dann fügen Sie im Ordner *Model* eine neue Klasse hinzu und wählen als Vorlage *ADO.NET Entity Data Model*. Vergeben Sie *NorthwindModel* als Name.



Im nächsten Dialog wählen Sie dann den Eintrag *CodeFirst aus Datenbank*.



Im nächsten Schritt wählen Sie die Verbindungseigenschaften zu Ihrer *Northwind*-Datenbank aus. Bei mir läuft die Datenbank in einer SQL Server Express-Instanz. Beim nächsten Dialog übernehmen Sie dann die vorgeschlagenen Einstellungen. Ändern Sie lediglich den Namen für die Verbindungszeichenfolge auf *Northwind_Model*, um eine Namenseindeutigkeit sicherzustellen. Die Verbindungszeichenfolge wird dann in der Konfigurationsdatei *app.config* der Applikation gespeichert.

Verbindungseigenschaften

Geben Sie Informationen zum Verbinden mit der ausgewählten Datenquelle ein, oder klicken Sie auf "Ändern", um eine andere Datenquelle und/oder einen anderen Anbieter auszuwählen.

Datenquelle: Microsoft SQL Server (SqlClient) Ändern...

Servername: .\SqlExpress Aktualisieren

Beim Server anmelden

Authentifizierung: Windows-Authentifizierung

Benutzername:

Kennwort:

Kennwort speichern

Mit Datenbank verbinden

Datenbanknamen auswählen oder eingeben: Northwind

Datenbankdatei anhängen: Durchsuchen...

Logischer Name:

Erweitert...

Testverbindung OK Abbrechen

Wählen Sie dann die drei Tabellen *Orders*, *Order_Details* und *Products* aus und machen Sie ein Häkchen bei der CheckBox *Generierte Objektnamen in den Singular oder Plural*. Dies bewirkt, dass die erzeugten Objektnamen im Singular (also ohne endendes „-s“) benannt werden.

Der Designer erstellt nachfolgend automatisch die erforderlichen C#-Entitätsklassen für die einzelnen Tabellen sowie deren Beziehungen. Damit sind wir aber auch schon wieder bei den schon bekannten Collections angekommen, die weitere Vorgehensweise dürfte Ihnen also bekannt vorkommen.



HINWEIS: Sie können neben reinen Tabellen auch Views dem Model hinzufügen. Views werden dabei wie Tabellen behandelt.

11.5.2 Die Programmoberfläche

Nach Schließen des Designers wollen wir uns mit einem einfachen WPF-Projekt von der Funktionsfähigkeit überzeugen.

```
<Window x:Class="DatenbankSample.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:DatenbankSample"
        mc:Ignorable="d">
```

Eine Ereignisprozedur beim Öffnen des Window:

```
Title="MainWindow" Height="450" Width="800" Loaded="Window_Loaded">
```

Zweispaltiges Layout per *Grid*:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="75" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
```

Hier die *ListView* mit den vorhandenen Bestellungen (Tabelle *Order*):

```
<ListView Grid.Column="0" Name="lvOrder" IsSynchronizedWithCurrentItem="True"
          ItemsSource="{Binding}" SelectionChanged="lvOrder_SelectionChanged"
          HorizontalAlignment="Left" >
```

Die eigentlich Bindung erfolgt per *DataContext*-Zuweisung im C#-Code. Über das *SelectionChanged*-Ereignis werden wir die Detaildaten in die zweite *ListView* „zaubern“.

```
<ListView.View>
  <GridView>
    <GridView.Columns>
```

Wir zeigen nur die Spalte mit der Bestellnummer an:

```
    <GridViewColumn Header="OrderID"
                    DisplayMemberBinding="{Binding OrderID}" />
  </GridView.Columns>
</GridView>
</ListView.View>
</ListView>
```

Die *ListView* für die Detaildaten (die *DataContext*-Eigenschaft setzen wir im *Selection-Changed*-Ereignis der obigen *ListView*):

```
<ListView Grid.Column="1" Name="lvOrderDetails"
    IsSynchronizedWithCurrentItem="True" ItemsSource="{Binding}" >
  <ListView.View>
    <GridView>
      <GridView.Columns>
        <GridViewColumn Header="OrderId"
          DisplayMemberBinding="{Binding OrderID}" />
        <GridViewColumn Header="ID"
          DisplayMemberBinding="{Binding ProductID}" />
```

Wer jetzt erwartet, dass wir uns jetzt noch die Mühe machen und noch eine dritte *GridView* für die Artikelnamen einbinden, hat nicht mit der Leistungsfähigkeit von LINQ und dem Entity Framework gerechnet. Es genügt die Abfrage der untergeordneten Collection *Product*:

```
<GridViewColumn Header="Artikelname"
    DisplayMemberBinding="{Binding Product.ProductName}" />
```

Ein sinnvoller Vorteil von objektrelationalen Mapper-Klassen!

```
</GridView.Columns>
</GridView>
</ListView.View>
</ListView>
</Grid>
</Window>
```

11.5.3 Der Zugriff auf die Daten

Jetzt müssen wir noch den erforderlichen C#-Code erstellen, um die Daten auch aus der Datenbank abzurufen.

```
...
public partial class MainWindow : Window
{
```

Die meiste Arbeit nimmt uns der *Entity Framework DataContext* ab, den wir gleich zu Beginn instanziiieren:

```
NorthwindModel context = new NorthwindModel();
```

Da wird das *Model* in einem eigenen Ordner *Model* gespeichert haben, benötigen wir noch einen Verweis auf den Namespace *DatenbankSample.Model*.

Wir wollen auch die Default-View zwischenspeichern, da wir diese für das Abrufen der Detaildatensätze benötigen:

```
ICollectionView view;
...
```


Auch hierfür benötigen wir wieder den Namespace *System.ComponentModel*.

Beim Laden des Fensters:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
```

Dem *DataContext* der linken *ListView* wird die Tabelle *Orders* zugewiesen:

```
lvOrder.DataContext = context.Orders.ToList();
```

Die Default-View abrufen:

```
    view = CollectionViewSource.GetDefaultView(lvOrder.DataContext);
}
```

So, das wäre schon alles, wenn da nicht noch die Detaildatenanzeige fehlen würde. Im *SelectionChanged*-Ereignis der linken *ListView* kümmern wir uns zunächst um das Abrufen des aktuellen Datensatzes und leiten aus diesem Objekt die *OrderDetails* ab (als Collection enthalten):

```
private void lvOrder_SelectionChanged(object sender,
                                     SelectionChangedEventArgs e)
{
    lvOrderDetails.DataContext = (view.CurrentItem as Order).Order_Details;
}
}
```

OrderID	Orderid	ID	Artikelname
10248	11		Queso Cabrales
10249	42		Singaporean Hokkien Fried Mee
10250	72		Mozzarella di Giovanni
10251			
10252			
10253			
10254			
10255			
10256			
10257			
10258			
10259			
10260			
10261			
10262			
10263			
10264			
10265			
10266			

Ach ja, wie kommen eigentlich neue Datensätze in die Tabellen? Hier genügt es, wenn Sie z. B. ein neues *Product*-Objekt erstellen und es an die *Products*-Collection anhängen. Mit einem *SaveChanges*-Aufruf des *Entity-Framework-DataContext*-Objekts ist die Änderung dann auch schon zum Server übertragen.



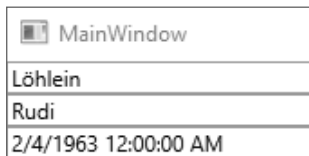
HINWEIS: In Kapitel 18 werden wir die Datenbankkonzepte noch mal ausführlich betrachten.

11.6 Formatieren von Werten

In unseren Beispielen haben wir uns bislang erfolgreich davor gedrückt, Datumswerte, Währungen etc. in einem sinnvollen Format anzuzeigen bzw. zu formatieren.

Wir erweitern unsere Personenklasse aus den vorigen Beispielen um eine Eigenschaft *Geburtstag* vom Datentyp *DateTime*.

Binden Sie beispielsweise einen Datumswert an eine *TextBox*, wird zunächst das Standardformat angezeigt:



Das sieht aus deutscher Sicht zunächst wenig erfreulich aus, aber mit dem *Language*-Attribut können Sie hier etwas nachhelfen.

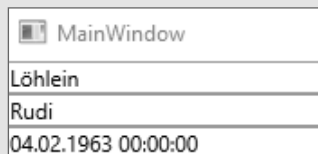
Beispiel 11.32: Verwendung *Language*-Attribut

XAML

```
<TextBox Text="{Binding Path=Geburtstag}" Language="de"/>
```

Ergebnis

Nachfolgend sollte zumindest ein deutscher Datumswert angezeigt werden:



Doch auch dies ist noch nicht der Weisheit letzter Schluss.

11.6.1 IValueConverter

Mithilfe der WPF-Wertkonvertierer können Sie jede beliebige Konvertierung zwischen Quelle und Ziel einer Datenbindung realisieren. Dazu erstellen Sie eine Klasse, die das *IValueConverter*-Interface unterstützt. Diese Klasse muss zwei Methoden implementieren:

- *Convert* (von der Quelle zum Ziel)
- *ConvertBack* (vom Ziel zur Quelle)

Sicher können Sie sich denken, dass die *ConvertBack*-Methode den höheren Programmieraufwand erfordert, hat doch hier der User die Möglichkeit, zunächst beliebige Werte in die Textfelder einzugeben, die Sie dann mühsam in den geforderten Datentyp umwandeln müssen.

Beispiel 11.33: Implementieren und Verwenden eines Wert-Konvertierers

C#

An dieser Stelle wollen wir allerdings nicht das Rad neu erfinden, sondern ein Beispiel aus dem Microsoft MSDN darstellen.

Erstellen Sie dazu eine neue Klasse *DateConverter*:

```
using System.Globalization;
using System.Windows.Data;
```

```
namespace Formatierungen
{
```

Hier die neue Klasse *DateConverter*, die Sie mit entsprechenden Attributen versehen sollten:

```
    [ValueConversion(typeof(DateTime), typeof(String))]
    public class DateConverter : IValueConverter
    {
```

Konvertieren von der Quelle zum Ziel (übergeben werden die Quelleigenschaft, der Zieleigenschaft-Typ, ein Konverter-Parameter sowie die aktuellen Landeseinstellungen):

```
        public object Convert(object value, Type targetType,
                               object parameter, CultureInfo culture)
        {
            DateTime date = (DateTime)value;
            return date.ToShortDateString();
        }
```

Konvertieren vom Ziel (z. B. *TextBox*) zur Quelle (z. B. Objekt):

```
        public object ConvertBack(object value, Type targetType,
                                   object parameter, CultureInfo culture)
        {
            string strValue = value.ToString();
            DateTime resultDateTime;
            if (DateTime.TryParse(strValue, out resultDateTime))
            {
                return resultDateTime;
            }
        }
```

```

    }
    return value;
}
}

```

XAML

Die Verwendung im XAML-Code:

```

<Window x:Class="Formatierungen.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

```

Zunächst den lokalen Namespace einbinden:

```

    xmlns:local="clr-namespace:Formatierungen"
    mc:Ignorable="d"
    Title="MainWindow" Height="450" Width="800" Loaded="Window_Loaded">

```

Die Einbindung der Klasse erfolgt per Ressource:

```

<Window.Resources>
    <local:DateConverter x:Key="dateConverter"/>
</Window.Resources>
<StackPanel>
    <TextBox Text="{Binding Path=Nachname}" />
    <TextBox Text="{Binding Path=Vorname}" />

```

Sollten Sie nach diesen Zeilen eine Fehlermeldung bekommen, dass der Typ *DateConverter* nicht im Namespace *Formatierungen* existiert, dann kompilieren Sie die Anwendung noch mal neu (F5).

Und hier verwenden wir den Konverter bei der Bindung:

```

    <TextBox Text="{Binding Path=Geburtstag,
        Converter={StaticResource dateConverter}}" />
</StackPanel>
</Window>

```

Ergebnis

Das neue Ergebnis sieht schon viel ansprechender aus:

 MainWindow
Löhlein
Rudi
04.02.1963



HINWEIS: Zusätzlich besteht die Möglichkeit, vorhandene Wertkonvertierer per Eigenschafteneditor (siehe folgende Abbildung) zuzuweisen. Der Eigenschafteneditor erstellt, falls erforderlich, die entsprechenden Einträge im *<Window.Resources>*-Abschnitt des Formulars und weist das Attribut *Converter* zu.

11.6.2 BindingBase.StringFormat-Eigenschaft

Nachdem Sie sich durch unser obiges Beispiel gequält haben, wollen wir Ihnen auch nicht die dritte Variante zur Formatierung von Werten vorenthalten.

Werfen Sie doch einmal einen Blick auf die *BindingBase.StringFormat*-Eigenschaft, welche die Verwendung eines *IValueConverters* in vielen Standardfällen überflüssig macht.

Beispiel 11.34: Zwei verschiedene Datumsformate zuweisen

XAML

```
...  
<TextBox Text="{Binding Path=Geburtstag, StringFormat= d. MMM yyyy}" />  
...  
<TextBox Text="{Binding Path=Geburtstag, StringFormat= dd.MM.yyyy }" />  
...
```

Ergebnis

Das Ergebnis:

4. Feb 1963

04.02.1963

11.7 Das DataGrid als Universalwerkzeug

Seit der WPF-Version 4 wird auch ein *DataGrid* regulär unterstützt, ohne zusätzliche Tool-kits etc. laden zu müssen. Wie die schon besprochene *ListView* erlaubt auch das *DataGrid* die Anzeige von Collections im Tabellenformat. Zusätzlich werden Funktionen zum Editieren, Löschen, Auswählen und Sortieren angeboten.



HINWEIS: Anhand einiger Fallbeispiele wollen wir Ihnen eine Übersicht des Funktionsumfangs geben. Für eine komplette Beschreibung aller Eigenschaften bzw. Möglichkeiten fehlt hier jedoch der Platz und wir verweisen auf die recht umfangreiche Hilfe zum *DataGrid*-Control.

11.7.1 Grundlagen der Anzeige

Wie fast nicht anders zu erwarten, erfolgt die Anbindung an die Datenquelle mittels *ItemsSource*-Eigenschaft. Wir erzählen Ihnen an dieser Stelle also nichts Neues und verweisen auf die vorhergehenden Abschnitte.

Im Unterschied zu den bereits beschriebenen Controls bietet uns das *DataGrid* einen wesentlichen Vorteil: Sie brauchen sich nicht um das Erstellen der einzelnen Spalten zu kümmern, dank *AutoGenerateColumns*-Eigenschaft werden automatisch die erforderlichen Spalten erzeugt.

Erzeugen Sie, wie in Abschnitt 11.5.1 bereits erläutert, noch mal das *NorthwindModel* in einem Unterordner *Model*.

Beispiel 11.35: Anbinden des *DataGrids* an die Entity-Framework-Daten

XAML

```
<DataGrid Name="dataGrid1" />
```

C#

```
public partial class MainWindow : Window
{
    NorthwindModel context = new NorthwindModel();
    ...

    private void Window_Loaded(object sender, RoutedEventArgs e)
    {
        dataGrid1.ItemsSource = context.Products.ToList();
    }
}
```

Ergebnis

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder	ReorderLevel	Discontinued
1	Chai	1	1	10 boxes x 20 bags	27.0000	40	0	10	<input type="checkbox"/>
2	Chang	1	1	24 - 12 oz bottles	19.0000	17	40	25	<input type="checkbox"/>
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10.0000	13	70	25	<input type="checkbox"/>
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22.0000	53	0	0	<input type="checkbox"/>
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.3500	0	0	0	<input checked="" type="checkbox"/>
6	Grandma's Boysenberry Spread	3	2	12 - 8 oz jars	25.0000	120	0	25	<input type="checkbox"/>
7	Uncle Bob's Organic Dried Pears	3	7	12 - 1 lb pkgs.	30.0000	15	0	10	<input type="checkbox"/>
8	Northwoods Cranberry Sauce	3	2	12 - 12 oz jars	40.0000	6	0	0	<input type="checkbox"/>
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97.0000	29	0	0	<input checked="" type="checkbox"/>
10	Ikura	4	8	12 - 200 ml jars	31.0000	31	0	0	<input type="checkbox"/>
11	Queso Cabrales	5	4	1 kg pkg.	21.0000	22	30	30	<input type="checkbox"/>
12	Queso Manchego La Pastora	5	4	10 - 500 g pkgs.	38.0000	86	0	0	<input type="checkbox"/>
13	Konbu	6	8	2 kg box	6.0000	24	0	5	<input type="checkbox"/>
14	Tofu	6	7	40 - 100 g pkgs.	23.2500	35	0	0	<input type="checkbox"/>
15	Genen Shouyu	6	2	24 - 250 ml bottles	15.5000	39	0	5	<input type="checkbox"/>
16	Pavlova	7	3	32 - 500 g boxes	17.4500	29	0	10	<input type="checkbox"/>
17	Alice Mutton	7	6	20 - 1 kg tins	39.0000	0	0	10	<input checked="" type="checkbox"/>
18	Carnarvon Tigers	7	8	16 kg pkg.	62.5000	42	0	0	<input type="checkbox"/>
19	Teatime Chocolate Biscuits	8	3	10 boxes x 12 pieces	9.2000	25	0	5	<input type="checkbox"/>

Die Verwendung der *AutoGenerateColumns*-Eigenschaft ist sicher recht praktisch, doch haben Sie in diesem Fall keinen Einfluss auf Anzahl, Reihenfolge und Aussehen der Spalten.

Das *DataGrid* selbst unterscheidet in diesem Fall lediglich zwischen Spalten der Typen *DataGridTextColumn* und *DataGridCheckBoxColumn*, deren Bedeutung sich bereits durch den Namen erklärt.

11.7.2 UI-Virtualisierung

Sicher interessiert es Sie auch, wie leistungsfähig das *DataGrid* ist. Erstellen Sie ruhig einmal eine Collection mit 1 000 000 Datensätzen und weisen Sie diese als *ItemsSource* zu. Sie werden feststellen, dass das Erzeugen der Collection wesentlich länger dauert als die Anzeige der Daten. Der Grund für dieses Verhalten basiert auf der UI-Virtualisierung, die mithilfe eines *VirtualizingStackPanel* als Layoutpanel innerhalb des *DataGrid* (auch *ListView*, *ListBox* etc.) verwendet wird.

Das *VirtualizingStackPanel* sorgt dafür, dass nur die gerade sichtbaren Einträge (bzw. die dazu notwendigen Controls) erzeugt werden. Was passiert, wenn dies nicht so ist, können Sie ganz einfach ausprobieren. Es genügt, wenn Sie das folgende Attribut in die Elementdefinition einfügen:

```
<DataGrid VirtualizingStackPanel.IsVirtualizing="False" Name="dataGrid1" />
```

Bitte besorgen Sie sich rechtzeitig eine Zeitung und eine Kanne Kaffee, wenn Sie versuchen wollen, eine große Collection an das *DataGrid* zu binden. Im extremsten Fall kommt es zur Meldung, dass der verfügbare Arbeitsspeicher nicht ausreicht. Die Ursache dürfte schnell klar werden, wenn Sie sich vorstellen, dass für jede erforderliche Zeile und alle angezeigten Spalten die entsprechenden Anzeige-Controls generiert werden müssen. Selbst bei unserer kleinen Datenbank wird man schon einen zeitlichen Unterschied feststellen.

11.7.3 Spalten selbst definieren

Gehen Ihnen die Möglichkeiten von *AutoGenerateColumns* nicht weit genug, können Sie alternativ auch selbst Hand anlegen und die einzelnen Spalten frei definieren. Setzen Sie in diesem Fall das Attribut *AutoGenerateColumns* auf *false* und fügen Sie die Spaltendefinitionen der *Columns*-Eigenschaft hinzu (die Reihenfolge der Definition entscheidet über die Anzeigereihenfolge).

Wir machen es uns im folgenden Beispiel etwas einfacher und definieren nur vier Spalten. Der Rest, denke ich, ist dann klar, wie es geht.

Beispiel 11.36: *DataGrid* mit einzeln definierten Spalten

XAML

```
<DataGrid AutoGenerateColumns="False" EnableRowVirtualization="True"
  ItemsSource="{Binding}" Name="dataGrid1" >
```

Und hier folgen die Definitionen der einzelnen Spalten:

```
<DataGrid.Columns>
```

Eine Textspalte erzeugen, Bindung an den Member *ProductID* (bitte auf komplette Groß-/Kleinschreibung achten) herstellen, die Kopfzeile mit „ID“ beschriften und eine Größenanpassung vornehmen:

```
<DataGridTextColumn Binding="{Binding Path=ProductID}"
  Header="ID" Width="SizeToHeader" />
```

Gleiches für den Produktnamen (ohne Größenanpassung):

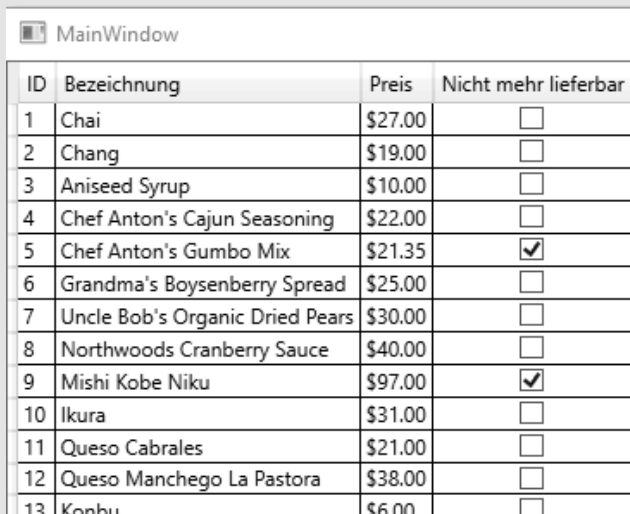
```
<DataGridTextBoxColumn Binding="{Binding Path=ProductName}"
    Header="Bezeichnung" />
```

Dann legen wir eine Spalte für den *UnitPrice* an. Diese Spalte wollen wir auch als Währung formatieren (*StringFormat=C*). Und zu guter Letzt die Spalte *Discontinued*, weil wir hier eine *DataGridCheckBox* verwenden wollen:

```
<DataGridTextBoxColumn Binding="{Binding UnitPrice, StringFormat=C}"
    Header="Preis" />
<DataGridCheckBoxColumn Binding="{Binding Discontinued}"
    Header="Nicht mehr lieferbar" />
</DataGrid.Columns>
</DataGrid>
</StackPanel>
```

Ergebnis

Das erzeugte *DataGrid*:



ID	Bezeichnung	Preis	Nicht mehr lieferbar
1	Chai	\$27.00	<input type="checkbox"/>
2	Chang	\$19.00	<input type="checkbox"/>
3	Aniseed Syrup	\$10.00	<input type="checkbox"/>
4	Chef Anton's Cajun Seasoning	\$22.00	<input type="checkbox"/>
5	Chef Anton's Gumbo Mix	\$21.35	<input checked="" type="checkbox"/>
6	Grandma's Boysenberry Spread	\$25.00	<input type="checkbox"/>
7	Uncle Bob's Organic Dried Pears	\$30.00	<input type="checkbox"/>
8	Northwoods Cranberry Sauce	\$40.00	<input type="checkbox"/>
9	Mishi Kobe Niku	\$97.00	<input checked="" type="checkbox"/>
10	Ikura	\$31.00	<input type="checkbox"/>
11	Queso Cabrales	\$21.00	<input type="checkbox"/>
12	Queso Manchego La Pastora	\$38.00	<input type="checkbox"/>
13	Konbu	\$6.00	<input type="checkbox"/>

Anmerkung

Im obigen Beispiel ist der Preis in Dollar angegeben. Wenn Sie die aktuelle Währung aus der Systemeinstellung wählen wollen, müssen Sie noch folgenden Code hinzufügen. Am besten machen Sie das in der überschriebenen *OnStart*-Methode der *App*, denn dann gilt das für die gesamte Anwendung:

```
public partial class App : Application
{
    protected override void OnStartup(StartupEventArgs e)
    {
        base.OnStartup(e);
        FrameworkElement.LanguageProperty.OverrideMetadata(
            typeof(FrameworkElement),
            new FrameworkPropertyMetadata(
```



```

        XmlLanguage.GetLanguage(
            CultureInfo.CurrentCulture.IetfLanguageTag));
    }
}

```



HINWEIS: Sie können die Spaltenbreite auch mit „*“ angeben, in diesem Fall verwendet die Spalte den restlichen verfügbaren Platz.

Außer den Standard-Spaltentypen

- *DataGridTextColumn*,
- *DataGridCheckBoxColumn*,
- *DataGridComboBoxColumn*,
- *DataGridHyperlinkColumn*

steht auch die recht flexible *DataGridTemplateColumn* zur Verfügung. Welche Controls Sie hier einbinden (*Image*, *Chart*, *RichTextBox* etc.), bleibt Ihrer Fantasie überlassen. Auf diese Art und Weise kann man dann auch den Preis zum Beispiel rechtsbündig darstellen.

Weitere Gestaltungsmöglichkeiten bieten sich mit dem Ein- und Ausblenden der Trennlinien, der Konfiguration der Spaltenköpfe per Template usw.

11.7.4 Zusatzinformationen in den Zeilen anzeigen

Nicht alle Informationen sollen immer gleich in einem Grid sichtbar sein, vielfach werden Detailfenster etc. eingeblendet, um nach der Auswahl eines Datensatzes weitere Informationen anzuzeigen. An dieser Stelle bietet das *DataGrid* mit dem *RowDetailsTemplate* ein recht interessantes Feature, versetzt Sie dieses Template doch in die Lage, unter bestimmten Umständen (*RowDetailsVisibilityMode*-Eigenschaft) zusätzliche Inhalte einzublenden.

Beispiel 11.37: Verwendung von *RowDetailsTemplate*

XAML

Zunächst müssen Sie bestimmen, wann die Details eingeblendet werden sollen:

```

<DataGrid AutoGenerateColumns="False" EnableRowVirtualization="True"
    ItemsSource="{Binding}"
    RowDetailsVisibilityMode="VisibleWhenSelected" >
  <DataGrid.Columns>
  ...
  </DataGrid.Columns>

```

Nach der Spaltendefinition können Sie das *RowDetailsTemplate* einfügen und mit den gewünschten Informationen füllen. Wir wollen hier alle Bestelldetails mit diesem Produkt anzeigen:

```

<DataGrid.RowDetailsTemplate>
  <DataTemplate>

```

```

<DataGrid ItemsSource="{Binding Order_Details}"
          AutoGenerateColumns="False">
  <DataGrid.Columns>
    <DataGridTextColumn Binding="{Binding OrderID}" Header="OrderID"/>
    <DataGridTextColumn Binding="{Binding Quantity}" Header="Anzahl" />
    <DataGridTextColumn Binding="{Binding Discount}" Header="Rabatt" />
  </DataGrid.Columns>
</DataGrid>
</DataTemplate>
</DataGrid.RowDetailsTemplate>
</DataGrid>

```

Ergebnis

ID	Bezeichnung	Preis	Nicht mehr lieferbar
1	Chai	27,00 €	<input type="checkbox"/>
2	Chang	19,00 €	<input type="checkbox"/>
3	Aniseed Syrup	10,00 €	<input type="checkbox"/>
4	Chef Anton's Cajun Seasoning	22,00 €	<input type="checkbox"/>
5	Chef Anton's Gumbo Mix	21,35 €	<input checked="" type="checkbox"/>

OrderID	Anzahl	Rabatt
10258	65	0,2
10262	12	0,2
10290	20	0
10382	32	0
10635	15	0,1
10708	4	0
10848	30	0
10958	20	0
11030	70	0
11047	30	0,25

6	Grandma's Boysenberry Spread	25,00 €	<input type="checkbox"/>
7	Uncle Bob's Organic Dried Pears	30,00 €	<input type="checkbox"/>
8	Northwoods Cranberry Sauce	40,00 €	<input type="checkbox"/>
9	Marie's Organic Applesauce	27,00 €	<input type="checkbox"/>

Mit *RowDetailsVisibilityMode* bestimmen Sie, wie die Zeilendetails angezeigt werden. Standardwert ist *Collapsed* (nicht sichtbar), alternativ steht *Visible* (immer sichtbar) oder *VisibleWhenSelected* zur Verfügung (nur die aktuelle Zeile).

11.7.5 Vom Betrachten zum Editieren

Auch wenn die umfangreichen Anzeigeeoptionen das *DataGrid* für diverse Aufgaben prädestinieren, eine Hauptaufgabe dürfte in den meisten Fällen auch das Editieren der Inhalte sein.

Grundsätzlich entscheidet zunächst die übergreifende Eigenschaft *IsReadOnly* über die Fähigkeit, Inhalte des *DataGrids* zu editieren oder nur zu betrachten. Gleiches gilt auch auf Spaltenebene, auch hier können Sie mit *IsReadOnly* darüber entscheiden, welche Spalten editierbar sind und welche nicht. Zusätzlich unterstützen Sie diverse Ereignisse vor, während und nach dem Editiervorgang (*BeginningEdit*, *PreparingCellForEdit*, *CellEditEnding*, ...).

■ 11.8 Praxisbeispiel – Collections in Hintergrundthreads füllen

In den vorhergehenden Beispielen haben wir es uns recht einfach gemacht. Eine Collection wurde erzeugt, gefüllt und angezeigt. So weit, so gut, aber was, wenn das Erzeugen der Collection etwas länger dauert? Ein kleines Beispielprogramm zeigt das Problem und natürlich auch die Lösung dafür. Dabei trennen wir aber zwischen der bisherigen Lösung und einer mit .NET 4.5 eingeführten Neuerung.

Oberfläche

Ein einfaches *Window* mit einigen Schaltflächen und einer *ListView* zur Anzeige der Daten:

```
<Window x:Class="PraxisbeispielDatenbindung.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:local="clr-namespace:PraxisbeispielDatenbindung"
  mc:Ignorable="d"
  Title="MainWindow" Height="450" Width="800">
  <DockPanel>
    <StackPanel DockPanel.Dock="Top" Orientation="Horizontal">
      <Button Click="ButtonVT_Click">Laden Vordergrund-Thread</Button>
      <Button Click="ButtonHT_Click">Laden Hintergrund-Thread</Button>
      <Button Click="ButtonHL_Click">Laden Hintergrund Lösung</Button>
      <Button Click="ButtonLN_Click">Laden neu</Button>
    </StackPanel>
    <ListView Name="listView1" IsSynchronizedWithCurrentItem="True"
      ItemsSource="{Binding}" VirtualizingPanel.IsVirtualizing="True"
    />
  </DockPanel>
</Window>
```

Das Problem

Stellen Sie sich folgendes Szenario vor: Sie füllen eine Liste von *Person*-Objekten³, leider dauert der Abruf jedes einzelnen Objekts etwas länger:

³ Definition siehe Abschnitt 11.2.3.

```

public partial MainWindow : Window
{
    public ObservableCollection<Person> Abteilung { get; set; }

    public MainWindow()
    {
        InitializeComponent();
    }

    private void Window_Loaded(object sender, RoutedEventArgs e)
    {
        Abteilung = new ObservableCollection<Person>();
        DataContext = Abteilung;
    }

    private void Datenabrufen()
    {
        Abteilung.Clear();
        for (int i = 0; i < 100; i++)
        {

```

Hier simulieren wir eine Zeitverzögerung, z. B. eine langsame Datenverbindung:

```

        System.Threading.Thread.Sleep(100);
        Abteilung.Add(new Person
        {
            Nachname = $"Müller{i}",
            Vorname = "Thomas"
        });
    }

    private void ButtonVT_Click(object sender, RoutedEventArgs e)
    {
        Datenabrufen();
    }

```

Starten Sie die Anwendung, werden Sie nach einem Klick auf die Schaltfläche feststellen, dass Ihre Anwendung „einfriert“. Diese Lösung wollen Sie dem Endanwender sicher nicht zumuten. Was liegt also näher, als diese Aufgabe in einen Hintergrundthread zu verlagern.



HINWEIS: Eine ausführliche Erläuterung zu Tasks und Threads folgt in den Kapiteln 14 und 15.

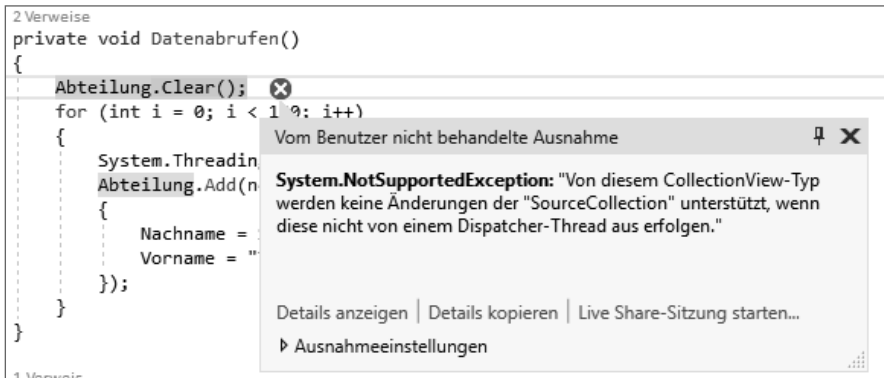
Gesagt, getan, wir kapseln obigen Methodenaufruf in einem extra Thread:

```

    private void ButtonHT_Click(object sender, RoutedEventArgs e)
    {
        Task.Run(Datenabrufen());
    }

```

Doch nach einem Start der Anwendung werden Sie schnell wieder auf den Boden zurückgeholt:



Sie können auf die Collection nicht per Hintergrundthread zugreifen. Das ist erst mal ein Show-Stopper. Doch es gibt zwei Lösungen:

- Laden einer extra Collection im Hintergrund und kopieren dieser Collection in den Vordergrund. Nachfolgend Abgleich mit der gebundenen Collection.
- Laden der Daten im Hintergrund, Einfügen der einzelnen Einträge durch jeweiligen Wechsel in den Vordergrundthread.

Die zweite Lösung ist mit häufigen Threadwechslern verbunden, wir sehen uns also die erste Lösung näher an.

Lösung (bis .NET 4.0)

Wir lagern das Laden der Daten in eine Funktion aus, die eine komplette Liste zurückgibt:

```

private ObservableCollection<Person> Datenabrufen_alteLoesung()
{
    ObservableCollection<Person> threadAbteilung =
        new ObservableCollection<Person>();
    for (int i = 0; i < 100; i++)
    {
        // simuliert Laden aus der Quelle
        System.Threading.Thread.Sleep(100);
        threadAbteilung.Add(new Person
        {
            Nachname = $"Müller{i}",
            Vorname = "Thomas"
        });
    }
    return threadAbteilung;
}

```

Unser Aufruf:

```

private void ButtonHL_Click(object sender, RoutedEventArgs e)
{

```

Anzeige, dass der Nutzer warten soll:

```

    Cursor = Cursors.Wait;

```

In einem extra Thread werden die Daten geladen:

```
Task.Factory.StartNew <ObservableCollection<Person>>(
    Datenabrufen_alteLoesung).ContinueWith(t =>
{
```

Ist dies erfolgt, kopieren wir die Daten in die gebundene Liste:

```
Abteilung.Clear();
foreach (var s in t.Result)
    klasse.Add(s);
```

Und blenden die Sanduhr aus:

```
    this.Cursor = null;
}, TaskScheduler.FromCurrentSynchronizationContext());
}
```

Test

Nach dem Start wird die „Sanduhr“ angezeigt, nach einigen Sekunden ist die Liste gefüllt. Wie Sie sehen, muss sich der Nutzer auch hier gedulden, die Oberfläche bleibt in dieser Zeit aber voll bedienbar.

Lösung (ab .NET 4.5)

.NET 4.5 bietet die Möglichkeit, Collections für die gleichzeitige Bearbeitung in Threads quasi anzumelden. Nutzen Sie dazu einen Aufruf der Methode *EnableCollectionSynchronization*.

```
private void ButtonLN_Click(object sender, RoutedEventArgs e)
{
```

Anmelden der Collection – wir übergeben noch das aktuelle Window-Objekt als Sperrojekt⁴:

```
BindingOperations.EnableCollectionSynchronization(Abteilung, this);
```

Wir rufen die Daten per extra Thread ab:

```
Task.Run(Datenabrufen);
}
```

Test

Nach dem Start werden Sie feststellen, dass die Oberfläche beweglich bleibt und dass die Daten „tröpfchenweise“ in die Liste geladen werden. Sie können beim Füllen quasi zusehen – eine einfache und recht elegante Lösung für das Einlesen größerer Datenmengen.



HINWEIS: Eine alternative Lösung wäre auch noch die Verwendung des *async-await*-Musters. Dazu mehr in Abschnitt 14.9.

⁴ Auch jede andere Objekt-Instanz ist geeignet.

Teil III: Technologien



- Zugriff auf das Dateisystem (Kapitel 12)
- Dateien lesen und schreiben (Kapitel 13)
- Asynchrone Programmierung (Kapitel 14)
- Die Task Parallel Library (Kapitel 15)
- Fehlersuche und Behandlung (Kapitel 16)
- JSON und XML in Theorie und Praxis (Kapitel 17)
- Einführung in ADO.NET und Entity Framework (Kapitel 18)
- Weitere Techniken (Kapitel 19)

Index

Symbole

.BAML 424
.G.CS 424
.NET-Framework 22
??-Operator 57

A

Abbruchbedingung 805
Abhängige Eigenschaften 543
Abort 714
Abs 268
abstract 167
Abstraktion 20
AcceptReturn 471
Access-Key 467
Accessor 133
Activated 453
Activator 975, 977
ActualHeight 467
ActualWidth 467
Add 869
AddAfterSelf 869
AddBeforeSelf 869
AddDays 264
AddExtension 686
AddFirst 869
AddHours 264
AddMinutes 264
AddMonths 264
AddRange 296
AddressList 982
AddressWidth 987
AddYears 264
ADO.NET 899
ADO.NET-Klassen 903
ADO.NET-Objektmodell 900
Aktionsabfrage 951
AllowsTransparency 456
Ancestors 866
Anfangswerte 52
Angehängte Eigenschaften 545
AngleX 581
AngleY 581
Animationen 584
Anonyme Methoden 308, 330
Anonyme Typen 341
App 417, 450
App.config 814
App.Current 417
Append 674
AppendChild 851, 852, 856, 857
AppendText 675
Application 983
ApplicationCommands 558
ApplyPropertyValue 516
App.xaml 417
App.xaml.cs 417
Array 108, 225, 237, 374, 379
ArrayList 295, 303
as-Operator 61
Assemblierung 10, 26, 35
Assembly
– dynamisch laden 975
– GetExecutingAssembly 972
– laden 972
– LoadFrom 972
async 748
Asynchrone Programmierentwurfsmuster 736
Atn 268
Attached Properties 429, 545
Attribute 27, 838, 841, 849

Attributes 650
 Auflistung 293
 Aufzählungszeichen 507
 Ausgabefenster 801
 Ausnahmen 825
 Ausnahmenfilter 393
 Ausschneiden 971
 Auswahlabfrage 955
 AutoGenerateColumns 628
 AutoProperties 391
 Auto-Property 138
 AutoToolTipPlacement 490
 AutoToolTipPrecision 490
 await 748

B

Background 444, 465, 514
 BandIndex 501
 Barrier 792
 base 157
 Basisklassen 156
 Beep 712
 Befehlsfenster 798
 Begin 586
 BeginAnimation 585
 BeginInit 485
 BeginInvoke 732, 744
 BeginningEdit 633
 Benannte Styles 562
 BigInteger 318
 Binärdatei 678
 Binary Application Markup Language 424
 BinaryFormatter 208, 210, 672, 681
 BinaryReader 672, 678, 679
 BinarySearch 237
 BinaryWriter 672, 678, 679
 Binding 589
 BindingBase.StringFormat 627
 BindingSource 208
 Bindungsarten 590
 BitmapFrame 485
 BitmapImage 485
 BlackoutDates 539
 BlockingCollection 791
 Blocks 515
 BlockUIContainer 510, 522
 bool 50
 Boolesche Operatoren 72
 Border 505
 BorderBrush 465

BorderThickness 505
 Boxing 65
 break 81, 107
 Breakpoints 803
 BringIntoView 534
 Bubbling Events 552
 BulletDecorator 507
 Button 468
 byte 49

C

Calendar 536
 Callback 739
 CallerFilePath 816
 Caller Information 816
 CallerLineNumber 816
 CallerMemberName 816
 CamelCase-Notation 120
 CancellationToken 783
 CancellationTokenSource 783
 CanExecute 561
 Canvas 426, 432
 Canvas.Bottom 433
 Canvas.Right 433
 CaretBrush 472
 CaretIndex 471
 case 75, 107
 C#-Compiler 5
 CenterOwner 454
 CenterScreen 454
 CenterX 581
 CenterY 581
 ChangeDatabase 909
 char 50
 CheckAccess 733
 CheckBox 474
 CheckFileExists 659, 685
 CheckPathExists 659
 ChildNodes 854
 class 9, 110, 119
 ClassLoader 25
 Clear 237
 Clipboard 969
 - ContainsText 970
 - SetText 970
 Clone 236, 245, 246
 Close 908
 CLR 22, 25
 CLR-Threadpool 764
 CLS 22

Code Contracts 829
Codefenster 19
Code Manager 25
Collapsed 466
Collection 293, 380, 601
CollectionView 612
CollectionViewSource 604
ColumnDefinitions 439
ColumnSpan 444
ComboBox 480
COM-Komponenten 26
Command 554, 556, 911, 927
CommandBuilder 921
CommandLine 988
CommandTarget 557
CommandText 912
CommandTimeout 913
CommandType 914
COM-Marshaller 25
Common Language Runtime 20, 25
Common Language Specification 22, 23
Common Type System 22, 24
CompareDocumentOrder 866
Completed 586
Complex 321
ComponentCommands 558
ConcurrentBag 791
ConcurrentDictionary 791
ConcurrentQueue 792
ConcurrentStack 792
Connection 904, 912
ConnectionString 907
ConnectionStringBuilder 909
ConnectionTimeout 908
const 56
Constraint 304
ContentPresenter 573
ContextMenu 498
continue 78
ControlTemplate 573
Convert 63, 625
ConvertBack 625
ConvertStringToByteArray 695
Copy 649
CopyTo 236, 245, 246, 649
CornerRadius 505
Cos 268
CountdownEvent 792
Create 674
CreateCommand 909, 912
CreateDirectory 646

CreateElement 852
CreateInstance 977
CreateNavigator 874, 894
CreateSubdirectory 646
CreateText 675
CreationTime 650
CryptoStream 683
CryptoStreams 693
C#-Source-Datei 7
CSV-Datei 702, 703
CTS 22
Current 294
CurrentClockSpeed 987
CurrentDirectory 988
CurrentItem 604, 612
CurrentPosition 612
Cursor 465

D

DataAdapter 925
Database 907
DataBindings 211
Data Encryption Standard 684, 693
DataFormats 513
DataGrid 535, 627
DataGridCheckBoxColumn 628
DataGridTextColumn 628
DataGridView 208
DataReader 922
DataSource 907
DataTemplate 606
Dateattribute 650
Dateien
– kopieren und verschieben 649
– umbenennen 649
– verschlüsseln 693
Dateiname
– erzeugen 706
Dateiparameter 673
Datenkonsument 899
Datenprovider 899, 900
Datenstrukturen 791
Daten-Trigger 571
Datentypen 49, 106
Datenzugriff 84
DatePicker 536
DateTime 263
Datumsformatierung 273
Datumsfunktionen 262
Day 263

DayOfWeek 263
 DayOfYear 263
 DaysInMonth 265
 DbProviderFactories 902
 Deactivated 453
 Deadlocks 709
 Debug 808
 – Write 809
 – WriteIf 809
 – WriteLineIf 809
 Debugger 797
 decimal 50
 DecodePixelWidth 485
 Decrypt 683
 default 78
 DefaultExt 685
 DefaultView 603
 Dekrement 68
 Delay 593
 delegate 110, 142
 Delegate 305, 330
 – instanziiieren 308
 Delete 646
 DeleteCommand 927
 Dependency Injection 171
 DependencyObject 544
 Dependency Properties 543
 Depth 877
 DereferenceLinks 659
 DES 693
 Descendants 866
 Deserialize 210
 Designer 18
 DesktopDirectory 660
 Destruktor 148, 152
 Diagnostics 712
 Dictionary 303
 Dimensionsgrenzen 231
 Direction 920
 Directory 640, 644
 DirectoryInfo 640
 DirectoryName 645
 DirectX 410
 Direkte Events 554
 DispatcherUnhandledException 453
 DisplayDate 537
 DisplayMemberPath 606
 DisplayMode 536
 Dispose 154, 917
 Distinct 371
 do 79, 80

DockPanel 426, 435
 DockPanel.Dock 435
 DOCTYPE 841
 Document 511
 Document Object Model 848
 Document Type 848
 DOM 848
 double 50
 DriveInfo 640
 Duplikate 371
 DynamicResource 549
 Dynamische Programmierung 313

E

EditingCommands 517, 558
 Eigenschaften 11, 128
 Eigenschaften-Fenster 19
 Eigenschaften-Trigger 568
 Eigenschaftsmethoden 218
 Einfügen 971
 Einzelschrittmodus 806
 Element 838, 841, 848
 Elements 866
 Ellipse 541
 else 107
 else if 75
 EnableRaisingEvents 654
 Encrypt 683
 EndInvoke 732, 744
 EndsWith 240
 Enter 722
 Entity Framework 899
 Entwicklungsumgebung 13
 enum 82, 109
 Enumerable 374, 378
 Enumerationen 109
 Ereignis 12, 110, 142
 – auslösen 144
 Ereignismethoden 558
 Ereignis-Trigger 570
 Erweiterungsmethoden 342, 362
 event 110, 142
 EventLog 815
 EventLogTraceListener 815
 Events 12
 Exception 826
 ExceptionManager 25
 ExecuteNonQuery 911, 915
 ExecuteReader 911, 915, 923
 ExecuteScalar 911, 916

Exists 650
Exit 453, 722
Exp 268
ExpandDirection 526
Expander 526
Exponentialfunktion 269
ExtClock 987
eXtensible Application Markup Language 411
Extension 645
Extension-Method-Syntax 345, 362

F

Fehlerbehandlung 817
Fehlerklassen 819, 827
FieldCount 924
File 640, 672
FileAccess 673
FileDropList 969
FileInfo 640, 672, 675
FileMode 674
FileName 659, 686
FileShare 674
FileStream 210, 672
FileSystemWatcher 640, 654
Fill 445, 928
Filter 614, 654, 685
FilterIndex 659
Filters 659
FindResource 595
FirstChild 853, 883, 892
float 50
FlowDirection 434
FlowDocument 510, 511, 521
FlowDocumentPageViewer 520
FlowDocumentReader 520
FlowDocumentScrollViewer 521
FontFamily 465
Fonts 660
FontSize 465
FontStyle 465
FontWeight 465
for 79, 108
for-each 881
foreach 108, 165, 229, 303
Foreground 465
Form1.cs 18
Format 273, 624
Formatters 210
Formulare 11
FromCurrentSynchronizationContext 788

FullName 645
Funktionen 109

G

Garbage Collector 152
Generics 300
Generische Schnittstelle 372
get 110, 133
GetCreationTime 650
GetCurrentDirectory 647
GetDataObject 969
GetDataPresent 970
GetDefaultView 603
GetDirectories 647
GetElementsByTagName 858
GetEnumerator 294, 302
GetFactoryClasses 902
GetFields 973
GetFiles 652
GetHostEntry 982
GetHostName 982
GetLength 236, 246
GetMembers 973
GetMethod 977
GetMethods 974
GetProperties 974
Getter-only Auto-Property 138
GetValue 925
GetValues 925
goto 78
Grafikausgabe 410
Grafikskalierung 487
Grid 426, 439
Grid.Column 442
Grid.Row 442
GridSplitter 444
GroupBox 506
GroupName 476
Gruppen 977

H

Haltepunkte 805
Hardwarebeschleunigung 410
Hashtable 297
HasValue 57
Header 526
Hidden 466
HorizontalAlignment 432, 443, 465
HorizontalContentAlignment 466

HorizontalOffset 529
 HorizontalScrollBarVisibility 492
 Hour 263
 HTML 836

I

IAsyncResult 738, 739, 741, 745
 ICollection 295
 IComparable 203
 IComparer 203
 Icon 456
 ICryptoTransform 696
 IDataObjekt 970
 IDisposable 917
 IEnumerable 293, 374, 382
 IEnumerator 294
 IEqualityComparer 372
 if 75, 107
 ILDASM 744
 Image 484
 immutable 278
 Indent 880
 IndentChars 880
 IndentLevel 810
 IndentSize 810
 Index 225
 Indexer 218, 292, 299, 329
 IndexOf 237, 240
 Indexprüfung 228
 InitialDirectory 686
 Initialisierer 374
 Initialisierung 121
 Initialize 236, 246
 InitializeComponent 417
 Inkrement 68
 InlineUIContainer 519
 INotifyCollectionChanged 601
 INotifyPropertyChanged 598
 InputGestureText 495
 Insert 240
 InsertBefore 515
 InsertCommand 927
 InsertParagraphBreak 514
 InsertTextInRun 514
 Instanz 116
 Instanzieren 120
 int 49
 Int16 49
 Int32 49
 Int64 50

Intellisense 126
 internal 118
 internal protected 118
 Interrupt 713
 InvalidOperationException 605
 Invoke 731, 741, 744, 975, 977
 InvokeRequired 733
 IP-Adresse 981
 IsAbstract 973
 IsAfter 866
 IsAlive 715
 IsBackGround 715
 IsBefore 866
 IsCheckable 498
 IsChecked 474, 476, 498
 IsClass 973
 IsClosed 924
 IsCOMObject 973
 IsCompleted 738, 771
 IsCurrentAfterLast 603, 612
 IsCurrentBeforeFirst 604, 612
 IsDirectionReversed 491
 IsEditable 481
 IsEnum 973
 IsExpanded 527, 534
 IsIndeterminate 504
 IsInterface 973
 IsLeapYear 265
 IsLocked 501
 IsMuted 488
 IsOpen 529
 IsPublic 973
 IsReadOnly 471
 IsSealed 973
 IsSelected 534
 IsSelectionRangeEnabled 491
 IsSnapToTickEnabled 491
 IsSynchronizedWithCurrentItem 605
 IsThreeState 474
 Item 924
 Iterator 302, 771
 iTextSharp 698
 IValueConverter 625

J

JIT-Compiler 21
 Join 713
 JsonIgnore 965

K

Kapselung 20, 116
 Kartenspiel 213
 Kartesische Koordinaten 194
 Kind-Elemente 442
 Klasse 116
 Klassendefinition 110
 Kommentare 48, 838
 Komplexe Zahlen 194
 Konsolenanwendung 94
 Konstante Felder 136
 Konstanten 49, 56
 Konstruktor 148
 – überladen 218
 Kontravarianz 317
 Kopieren 971
 Kovarianz 317
 Kurz-Operatoren 70
 Kurzschlussauswertung 72

L

Label 467
 Lambda-Ausdruck 310, 362, 368
 Lambda Expression 330
 LastAccessTime 650
 LastChildFill 436
 LastWriteTime 650
 Laufwerke 648
 Layout 426
 Leerzeichen 448
 Length 236, 240, 245, 246
 Line 542
 LineBreak 449
 LINQ 367, 370, 379, 380, 383, 704
 – Abfrageoperatoren 346
 – Aggregat-Operatoren 355
 – AsEnumerable 358
 – Count 355
 – GroupBy 352
 – Grundlagen 339
 – Gruppierungsoperator 352
 – Join 354
 – Konvertierungsmethoden 358
 – OrderBy 350
 – OrderByDescending 350
 – Projektionsoperatoren 348
 – Restriktionsoperator 350
 – Reverse 352
 – Select 348

– SelectMany 348
 – Sortierungsoperatoren 350
 – Sum 356
 – ThenBy 350
 – ToArray 358
 – ToDictionary 358
 – ToList 358
 – ToLookup 358
 – Where 350
 LINQ-Abfrageoperatoren 344
 LINQ-Architektur 339
 LINQ-Provider 340
 LINQ-Syntax 344
 LINQ to XML-API 861
 List 300, 303, 510, 522
 ListBox 477
 List-Klasse 303
 ListView 535, 609
 Live Share 403
 Load 864
 LoadedBehavior 488
 LoadXml 850
 LocalApplicationData 660
 lock 719
 Log 268
 Log10 268
 Logarithmus 269
 Logische Operatoren 71
 Lokale Variablen 59
 Lokal-Fenster 800
 long 50
 LongRunning 787
 LowestBreakIteration 771

M

MainWindow.xaml 418
 MainWindow.xaml.cs 418
 ManagementObject 978
 ManagementObjectSearcher 978
 ManualResetEventSlim 792
 Manufacturer 987
 Margin 430
 Marshalling 682
 Matrix 218
 MatrixTransform 579
 Matrizen 222
 Max 268
 MaxHeight 429, 466
 MaxLength 471
 MaxLines 471

MaxWidth 429, 466
 Media 712
 MediaCommands 558
 MediaElement 488
 Menu 494
 Menü
 – Grafiken 496
 – Tastenkürzel 495
 MenuItem 494
 Menüleiste 493
 MenuStrip 983
 Messwertliste 365
 Metadaten 27
 Metasprache 836
 Methoden 12, 86, 109, 138
 – generische 301
 – überladen 101, 218
 Methodenzeiger 305
 MethodImpl 727
 Methods 12
 Microsoft Intermediate Language Code 21
 Min 268
 MinHeight 429, 466
 MinLines 471
 Minute 263
 MinWidth 429, 466
 Modale Dialoge 468
 Monitor 722
 Month 263
 Move 646, 649
 MoveCurrentTo 613
 MoveCurrentToFirst 604, 613
 MoveCurrentToLast 603, 613
 MoveCurrentToNext 603, 613
 MoveCurrentToPosition 613
 MoveCurrentToPrevious 604, 613
 MoveNext 294
 MoveTo 649
 MoveToNext 874, 891
 MoveToPrevious 874, 891
 MoveToRoot 892
 MSIL-Code 21, 35
 MultipleRange 537
 MultiSelect 659
 Multitasking 708
 Multithreading 29, 708
 Mutex 726
 MyComputer 660
 MyDocuments 660

N

Name-Attribut 420
 Namespace 25, 121, 973
 NaturalDuration 488
 NaturalVideoHeight 488
 NaturalVideoWidth 488
 NavigationCommands 558
 NET-Reflection 971
 new 110, 148, 226
 Next 376
 NextSibling 854, 883
 Nodes 866
 NodeType 878
 NotifyFilter 654
 Now 265
 Nuget 698
 NuGet 395
 null 57, 123
 Nullable Type 56
 Nutzer ermitteln 977

O

object 50, 55
 Objekt 116
 Objektbaum 208, 688
 Objekte 110
 Objektinitialisierer 151, 341, 342
 ObservableCollection 601
 ODER 73
 OneTime 590
 OneWay 590
 OneWayToSource 590
 OnExplicitShutdown 453
 OnLastWindowClose 453
 OnMainWindowClose 453
 OnStartup 452
 OOP 213
 Opacity 456, 466
 Open 674, 908
 OpenFileDialog 486, 657
 OpenOrCreate 674
 OpenText 676
 Operatoren 67, 107
 – arithmetische 68
 – boolesche 72
 – logische 71
 Operatorenüberladung 194
 Optionale Parameter 317
 orderby 380

Orientation 433, 490, 492
 OSVersion 985
 out 90
 OverflowMode 502
 override 157
 OverwritePrompt 686

P

Padding 430, 466
 PadLeft 240
 PadRight 240
 PAP 94
 Paragraph 510, 522
 Parallel.For 767
 Parallel.ForEach 772
 Parallel.Invoke 765
 Parallel LINQ 792
 ParallelLoopResult 770
 Parallel-Programmierung 761
 Parameter 919
 ParameterName 920
 Parameterübergabe 89, 90
 Parent 466
 ParentNode 883
 Parse 64, 265
 Parser 840
 Pascal-Notation 120
 PasswordBox 471, 473
 Path 640, 653
 Pattern Matching 396
 Pause 586
 PDF 698
 PDFsharp 700
 PeekChar 679
 Pi 268
 PI 840
 Placement 529
 PlacementRectangle 530
 PlacementTarget 530
 Platform 985
 PLINQ 359, 792
 Polarkoordinaten 194
 Polling 738
 Polymorphes Verhalten 163
 Polymorphie 21, 117, 155, 165
 Popup 529
 Portieren 104
 Potenz 269
 Pow 268
 PreferFairness 787

PresentationHost 413
 PreviousSibling 883
 Priority 715
 private 118
 private protected 402
 Procedure-Step 803
 Process 755, 759
 Processing Instructions 837, 840, 848
 ProcessorCount 987
 Process.Start 760
 ProcessThread 755
 Programm starten 758
 ProgressBar 504
 Projektmappen-Explorer 17
 Projekttyp 16
 Properties 11, 174
 Property-Accessoren 133
 PropertyChanged 592
 protected 118
 Provider 907
 Prozeduren 109
 Prozedurschritt 807
 Prozesse 755
 public 118
 Pulse 722, 723
 PulseAll 722, 723

Q

Query-Expression-Syntax 345, 362
 Queue 300, 303
 QueueUserWorkItem 716

R

Racing 710
 RadioButton 476
 Rahmenbreite 505
 Random 214, 270, 271, 376
 Range 375
 Rank 236, 245, 246
 Read 673
 ReadAllBytes 679
 ReadAllLines 677
 ReadAllText 677
 ReadContentAsFloat 878
 ReadLine 9
 ReadLines 677
 ReadToEnd 676
 ReadWrite 673
 Rechtschreibkontrolle 519

Rectangle 541
 ref 89
 Referenzieren 120
 Referenztyp 55, 239
 Reflexion 27
 ReleaseMutex 726
 Remove 240, 870
 RemoveAll 870
 RemoveAnnotations 870
 RemoveAttributes 870
 RemoveContent 870
 Repeat 376
 RepeatButton 468
 Replace 240
 Reset 294
 Resources 466
 Ressourcen 545
 Resume 586
 return 78, 110, 302
 RichTextBox 509
 RotateTransform 579
 Round 268
 Routed Events 552
 RowDefinitions 439
 RowDetailsTemplate 631
 RowDetailsVisibilityMode 631, 632
 RowSpan 444
 Rückrufmethode 737
 Rücksprung 807

S

SaveFileDialog 657, 660
 ScaleTransform 579
 ScaleX 581
 ScaleY 581
 Schaltjahr 265
 Schleifen 108
 Schleifenabbruch 769
 Schleifenanweisungen 79
 Schlüsselwörter 47, 290, 335
 ScrollBar 492
 ScrollViewer 492
 sealed 168
 Second 263
 Section 510, 522
 Security Engine 25
 Seek 586
 select 380
 Select 874
 SELECT 955
 SelectCommand 926, 927
 SelectedDate 537
 SelectedItem 534
 SelectedItemChanged 534
 SelectedItems 479
 Selection 514
 SelectionBrush 472
 SelectionMode 477
 SelectNodes 858
 SelectSingleNode 854, 856, 857, 858, 892
 Semaphore 728
 SemaphoreSlim 792
 Separator 494
 Sequenzielle Datei 680
 Serialisieren 681
 Serialisierung 28
 Serializable 208, 681
 Serializable-Attribut 689
 Serialization 210
 Serialize 210
 ServerVersion 908
 ServicePack 985
 SessionEnding 453
 set 110, 129, 133
 SetAttributeValue 869
 SetCurrentDirectory 647
 SetDataObject 969
 SetElementValue 869
 Shared-Methoden 218
 short 49
 Show 456
 ShowDialog 456
 Shutdown 453
 Sign 268
 Sin 268
 SingleRange 537
 Single-Step 803
 Skalieren mit ScaleTransform 580
 SkewTransform 579
 Skip 868
 SkipWhile 868
 Sleep 714
 Slider 490
 SocketDesignation 987
 Sort 203, 237
 SortDescriptions 614
 SortedList 303
 SortedSet 322
 Sortieren 379
 Source 484
 SpecialFolder 660

SpeedRatio 488
 SpellCheck.IsEnabled 471
 Sperrmechanismen 717
 SpinLock 792
 SpinWait 792
 Split 240
 SqlConnection 904
 Sqr 268
 Stack 300
 Stackpanel 426
 StackPanel 433
 Stand-alone-XAML 413
 StartInfo 759
 StartsWidth 240
 Startup 452, 453
 StartupEventArgs 452
 StartupUri 417, 450
 State 908
 static 110, 170
 StaticResource 548, 565
 Statische Klassen 170
 Statische Methoden 140
 Statischer Konstruktor 151
 StatusBar 502
 StatusBarItems 503
 Steuerelemente 11
 Stop 586
 StoredProcedure 914
 StoryBoard 584
 StreamReader 672, 676
 StreamWriter 672, 675, 702
 Stretch 487
 StretchDirection 487
 string 50
 String 239
 Stringaddition 278
 StringReader 672
 StringWriter 672
 struct 83, 108
 Strukturen 108
 Strukturvariable 85
 Style 466, 564
 Style anpassen 565, 567
 Style ersetzen 565
 Styles vererben 566
 Subklassen 157
 SubString 240
 switch 75, 107
 System 49, 660
 System.Collections.Concurrent 791
 System.Diagnostics 756

System.Environment 983
 SystemInformation 983
 System.IO.FileStream 671
 System.IO.Stream 671
 System.Management 978
 System.Net 982
 System.Nullable 57
 System.Object 166
 SystemParameters 551
 System.Reflection 971
 Systemressourcen 550
 System.Security.Cryptography 683
 System.Threading 711, 764
 System.Threading.Tasks 764
 System.Xml 849, 885, 889
 System.Xml.Linq 861
 System.Xml.XPath 877
 System.Xml.Xsl 881

T

TabControl 527
 TabIndex 466
 Table 510, 522
 TableDirect 914
 TabPanel 426
 Tag 466
 Take 868
 TakeWhile 868
 Tan 268
 Target 467
 Task

- Canceled 787
- ContinueWith 779, 788
- Created 787
- Datenübergabe 775
- Faulted 787
- Fehlerbehandlung 785
- IsCanceled 787
- IsCompleted 787
- IsFaulted 787
- Klasse 773
- RanToCompletion 787
- Result 779
- return 782
- Rückgabewerte 778
- Running 787
- starten 774
- Status 787
- TaskCreationOptions 787
- Task-Ende 788

- Task-Id 786
- User Interface 788
- Verarbeitung abberechnen 781
- Wait 777
- WaitAll 778
- WaitingForActivation 787
- WaitingForChildrenToComplete 787
- WaitingToRun 787
- weitere Eigenschaften 786
- Task<> 752
- Task.Factory.StartNew 773
- Task Parallel Library 761
- TaskScheduler 788
- Template 572
- Text 481
- Textausrichtung 449
- TextBlock 446
- TextBox 471
- Textdatei 675, 684
- Texte formatieren 515
- Textformatierungen 447
- TextFormattingMode 459
- TextPointer 514
- TextRange 512, 513
- TextWriterTraceListener 813
- Thin Client 179
- Thread 711, 713
- ThreadInterruptedException 713
- ThreadPool 716
- Threads 755
- Thread Service 25
- threadsicher 730
- Threadsichere Collections 791
- ThreadState 715
- Throw 820, 827
- ThrowIfCancellationRequested 783
- TickFrequency 491
- TickPlacement 491
- Timer-Threads 735
- TimeSpan-Klasse 278
- Title 455, 659, 686
- ToArray 371, 380, 382
- ToCharArray 240
- Today 265
- ToggleButton 468
- ToLongDateString 264
- ToLongTimeString 264
- ToLower 240
- ToolBar 499
- ToolBarTray 499, 500
- Toolbox 18

- ToolTip 466
- ToShortDateString 264
- ToShortTimeString 264
- ToString 62, 272
- ToUpper 240
- Trace 808, 812
- TraceListener 813
- TrackBar 278
- Transform 881
- Transformationen 579
- Transformationsdatei 881
- TransformGroup 582
- TranslateTransform 579
- Transparenz 456
- TreeView 531, 666, 887
- Trefferanzahl 806
- Trigger 568
- Trim 240
- Truncate 674
- try 107
- try-catch 818
- TryEnter 722, 725
- try-finally 822
- Tunneling Events 552
- Tuple 321
- TwoWay 590
- Type 973
- Typecasting 327
- Typinferenz 58, 341, 362
- Typ-Styles 564
- Typsuffixe 52

U

- Überladene Methoden 139
- Überwachungsfenster 800
- Uhr anzeigen 265
- UI-Virtualisierung 629
- Unboxing 65
- UND 73
- Unicode 53
- UnicodeEncoding 695
- Uniform 445
- UniformGrid 426, 438
- UniformToFill 445
- UnIndent 810
- Unmarshalling 682
- Unterverzeichnis ermitteln 647
- Update 922, 929
- UpdateCommand 927
- UpdateSourceTrigger 592

Uri 485, 550
 UriSource 485
 UserInteractive 988
 UserProfile 660
 using 9, 121, 289
 UTF-8 841
 UTF-16 841

V

ValidateNames 686
 Value 490, 920
 var 58
 Variablen 49
 Variablentypen 49
 VB 104
 Verarbeitungsstatus 770
 Vererbung 117
 Verformen mit SkewTransform 581
 Vergleichsoperatoren 71
 Verschieben mit TranslateTransform 581
 Verschlüsseln 682
 Version 989
 VersionString 985
 VerticalAlignment 432, 443, 465
 VerticalContentAlignment 466
 VerticalOffset 529
 VerticalScrollBarVisibility 492
 Verweistypen 50
 Verzweigungen 107
 ViewBox 426, 444
 VirtualizingStackPanel 629
 Visibility 466
 Visual Studio 4
 Visual Studio Enterprise 4
 Visual Studio Professional 4
 void 88

W

W3C 848
 Wait 722, 723
 WaitOne 726, 728
 Werkzeugkasten 18
 Wertetypen 50
 where 304
 Where 868
 while 79, 80, 108
 Wiederholmuster 378
 Wiederverwendbarkeit 117
 WindowsIdentity 980

Windows Management Instrumentations 983
 Windows Presentation Foundation 409
 WindowStartupLocation 454
 WindowStyle 455, 457
 Winkel 269
 WMI 983
 WorkingSet 988
 work stealing 765
 WPF 409, 733

- Anwendung beenden 453
- Applikationstypen 422
- Eigenschaften 465
- Ereignis-Handler 419
- Ereignismodell 551
- Height 429
- Kommandozeilenparameter 452
- Left 429
- Maßangaben 428
- Startobjekt festlegen 450
- Style-System 561
- Top 429
- Width 429
- Window-Klasse 454
- Zielplattformen 421

 WPF-Programm 449
 WPF-Wertkonvertierer 625
 Wrap 447
 WrapPanel 426, 437
 WrapWithOverflow 447
 Write 673
 WriteAllBytes 679
 WriteAttributeString 833, 834, 879
 WriteEndDocument 833, 834, 879
 WriteEndElement 833, 834, 879
 WriteIf 808
 WriteLine 9, 808
 WriteStartDocument 833, 834, 879
 WriteStartElement 833, 834, 879
 Wurzel 269

X

XAML 411
 XAML-Anwendung 412
 XAML-Editor 416
 XAttribute 862
 x:Class 418
 XComment 862
 XDeclaration 863
 XDocument 862, 864

- XDocumentType 863
- XElement 862
- XElement.Load 865
- XElement.Parse 865
- XML 831
- XmlAttribute 849
- XmlCDATASection 849
- XmlCharacterData 849
- XmlComment 849
- XmlDocument 874, 882
- XmlDocumentType 849
- XmlElement 849
- XmlEntity 849
- XmlImplementation 849
- XmlNamedNodeMap 849
- XmlNode 849, 853
- XmlNodeList 849
- XmlProcessingInstruction 849
- XmlReader 876
- XmlReaderSettings 877
- XML-Schema 843
- xml:space 449
- XmlText 849
- XML transformieren 870
- XmlWriter 878

- XmlWriterSettings 880
- XNode 863
- XOR 73
- XPathDocument 874
- XPathNavigator 874, 891, 894
- XPathNodeIterator 874
- XProcessingInstruction 863
- xsd.exe 848
- XSD-Schema 842
- XslCompiledTransform 881
- XSLT 880

Y

- Year 263
- yield 302, 330

Z

- Zahlenformatierung 272
- Zeilenumbrüche 448
- Zeitfunktionen 262
- Zeitmessung 279
- Zufallszahlen 270, 271, 376
- Zuweisungsoperatoren 70