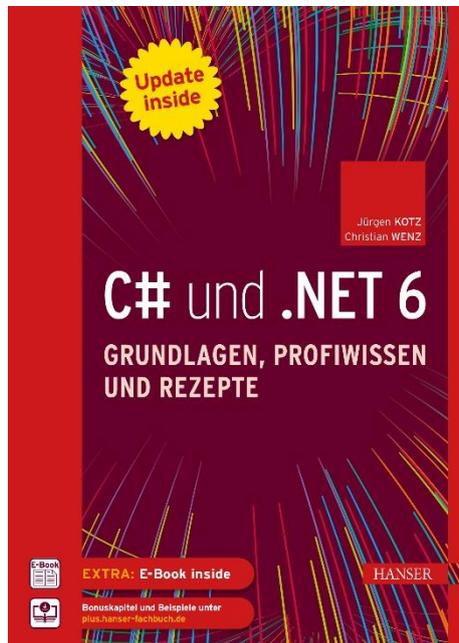


# HANSER



## Leseprobe

zu

## C# und .NET 6

von Jürgen Kotz und Christian Wenz

Print-ISBN: 978-3-446-46930-3

E-Book-ISBN: 978-3-446-47349-2

E-Pub-ISBN: 978-3-446-47423-9

Weitere Informationen und Bestellungen unter

<https://www.hanser-kundencenter.de/fachbuch/artikel/9783446469303>

sowie im Buchhandel

© Carl Hanser Verlag, München

# Inhalt

<b>Vorwort</b> .....	<b>XXI</b>
Zusatzmaterial online .....	XXIII
<b>Teil I: Grundlagen</b> .....	<b>1</b>
<b>1 .NET 6</b> .....	<b>3</b>
1.1 Microsofts .NET-Technologie .....	4
1.1.1 Zur Geschichte von .NET .....	4
1.1.2 .NET-Features und Begriffe .....	6
1.2 .NET Core .....	13
1.2.1 Geschichte von .NET Core .....	13
1.2.2 LTS - Long Term Support und zukünftige Versionen .....	15
1.2.3 .NET Standard .....	15
1.3 Features von .NET 6 .....	16
<b>2 Einstieg in Visual Studio 2022</b> .....	<b>19</b>
2.1 Die Installation von Visual Studio 2022 .....	19
2.1.1 Überblick über die Produktpalette .....	19
2.1.2 Anforderungen an Hard- und Software .....	20
2.2 Unser allererstes C#-Programm .....	21
2.2.1 Vorbereitungen .....	21
2.2.2 Quellcode schreiben .....	24
2.2.3 Programm kompilieren und testen .....	24
2.2.4 Einige Erläuterungen zum Quellcode .....	25
2.2.5 Konsolenanwendungen sind out .....	26
2.3 Die Windows-Philosophie .....	27
2.3.1 Mensch-Rechner-Dialog .....	27
2.3.2 Objekt- und ereignisorientierte Programmierung .....	27
2.3.3 Programmieren mit Visual Studio 2022 .....	29
2.4 Die Entwicklungsumgebung Visual Studio 2022 .....	30
2.4.1 Neues Projekt .....	30
2.4.2 Die wichtigsten Fenster .....	33
2.4.3 Projektvorlagen in Visual Studio 2022 - Minimal APIs .....	36

2.5	Praxisbeispiele .....	37
2.5.1	Unsere erste Windows-Forms-Anwendung .....	37
2.5.2	Umrechnung Euro-Dollar .....	43
<b>3</b>	<b>Grundlagen der Sprache C# .....</b>	<b>53</b>
3.1	Grundbegriffe .....	53
3.1.1	Anweisungen .....	53
3.1.2	Bezeichner .....	54
3.1.3	Schlüsselwörter .....	55
3.1.4	Kommentare .....	56
3.2	Datentypen, Variablen und Konstanten .....	57
3.2.1	Fundamentale Typen .....	57
3.2.2	Werttypen versus Verweistypen .....	58
3.2.3	Benennung von Variablen .....	59
3.2.4	Deklaration von Variablen .....	59
3.2.5	Typsuffixe .....	60
3.2.6	Zeichen und Zeichenketten .....	61
3.2.7	object-Datentyp .....	63
3.2.8	Konstanten deklarieren .....	64
3.2.9	Nullable Types .....	64
3.2.10	Typinferenz .....	66
3.2.11	Gültigkeitsbereiche und Sichtbarkeit .....	66
3.3	Konvertieren von Datentypen .....	67
3.3.1	Implizite und explizite Konvertierung .....	67
3.3.2	Welcher Datentyp passt zu welchem? .....	69
3.3.3	Konvertieren von string .....	70
3.3.4	Die Convert-Klasse .....	71
3.3.5	Die Parse-Methode .....	72
3.3.6	Boxing und Unboxing .....	73
3.4	Operatoren .....	74
3.4.1	Arithmetische Operatoren .....	75
3.4.2	Zuweisungsoperatoren .....	77
3.4.3	Logische Operatoren .....	78
3.4.4	Rangfolge der Operatoren .....	81
3.5	Kontrollstrukturen .....	82
3.5.1	Verzweigungsbefehle .....	82
3.5.2	Schleifenanweisungen .....	87
3.6	Benutzerdefinierte Datentypen .....	89
3.6.1	Enumerationen .....	90
3.6.2	Strukturen .....	91
3.7	Nutzerdefinierte Methoden .....	93
3.7.1	Methoden mit Rückgabewert .....	94
3.7.2	Methoden ohne Rückgabewert .....	95
3.7.3	Parameterübergabe mit ref .....	97

3.7.4	Parameterübergabe mit out .....	98
3.7.5	Methodenüberladung .....	99
3.7.6	Optionale Parameter .....	100
3.7.7	Benannte Parameter .....	101
3.8	Praxisbeispiele .....	102
3.8.1	Vom PAP zur Konsolenanwendung .....	102
3.8.2	Ein Konsolen- in ein Windows-Programm verwandeln .....	105
3.8.3	Schleifenanweisungen verstehen .....	107
3.8.4	Benutzerdefinierte Methoden überladen .....	110
3.8.5	Anwendungen von Visual Basic nach C# portieren .....	113
<b>4</b>	<b>OOO-Konzepte .....</b>	<b>121</b>
4.1	Kleine Einführung in die OOP .....	121
4.1.1	Historische Entwicklung .....	122
4.1.2	Grundbegriffe der OOP .....	123
4.1.3	Sichtbarkeit von Klassen und ihren Mitgliedern .....	125
4.1.4	Allgemeiner Aufbau einer Klasse .....	126
4.1.5	Das Erzeugen eines Objekts .....	128
4.1.6	Einführungsbeispiel .....	131
4.2	Eigenschaften .....	137
4.2.1	Eigenschaften mit Zugriffsmethoden kapseln .....	137
4.2.2	Berechnete Eigenschaften .....	139
4.2.3	Lese-/Schreibschutz .....	141
4.2.4	Property-Accessoren .....	142
4.2.5	Statische Felder/Eigenschaften .....	143
4.2.6	Einfache Eigenschaften automatisch implementieren .....	145
4.3	Methoden .....	147
4.3.1	Öffentliche und private Methoden .....	147
4.3.2	Überladene Methoden .....	148
4.3.3	Statische Methoden .....	149
4.4	Ereignisse .....	150
4.4.1	Ereignis hinzufügen .....	151
4.4.2	Ereignis verwenden .....	154
4.5	Arbeiten mit Konstruktor und Destruktor .....	157
4.5.1	Konstruktor und Objektinitialisierer .....	158
4.5.2	Destruktor und Garbage Collector .....	161
4.5.3	Mit using den Lebenszyklus des Objekts kapseln .....	163
4.6	Vererbung und Polymorphie .....	164
4.6.1	Method-Overriding .....	164
4.6.2	Klassen implementieren .....	164
4.6.3	Implementieren der Objekte .....	167
4.6.4	Ausblenden von Mitgliedern durch Vererbung .....	169
4.6.5	Allgemeine Hinweise und Regeln zur Vererbung .....	170

4.6.6	Polymorphes Verhalten	172
4.6.7	Die Rolle von System.Object	175
4.7	Spezielle Klassen	176
4.7.1	Abstrakte Klassen	176
4.7.2	Versiegelte Klassen	177
4.7.3	Partielle Klassen	178
4.7.4	Statische Klassen	179
4.8	Schnittstellen (Interfaces)	180
4.8.1	Definition einer Schnittstelle	181
4.8.2	Implementieren einer Schnittstelle	181
4.8.3	Abfragen, ob Schnittstelle vorhanden ist	182
4.8.4	Mehrere Schnittstellen implementieren	183
4.8.5	Schnittstellenprogrammierung ist ein weites Feld	183
4.9	Datensatztypen – Records	183
4.9.1	Definition eines Record	184
4.9.2	Mutable Properties	186
4.9.3	Nicht-destruktive Änderung	188
4.9.4	Dekonstruktion	189
4.10	Praxisbeispiele	190
4.10.1	Eigenschaften sinnvoll kapseln	190
4.10.2	Eine statische Klasse anwenden	193
4.10.3	Vom fetten zum schlanken Client	194
4.10.4	Schnittstellenvererbung verstehen	205
4.10.5	Rechner für komplexe Zahlen	210
4.10.6	Sortieren mit IComparable/IComparer	218
4.10.7	Einen Objektbaum in generischen Listen abspeichern	223
4.10.8	OOP beim Kartenspiel erlernen	228
4.10.9	Eine Klasse zur Matrizenrechnung entwickeln	233
4.10.10	Vererbung von Records	239
<b>5</b>	<b>Arrays, Strings, Funktionen</b>	<b>241</b>
5.1	Datenfelder (Arrays)	241
5.1.1	Array deklarieren	242
5.1.2	Array instanziiieren	242
5.1.3	Array initialisieren	243
5.1.4	Zugriff auf Array-Elemente	244
5.1.5	Zugriff mittels Schleife	245
5.1.6	Mehrdimensionale Arrays	246
5.1.7	Zuweisen von Arrays	248
5.1.8	Arrays aus Strukturvariablen	249
5.1.9	Löschen und Umdimensionieren von Arrays	250
5.1.10	Eigenschaften und Methoden von Arrays	252
5.1.11	Übergabe von Arrays	253

5.2	Verarbeiten von Zeichenketten .....	255
5.2.1	Zuweisen von Strings .....	255
5.2.2	Eigenschaften und Methoden von String-Variablen .....	256
5.2.3	Wichtige Methoden der String-Klasse .....	258
5.2.4	Die StringBuilder-Klasse .....	260
5.3	Datums- und Zeitberechnungen .....	263
5.3.1	Die DateTime-Struktur .....	263
5.3.2	Wichtige Eigenschaften von DateTime-Variablen .....	264
5.3.3	Wichtige Methoden von DateTime-Variablen .....	265
5.3.4	Wichtige Mitglieder der DateTime-Struktur .....	266
5.3.5	Konvertieren von Datumstrings in DateTime-Werte .....	267
5.3.6	Die TimeSpan-Struktur .....	267
5.3.7	DateOnly und TimeOnly .....	269
5.4	Mathematische Funktionen .....	270
5.4.1	Überblick .....	270
5.4.2	Zahlen runden .....	270
5.4.3	Winkel umrechnen .....	271
5.4.4	Potenz- und Wurzeloperationen .....	271
5.4.5	Logarithmus und Exponentialfunktionen .....	271
5.4.6	Zufallszahlen erzeugen .....	272
5.4.7	Kreisberechnung .....	273
5.5	Zahlen- und Datumsformatierungen .....	273
5.5.1	Anwenden der ToString-Methode .....	274
5.5.2	Anwenden der Format-Methode .....	275
5.5.3	Stringinterpolation .....	276
5.6	Praxisbeispiele .....	277
5.6.1	Zeichenketten verarbeiten .....	277
5.6.2	Zeichenketten mit StringBuilder addieren .....	280
5.6.3	Methodenaufrufe mit Array-Parametern .....	283
<b>6</b>	<b>Weitere Sprachfeatures .....</b>	<b>289</b>
6.1	Namespaces (Namensräume) .....	289
6.1.1	Ein kleiner Überblick .....	289
6.1.2	Einen eigenen Namespace einrichten .....	290
6.1.3	Die using-Anweisung .....	292
6.1.4	Namespace Alias .....	293
6.1.5	Globale using-Anweisungen .....	293
6.2	Operatorenüberladung .....	294
6.2.1	Syntaxregeln .....	294
6.2.2	Praktische Anwendung .....	295
6.3	Collections (Auflistungen) .....	296
6.3.1	Die Schnittstelle IEnumerable .....	296
6.3.2	ArrayList .....	299

6.3.3	Hashtable	300
6.3.4	Indexer	301
6.4	Generics	303
6.4.1	Generics bieten Typsicherheit	304
6.4.2	Generische Methoden	305
6.4.3	yield - Iteratoren	306
6.5	Generische Collections	306
6.5.1	List-Collection statt ArrayList	306
6.5.2	Vorteile generischer Collections	308
6.5.3	Constraints	308
6.6	Das Prinzip der Delegates	309
6.6.1	Delegates sind Methodenzeiger	309
6.6.2	Einen Delegate-Typ deklarieren	310
6.6.3	Ein Delegate-Objekt erzeugen	310
6.6.4	Anonyme Methoden	312
6.6.5	Lambda-Ausdrücke	313
6.6.6	Lambda-Ausdrücke in der Task Parallel Library	316
6.6.7	Action<> und Func<>	317
6.7	Dynamische Programmierung	319
6.7.1	Wozu dynamische Programmierung?	319
6.7.2	Das Prinzip der dynamischen Programmierung	320
6.7.3	Optionale Parameter sind hilfreich	322
6.7.4	Kovarianz und Kontravarianz	323
6.8	Weitere Datentypen	324
6.8.1	BigInteger	324
6.8.2	Complex	326
6.8.3	Tuple<>	327
6.8.4	SortedSet<>	328
6.9	Praxisbeispiele	329
6.9.1	ArrayList versus generische List	329
6.9.2	Generische IEnumerable-Interfaces implementieren	332
6.9.3	Delegates, Func, anonyme Methoden, Lambda Expressions	336
<b>7</b>	<b>Einführung in LINQ</b>	<b>341</b>
7.1	LINQ-Grundlagen	341
7.1.1	Die LINQ-Architektur	341
7.1.2	Anonyme Typen	343
7.1.3	Erweiterungsmethoden	344
7.2	Abfragen mit LINQ to Objects	345
7.2.1	Grundlegendes zur LINQ-Syntax	346
7.2.2	Zwei alternative Schreibweisen von LINQ-Abfragen	347
7.2.3	Übersicht der wichtigsten Abfrageoperatoren	348

7.3	LINQ-Abfragen im Detail .....	349
7.3.1	Die Projektionsoperatoren Select und SelectMany .....	350
7.3.2	Der Restriktionsoperator Where .....	351
7.3.3	Die Sortierungsoperatoren OrderBy und ThenBy .....	352
7.3.4	Der Gruppierungsoperator GroupBy .....	353
7.3.5	Verknüpfen mit Join .....	356
7.3.6	Aggregat-Operatoren .....	356
7.3.7	Verzögertes Ausführen von LINQ-Abfragen .....	358
7.3.8	Konvertierungsmethoden .....	359
7.3.9	Abfragen mit PLINQ .....	360
7.4	Praxisbeispiele .....	363
7.4.1	Die Syntax von LINQ-Abfragen verstehen .....	363
7.4.2	Aggregat-Abfragen mit LINQ .....	366
7.4.3	LINQ im Schnelldurchgang erlernen .....	369
7.4.4	Strings mit LINQ abfragen und filtern .....	371
7.4.5	Duplikate aus einer Liste entfernen .....	373
7.4.6	Arrays mit LINQ initialisieren .....	375
7.4.7	Arrays per LINQ mit Zufallszahlen füllen .....	377
7.4.8	Einen String mit Wiederholmuster erzeugen .....	379
7.4.9	Mit LINQ Zahlen und Strings sortieren .....	380
7.4.10	Mit LINQ Collections von Objekten sortieren .....	382
7.4.11	Where - Deep Dive .....	384
<b>8</b>	<b>Neuerungen von C# im Überblick .....</b>	<b>393</b>
8.1	C# 4.0 - Visual Studio 2010 .....	394
8.1.1	Datentyp dynamic .....	394
8.1.2	Benannte und optionale Parameter .....	395
8.1.3	Kovarianz und Kontravarianz .....	396
8.2	C# 5.0 - Visual Studio 2012 .....	396
8.2.1	Async und Await .....	397
8.2.2	CallerInfo .....	398
8.3	Visual Studio 2013 .....	399
8.4	C# 6.0 - Visual Studio 2015 .....	399
8.4.1	String Interpolation .....	399
8.4.2	Schreibgeschützte AutoProperties .....	399
8.4.3	Initialisierer für AutoProperties .....	400
8.4.4	Expression Body Member .....	400
8.4.5	using static .....	400
8.4.6	Bedingter Nulloperator .....	401
8.4.7	Ausnahmenfilter .....	402
8.4.8	nameof-Ausdrücke .....	402
8.4.9	await in catch und finally .....	403
8.4.10	Indexinitialisierer .....	403

8.5	C# 7.0 – Visual Studio 2017	403
8.5.1	out-Variablen	403
8.5.2	Tupel	404
8.5.3	Mustervergleich	405
8.5.4	Discards	406
8.5.5	Lokale ref-Variablen und Rückgabetypen	407
8.5.6	Lokale Funktionen	407
8.5.7	Mehr Expression Body Member	407
8.5.8	throw-Ausdrücke	408
8.5.9	Verbesserung der numerischen literalen Syntax	408
8.6	C# 7.1 bis 7.3 – Visual Studio 2019	408
8.6.1	C# 7.1	408
8.6.2	C# 7.2	410
8.6.3	C# 7.3	411
8.6.4	Visual Studio 2019 – Live Share	411
8.7	C# 8.0	414
8.7.1	Standardschnittstellenmethoden	414
8.7.2	Vereinfachung von switch-Ausdrücken	416
8.7.3	Eigenschaftenmuster	417
8.7.4	Vereinfachte using-Ressourcenschutzblöcke	417
8.7.5	Null-Coalescing-Zuweisungen	418
8.7.6	Nullable Referenztypen	419
8.7.7	Indizes und Bereiche	420
8.7.8	Weitere Sprachneuerungen	421
8.8	C# 9.0	422
8.8.1	Records	422
8.8.2	Init-only Setter	422
8.8.3	Weitere Verbesserungen in Musterausdrücken	422
8.8.4	Weitere Sprachneuerungen	423
8.9	C# 10	423
8.9.1	Datensatzstrukturen	423
8.9.2	Globale using-Anweisungen	423
8.9.3	Weitere Sprachneuerungen	424
<b>Teil II: Desktop-Anwendungen</b>		<b>425</b>
<b>9</b>	<b>Einführung in WPF</b>	<b>427</b>
9.1	Einführung	427
9.1.1	Was kann eine WPF-Anwendung?	428
9.1.2	Die eXtensible Application Markup Language	430
9.1.3	Unsere erste XAML-Anwendung	431
9.1.4	Zielplattformen	437
9.1.5	Applikationstypen	437
9.1.6	Vor- und Nachteile von WPF-Anwendungen	438
9.1.7	Weitere Dateien im Überblick	439

9.2	Alles beginnt mit dem Layout	441
9.2.1	Allgemeines zum Layout	441
9.2.2	Positionieren von Steuerelementen	443
9.2.3	Canvas	446
9.2.4	StackPanel	447
9.2.5	DockPanel	449
9.2.6	WrapPanel	451
9.2.7	UniformGrid	452
9.2.8	Grid	453
9.2.9	ViewBox	458
9.2.10	TextBlock	459
9.3	Das WPF-Programm	463
9.3.1	Die App-Klasse	463
9.3.2	Das Startobjekt festlegen	463
9.3.3	Kommandozeilenparameter verarbeiten	465
9.3.4	Die Anwendung beenden	466
9.3.5	Auswerten von Anwendungsereignissen	466
9.4	Die Window-Klasse	467
9.4.1	Position und Größe festlegen	468
9.4.2	Rahmen und Beschriftung	468
9.4.3	Das Fenster-Icon ändern	468
9.4.4	Anzeige weiterer Fenster	469
9.4.5	Transparenz	469
9.4.6	Abstand zum Inhalt festlegen	470
9.4.7	Fenster ohne Fokus anzeigen	470
9.4.8	Ereignisfolge bei Fenstern	471
9.4.9	Ein paar Worte zur Schriftdarstellung	471
9.4.10	Ein paar Worte zur Darstellung von Controls	474
9.4.11	Wird mein Fenster komplett mit WPF gerendert?	475
<b>10</b>	<b>Übersicht WPF-Controls</b>	<b>477</b>
10.1	Allgemeingültige Eigenschaften	477
10.2	Label	479
10.3	Button, RepeatButton, ToggleButton	480
10.3.1	Schaltflächen für modale Dialoge	480
10.3.2	Schaltflächen mit Grafik	482
10.4	TextBox, PasswordBox	483
10.4.1	TextBox	483
10.4.2	PasswordBox	485
10.5	CheckBox	486
10.6	RadioButton	488
10.7	ListBox, ComboBox	489
10.7.1	ListBox	489

10.7.2	ComboBox	492
10.7.3	Den Content formatieren	494
10.8	Image	496
10.8.1	Grafik per XAML zuweisen	496
10.8.2	Grafik zur Laufzeit zuweisen	496
10.8.3	Bild aus Datei laden	498
10.8.4	Die Grafiskalierung beeinflussen	499
10.9	Slider, ScrollBar	500
10.9.1	Slider	500
10.9.2	ScrollBar	502
10.10	ScrollViewer	502
10.11	Menu, ContextMenu	503
10.11.1	Menu	504
10.11.2	Tastenkürzel	505
10.11.3	Grafiken	506
10.11.4	Weitere Möglichkeiten	507
10.11.5	ContextMenu	508
10.12	ToolBar	509
10.13	StatusBar, ProgressBar	512
10.13.1	StatusBar	512
10.13.2	ProgressBar	514
10.14	Border, GroupBox, BulletDecorator	514
10.14.1	Border	515
10.14.2	GroupBox	516
10.14.3	BulletDecorator	517
10.15	Expander, TabControl	519
10.15.1	Expander	519
10.15.2	TabControl	521
10.16	Popup	522
10.17	TreeView	525
10.18	ListView	529
10.19	DataGrid	529
10.20	Calendar/DatePicker	530
10.21	Ellipse, Rectangle, Line und Co.	534
10.21.1	Ellipse	534
10.21.2	Rectangle	535
10.21.3	Line	536
<b>11</b>	<b>Wichtige WPF-Techniken</b>	<b>537</b>
11.1	Eigenschaften	537
11.1.1	Abhängige Eigenschaften (Dependency Properties)	537
11.1.2	Angehängte Eigenschaften (Attached Properties)	539

11.2	Einsatz von Ressourcen	539
11.2.1	Was sind eigentlich Ressourcen?	539
11.2.2	Wo können Ressourcen gespeichert werden?	540
11.2.3	Wie definiere ich eine Ressource?	541
11.2.4	Statische und dynamische Ressourcen	542
11.2.5	Wie werden Ressourcen adressiert?	544
11.2.6	Systemressourcen einbinden	545
11.3	Das WPF-Ereignismodell	545
11.3.1	Einführung	545
11.3.2	Routed Events	546
11.3.3	Direkte Events	549
11.4	Verwendung von Commands	549
11.4.1	Einführung zu Commands	549
11.4.2	Verwendung vordefinierter Commands	550
11.4.3	Das Ziel des Commands	552
11.4.4	Vordefinierte Commands	553
11.4.5	Commands an Ereignismethoden binden	553
11.4.6	Wie kann ich ein Command per Code auslösen?	554
11.4.7	Command-Ausführung verhindern	555
11.5	Das WPF-Style-System	555
11.5.1	Übersicht	555
11.5.2	Benannte Styles	556
11.5.3	Typ-Styles	558
11.5.4	Styles anpassen und vererben	559
11.6	Verwenden von Triggern	561
11.6.1	Eigenschaften-Trigger (Property Triggers)	562
11.6.2	Ereignis-Trigger	564
11.6.3	Daten-Trigger	565
11.7	Einsatz von Templates	566
11.7.1	Neues Template erstellen	566
11.7.2	Template abrufen und verändern	570
11.8	Transformationen, Animationen, StoryBoards	573
11.8.1	Transformationen	573
11.8.2	Animationen mit dem StoryBoard realisieren	578
<b>Teil III: Technologien</b>		<b>585</b>
<b>12</b>	<b>WPF-Datenbindung</b>	<b>587</b>
12.1	Grundprinzip	587
12.1.1	Bindungsarten	588
12.1.2	Wann eigentlich wird die Quelle aktualisiert?	590
12.1.3	Geht es auch etwas langsamer?	591
12.1.4	Bindung zur Laufzeit realisieren	592

12.2	Binden an Objekte	593
12.2.1	Objekte im XAML-Code instanziiieren	594
12.2.2	Verwenden der Instanz im C#-Quellcode	595
12.2.3	Anforderungen an die Quell-Klasse	596
12.2.4	Instanziiieren von Objekten per C#-Code	597
12.3	Binden von Collections	599
12.3.1	Anforderung an die Collection	599
12.3.2	Einfache Anzeige	600
12.3.3	Navigieren zwischen den Objekten	601
12.3.4	Einfache Anzeige in einer ListBox	603
12.3.5	DataTemplates zur Anzeigeformatierung	604
12.3.6	Mehr zu List- und ComboBox	605
12.3.7	Verwendung der ListView	607
12.4	Noch einmal zurück zu den Details	610
12.4.1	Navigieren in den Daten	610
12.4.2	Sortieren	612
12.4.3	Filtern	612
12.4.4	Live Shaping	613
12.5	Anzeige von Datenbankinhalten	615
12.5.1	Installieren der benötigten NuGet-Pakete	615
12.5.2	Anlegen der Entitätsklassen	616
12.5.3	Die Programmoberfläche	619
12.5.4	Der Zugriff auf die Daten	620
12.6	Formatieren von Werten	623
12.6.1	IValueConverter	623
12.6.2	BindingBase.StringFormat-Eigenschaft	625
12.7	Das DataGrid als Universalwerkzeug	626
12.7.1	Grundlagen der Anzeige	626
12.7.2	UI-Virtualisierung	628
12.7.3	Spalten selbst definieren	628
12.7.4	Zusatzinformationen in den Zeilen anzeigen	630
12.7.5	Vom Betrachten zum Editieren	632
12.8	Praxisbeispiel – Collections in Hintergrundthreads füllen	632
<b>13</b>	<b>.NET MAUI</b>	<b>637</b>
13.1	Einführung	637
13.2	Was kann eine .NET-MAUI-Anwendung?	639
13.3	Die erste .NET MAUI App	640
<b>14</b>	<b>Asynchrone Programmierung</b>	<b>647</b>
14.1	Übersicht	648
14.1.1	Multitasking versus Multithreading	648
14.1.2	Deadlocks	649
14.1.3	Racing	650

14.2	Programmieren mit Threads	651
14.2.1	Einführungsbeispiel	651
14.2.2	Wichtige Thread-Methoden	653
14.2.3	Wichtige Thread-Eigenschaften	655
14.2.4	Einsatz der ThreadPool-Klasse	656
14.3	Sperrmechanismen	657
14.3.1	Threading ohne lock	658
14.3.2	Threading mit lock	659
14.3.3	Die Monitor-Klasse	662
14.3.4	Mutex	666
14.3.5	Methoden für die parallele Ausführung sperren	667
14.3.6	Semaphore	668
14.4	Interaktion mit der Programmoberfläche	670
14.4.1	Die Werkzeuge	671
14.4.2	Einzelne Steuerelemente mit Invoke aktualisieren (Windows Forms)	671
14.4.3	Mehrere Steuerelemente aktualisieren	672
14.4.4	Ist ein Invoke-Aufruf nötig?	673
14.4.5	Und was ist mit WPF?	673
14.5	Timer-Threads	675
14.6	Asynchrone Programmierentwurfsmuster	677
14.6.1	Kurzübersicht	677
14.6.2	Polling	678
14.6.3	Callback verwenden	680
14.6.4	Callback mit Parameterübergabe verwenden	681
14.6.5	Callback mit Zugriff auf die Programmoberfläche	682
14.7	Es geht auch einfacher – async und await	684
14.7.1	Der Weg von synchron zu asynchron	684
14.7.2	Achtung: Fehlerquellen!	686
14.7.3	Eigene asynchrone Methoden entwickeln	689
14.8	Asynchrone Streams	691
14.8.1	Datei erstellen	691
14.8.2	Datei lesen mit <code>IAsyncEnumerable&lt;T&gt;</code>	693
14.9	Praxisbeispiele	694
14.9.1	Prozess- und Thread-Informationen gewinnen	694
14.9.2	Ein externes Programm starten	697
<b>15</b>	<b>Die Task Parallel Library</b>	<b>701</b>
15.1	Überblick	701
15.1.1	Parallel-Programmierung	701
15.1.2	Möglichkeiten der TPL	704
15.1.3	Der CLR-Threadpool	705
15.2	Parallele Verarbeitung mit <code>Parallel.Invoke</code>	706
15.2.1	Aufrufvarianten	706
15.2.2	Einschränkungen	708

15.3	Verwendung von Parallel.For	709
15.3.1	Abbrechen der Verarbeitung	710
15.3.2	Auswerten des Verarbeitungsstatus	712
15.3.3	Und was ist mit anderen Iterator-Schrittweiten?	713
15.4	Collections mit Parallel.ForEach verarbeiten	713
15.5	Die Task-Klasse	714
15.5.1	Einen Task erzeugen	714
15.5.2	Den Task starten	715
15.5.3	Datenübergabe an den Task	717
15.5.4	Wie warte ich auf das Ende des Tasks?	718
15.5.5	Tasks mit Rückgabewerten	720
15.5.6	Die Verarbeitung abbrechen	723
15.5.7	Fehlerbehandlung	728
15.5.8	Weitere Eigenschaften	729
15.6	Zugriff auf das User Interface	730
15.6.1	Task-Ende und Zugriff auf die Oberfläche	730
15.6.2	Zugriff auf das UI aus dem Task heraus	732
15.7	Weitere Datenstrukturen im Überblick	734
15.7.1	Threadsichere Collections	734
15.7.2	Primitive für die Threadsynchrisation	734
15.8	Parallel LINQ (PLINQ)	735
15.9	Praxisbeispiele	735
15.9.1	BlockingCollection	735
15.9.2	PLINQ	739
<b>16</b>	<b>Debugging, Fehlersuche und Fehlerbehandlung</b>	<b>741</b>
16.1	Der Debugger	741
16.1.1	Allgemeine Beschreibung	741
16.1.2	Die wichtigsten Fenster	742
16.1.3	Debugging-Optionen	746
16.1.4	Praktisches Debugging am Beispiel	749
16.2	Arbeiten mit Debug und Trace	754
16.2.1	Wichtige Methoden von Debug und Trace	754
16.2.2	Besonderheiten der Trace-Klasse	758
16.2.3	TraceListener-Objekte	758
16.3	Caller Information	760
16.3.1	Attribute	761
16.3.2	Anwendung	761
16.4	Fehlerbehandlung	762
16.4.1	Anweisungen zur Fehlerbehandlung	762
16.4.2	try-catch	763
16.4.3	try-finally	767
16.4.4	Das Standardverhalten bei Ausnahmen festlegen	769

16.4.5	Die Exception-Klasse .....	770
16.4.6	Fehler/Ausnahmen auslösen .....	771
16.4.7	Eigene Fehlerklassen .....	771
16.4.8	Exceptionhandling zur Debugzeit .....	773
<b>17</b>	<b>Entity Framework Core 6.0 .....</b>	<b>775</b>
17.1	Überblick .....	775
17.1.1	Objektrelationaler Mapper (ORM) .....	776
17.1.2	Provider .....	778
17.1.3	Relationale Beziehungen .....	779
17.1.4	Benötigte NuGet-Pakete .....	782
17.2	CodeFirst .....	783
17.2.1	CodeFirst aus Model .....	784
17.2.2	CodeFirst mittels ReverseEngineering von bestehender Datenbank ...	793
17.3	Migrationen .....	798
17.3.1	Initiale Migration bei ReverseEngineering .....	798
17.3.2	Weitere Migrationen .....	800
17.4	Lesen und Schreiben von Daten mit EF Core 6 .....	802
17.5	Praxisbeispiele .....	806
17.5.1	Daten mit EF Core 6 laden und als JSON speichern .....	806
17.5.2	Eine Datenbank mit EF Core 6 anlegen und Testdaten generieren und anzeigen .....	815
<b>Teil IV:</b>	<b>Webanwendungen .....</b>	<b>823</b>
<b>18</b>	<b>Webanwendungen mit ASP.NET Core .....</b>	<b>825</b>
18.1	Grundlagen .....	826
18.2	Razor Pages .....	829
18.2.1	Projektaufbau .....	829
18.2.2	Razor-Syntax .....	833
18.2.3	Layout-Vorlagen .....	838
18.2.4	Modelle für Razor Pages .....	842
18.2.5	Mit Formularen arbeiten .....	843
18.3	MVC .....	850
18.3.1	Projektaufbau .....	851
18.3.2	Action-Methoden .....	852
18.3.3	Zustandsmanagement .....	864
18.4	Praxisbeispiele .....	869
18.4.1	CRUD mit Entity Framework .....	869
18.4.2	Authentifizierung und Autorisierung .....	885
<b>19</b>	<b>ASP.NET Web API .....</b>	<b>893</b>
19.1	REST .....	893
19.2	Vorlagen .....	897

19.3	Daten lesen .....	903
19.4	Daten schreiben, aktualisieren, löschen .....	912
19.5	Minimale APIs .....	921
19.6	Praxisbeispiele .....	925
19.6.1	Paginierung .....	925
19.6.2	XML statt JSON .....	927
19.6.3	CORS .....	928
<b>20</b>	<b>Blazor .....</b>	<b>933</b>
20.1	Hosting-Modelle .....	934
20.2	Projektvorlagen .....	937
20.3	Blazor-Komponenten .....	946
20.3.1	Code in/für Komponenten .....	946
20.3.2	Event-Handling .....	949
20.3.3	Datenbindung .....	953
20.4	Services und APIs aufrufen .....	955
20.5	Weitere Blazor-Features .....	961
20.5.1	Zustandsmanagement .....	961
20.5.2	JavaScript-Interoperabilität .....	963
20.6	Praxisbeispiele .....	967
20.6.1	Online-Status ermitteln .....	967
20.6.2	File-Uploads .....	972
20.6.3	Fehlerbehandlung .....	974
<b>Anhang A: Glossar .....</b>		<b>981</b>
<b>Anhang B: Wichtige Dateieindungen .....</b>		<b>985</b>
<b>Index .....</b>		<b>987</b>

# 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.

Mit .NET 6 hat Microsoft jetzt auch die neueste Version von .NET veröffentlicht, die endlich die bereits für .NET 5 geplante, aber aufgrund der Covid-Pandemie verschobene Vereinheitlichung *One .NET* sowie weitere neue Anwendungsarten (z. B. *.NET MAUI*, auch wenn das jetzt auf das 2. Quartal 2022 verschoben wurde) und weitere Verbesserungen, vor allem im Performancebereich, für die C#-Entwickler enthält.

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

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

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

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

## Zum Buchinhalt

Dieses Buch wagt den Spagat zwischen einem Grundlagen- und einem Profibuch. Sinn eines solchen Buches kann es nicht sein, eine umfassende Schritt-für-Schritt-Einführung in C# 10.0 zu liefern oder all die Informationen noch einmal zur Verfügung zu stellen, die Sie in der Produktdokumentation ohnehin schon finden und von denen Sie in der Regel nur ein Mausklick oder die F1-Taste trennt.

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

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

**1. Grundlagen der Programmierung mit C# (Buch)**

**2. Desktop-Anwendungen (Buch)**

**3. Technologien (Buch)**

**4. Webtechnologien (Buch)**

5. Windows-Forms-Anwendungen (online)

6. Zugriff auf das Dateisystem (online)

7. ADO.NET (online)

8. XML (online)

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

## Updates und Aktualität

Damit Sie möglichst lange mit diesem Buch arbeiten können, haben Sie die Möglichkeit, sich für den kostenlosen Update inside-Service zu registrieren: Geben Sie unter

*[www.hanserfachbuch.de/Csharp-update](http://www.hanserfachbuch.de/Csharp-update)*

diesen Code ein:

`plus-12abc-8xyz9`

Dann erhalten Sie bis März 2024 Aktualisierungen in Form zusätzlicher Kapitel als PDF. Darin stellen wir Ihnen wichtige Neuerungen vor und gehen auf Änderungen ein, die die Inhalte dieses Buches betreffen.

## Nobody is perfect

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

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

*[juergen.kotz@primetime-software.de](mailto:juergen.kotz@primetime-software.de)*

*[info@christianwenz.de](mailto:info@christianwenz.de)*

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

### **Vielen Dank!**

Ein Buch – gerade, wenn es so umfangreich ist – ist stets Teamarbeit. Herzlichen Dank an unsere Lektorin Sylvia Hasselbach, mit der wir seit über 20 Jahren zusammenarbeiten dürfen, sowie an Kristin Rothe und Irene Weilhart. Walter Saumweber hat als Fachlektor seit mehreren Auflagen den Röntgenblick für Ungenauigkeiten und Fehler (sollten wir trotzdem auf weitere Errata aufmerksam gemacht werden, veröffentlichen wir diese auf der Buchseite auf [www.hanserfachbuch.de](http://www.hanserfachbuch.de) und auf <https://plus.hanserfachbuch.de> bei den Codebeispielen). Lenny Kotz hat uns bei einigen der Grafiken im Buch unterstützt.

*Jürgen Kotz und Christian Wenz*

*München, im Januar 2022*

## ■ **Zusatzmaterial online**

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

Geben Sie auf

<https://plus.hanserfachbuch.de>

diesen Code ein:

```
plus-12abc-8xyz9
```

Beim Nachvollziehen der Buchbeispiele beachten Sie bitte Folgendes:

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



# 11

## Wichtige WPF-Techniken

Nachdem wir im vorhergehenden Kapitel einige praktische Erfahrungen mit den WPF-Controls gesammelt haben, wollen wir uns jetzt mit den Besonderheiten der WPF-Programmierung im Vergleich zur Windows-Forms-/Win32-Programmierung beschäftigen.

Damit Sie die WPF-Philosophie verstehen, wollen wir Ihnen zu Beginn einige wichtige Eckpfeiler dieser Technologie erklären.

### ■ 11.1 Eigenschaften

Da die Definition und Verwendung von Eigenschaften in WPF etwas anders organisiert ist als in den bekannten Windows-Forms-Anwendungen, sollten wir zunächst auf dieses Thema näher eingehen.

#### 11.1.1 Abhängige Eigenschaften (Dependency Properties)

Mit den abhängigen (Dependency) und angehängten (Attached) Eigenschaften erweitert WPF das Spektrum der CLR-Eigenschaften. Abhängige Eigenschaften bieten gegenüber den „normalen“ Eigenschaften folgende erweiterte Möglichkeiten:

- eine interne Prüfung (Validierung),
- automatisches Aktualisieren von Werten,
- die Verwendung von Callback-Methoden zur Signalisierung von Wertänderungen
- sowie die Vorgabe von Defaultwerten.

Notwendig wurde diese Erweiterung des Eigenschaftensystems, um viele der WPF-Features (Animationen, Datenbindung, Styles etc.) zu realisieren. So werden beispielsweise Werte von Eigenschaften überwacht und Änderungen führen automatisch zu Änderungen an den abhängigen Objekten.

Abhängige Eigenschaften werden nicht als private Felder definiert, sondern als statische, öffentliche Instanzen der Klasse *System.Windows.DependencyProperty*, die über *get*- und *set*-Methoden angesprochen werden. Die Verwaltung der Eigenschaft wird vom WPF-Subsys-

tem übernommen (daher auch die *Register*-Methode, siehe folgendes Beispiel 11.1). Die der Eigenschaft übergeordnete Klasse stellt quasi nur noch eine Schnittstelle zu dieser Eigenschaft zur Verfügung.

Neben dem Wert können mit dem Defaultwert und der Callback-Methode auch weitere Metadaten zu einer Eigenschaft gespeichert werden.

**Beispiel 11.1:** Definition einer abhängigen Eigenschaft *Durchmesser* für ein Objekt *Kreis*

**C#**

```
using System.Windows;
```

```
namespace Dependency_Attached_Properties;
```

Wir definieren eine neue Klasse *Kreis* und leiten diese gleich von *DependencyObject* ab:

```
public class Kreis : DependencyObject
{
```

Hier die eigentliche Definition der abhängigen Eigenschaft (übergeben werden der Name, der Datentyp, das abhängige Objekt und optional die Metadaten, d. h. der Defaultwert und eine Callback-Methode):

```
    public static readonly DependencyProperty DurchmesserProperty =
        DependencyProperty.Register(nameof(Durchmesser), typeof(double),
            typeof(Kreis), new FrameworkPropertyMetadata(11.1,
                new PropertyChangedCallback(OnDurchmesserChanged)));
```

Wie Sie sehen, handelt es sich nicht mehr um ein privates Feld. Vielmehr wird die bisher übliche Kapselung aufgegeben, die Verwaltung des Zustands wird von WPF übernommen, die Instanz meldet seine „lokalen Speicher“ nur noch an (*Register*).

Die folgende Definition ist nur noch die allgemeine Schnittstelle nach außen, wie Sie es auch von den normalen Eigenschaften kennen:

```
    public double Durchmesser
    {
        get
        {
            return (double)GetValue(DurchmesserProperty);
        }
        set
        {
            SetValue(DurchmesserProperty, value);
        }
    }
}
```

Hier eine Callback-Methode, mit der eine Wertänderung überwacht werden kann:

```
    private static void OnDurchmesserChanged(DependencyObject obj,
        DependencyPropertyChangedEventArgs args)
    {
        MessageBox.Show((obj as Kreis)?.Durchmesser.ToString());
    }
}
```

### 11.1.2 Angehängte Eigenschaften (Attached Properties)

Mit den angehängten Eigenschaften (*Attached Properties*), einer speziellen Form der *Dependency Properties*, wird der Versuch unternommen, die Flut von WPF-Element-Eigenschaften etwas einzudämmen. Das Problem: Wird ein Element/Control in einem Container platziert, hängt beispielsweise dessen Position vom umgebenden Container (*Grid*, *Canvas*, *DockPanel* etc.) ab. Für jede Art von Container werden aber andere Eigenschaften zur Positionierung benötigt. Aus diesem Grund stellen die übergeordneten Elemente die zum Positionieren nötigen Eigenschaften zur Verfügung (*Canvas.Top*, *Canvas.Left*, *DockPanel.Dock*, *Grid.Column*, ...), das eingelagerte Element nutzt diese lediglich in seinem Kontext. Der Vorteil: Nur wenn sich beispielsweise ein *Button* in einem *Canvas* befindet, werden auch die Eigenschaften *Canvas.Top*, *Canvas.Left* bereitgestellt und verwendet.

**Beispiel 11.2:** Positionieren einer Schaltfläche in einem *Canvas*-Control mit *Attached Properties*

#### XAML

```
<Canvas>
  <Button Canvas.Left="74" Canvas.Top="70" Height="45" Width="89">
    Beschriftung
  </Button>
</Canvas>
```

## ■ 11.2 Einsatz von Ressourcen

Bevor wird uns im Weiteren mit *Styles* etc. beschäftigen, müssen wir zunächst noch einen Blick auf die Ressourcenverwaltung in WPF-Anwendungen werfen, da diese die Voraussetzung für die Zuordnung darstellt.

### 11.2.1 Was sind eigentlich Ressourcen?

In WPF-Anwendungen zählen nicht nur Grafiken, Strings, Sprachinformationen oder beliebige binäre Informationen zu den Ressourcen, sondern auch die Beschreibung von *Styles*, Füllmustern oder sogar ganzer Controls.

Eine Ressource besteht immer aus einem eindeutigen Schlüssel (*Key*) und dem eigentlichen Content, der jederzeit austauschbar ist, da Bezüge auf die Ressource immer nur den Schlüssel verwenden.

## 11.2.2 Wo können Ressourcen gespeichert werden?

Eine Möglichkeit, Ressourcen in Ihrer Anwendung abzulegen, haben Sie sicher ganz unbewusst schon zur Kenntnis genommen. Die Rede ist von der Datei *App.xaml*, in der bereits ein *Resources*-Abschnitt vordefiniert ist:

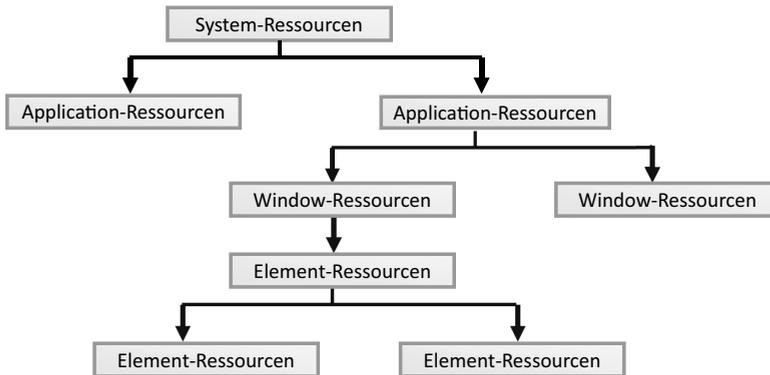
```
<Application x:Class="RessourcenSample.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="clr-namespace:RessourcenSample"
  StartupUri="MainWindow.xaml">
  <Application.Resources>

  </Application.Resources>
</Application>
```

Hierbei handelt es sich um anwendungsweit verfügbare Ressourcen, die allen Windows/Pages und deren Elementen zur Verfügung stehen.

Neben diesen Ressourcen können Sie auch jedem Element und jedem Window eigene Ressourcen zuordnen, zusätzlich sind auch sogenannte Systemressourcen (z. B. Systemfarben) verfügbar.

Bild 11.1 zeigt die mögliche Hierarchie von Ressourcen.



**Bild 11.1** Hierarchie von Ressourcen



**HINWEIS:** Wird eine Ressource referenziert und damit auch gesucht, beginnt die Suche immer beim aktuellen Element. Wird die Ressource hier nicht gefunden, wird im übergeordneten Element (Container → Window → Application → System) gesucht. Damit ist auch klar, dass untergeordnete Elemente gleichnamige Ressourcen von übergeordneten Elementen überschreiben können. Innerhalb einer Hierarchieebene ist dies nicht möglich, da es sich in diesem Fall nicht um einen eindeutigen Schlüssel handelt.

Im XAML-Quellcode stellt sich die Definition von *Resources*-Abschnitten wie folgt dar:

Die Window-Ressourcen:

```
<Window>
  <Window.Resources>
    ...
  </Window.Resources>

  <StackPanel>
```

Die Ressourcen eines Containers:

```
  <StackPanel.Resources>
    ...
  </StackPanel.Resources>
  ...
  <Button>
```

Die Ressourcen eines Elements:

```
    <Button.Resources>
      ...
    </Button.Resources>
  </Button>
</StackPanel>
</Window>
```

### 11.2.3 Wie definiere ich eine Ressource?

Haben Sie sich dafür entschieden, auf welcher Ebene Sie die Ressource definieren, können Sie zur Tat schreiten:

- Zunächst müssen Sie sich über die Art der Ressource im Klaren sein; diese bestimmt den Elementnamen (z. B. ein *ImageBrush*, mit dem ein Hintergrund festgelegt werden kann).
- Neben dieser Information müssen Sie sich noch für einen eindeutigen Schlüssel entscheiden, über den die Ressource angesprochen werden soll.
- Last, but not least, müssen Sie auch noch die eigentlichen Informationen zur Ressource (beim *ImageBrush* ist dies die *ImageSource*) festlegen.

**Beispiel 11.3:** Erzeugen und Verwenden einer Ressource auf Fensterebene

#### XAML

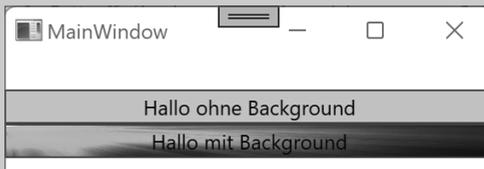
```
<Window x:Class="RessourcenSample.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:local="clr-namespace:RessourcenSample"
  mc:Ignorable="d"
  Title="MainWindow" Height="250" Width="400">
```

```
<Window.Resources>
  <ImageBrush x:Key="bck" ImageSource="back.jpg" />
</Window.Resources>
```

Die Verwendung:

```
<StackPanel Margin="0, 20, 0, 0">
  <Button>Hallo ohne Background</Button>
  <Button Background="{StaticResource bck}">Hallo mit Background</Button>
</StackPanel>
</Window>
```

### Ergebnis



**Bild 11.2**  
Button mit Ressource als Hintergrund



**HINWEIS:** Beim Einsatz von (statischen) Ressourcen ist es wichtig, dass diese **vor** der Verwendung definiert wurden, andernfalls kann die Ressource nicht gefunden werden. Zusätzlich müssen Sie auf die Schreibweise achten, hier wird die Groß-/Kleinschreibung berücksichtigt!

Alternativ können Sie bei der Verwendung der Ressource auch die folgende Syntax nutzen:

```
<Button Height="50">
  <Button.Background>
    <StaticResource ResourceKey="bck" />
  </Button.Background>
  Hallo mit Background2
</Button>
```

## 11.2.4 Statische und dynamische Ressourcen

Wie Sie im vorhergehenden Beispiel bereits gesehen haben, können Ressourcen statisch, d. h. unveränderlich, über den Schlüssel zugeordnet werden:

```
<StackPanel>
  <Button Background="{StaticResource bck}">Hallo mit Background</Button>
</StackPanel>
```

Voraussetzung ist das vorherige Definieren der Ressource. Spätere Änderungen an der Ressource werden nicht registriert und haben keine Auswirkung auf das verwendende Element.

Neben dieser Variante besteht auch die Möglichkeit, Ressourcen dynamisch festzulegen. In diesem Fall können Sie die Ressourcen zur Laufzeit zuordnen bzw. bereits vorhandene Ressourcen einfach austauschen.

**Beispiel 11.4:** Verwendung einer noch nicht definierten dynamischen Ressource.

#### XAML

Zunächst die Zuordnung von Eigenschaft und Ressource:

```
<Button Background="{DynamicResource bck2}"
        Click="Button_Click">Hallo</Button>
```

Da es die Ressource zu diesem Zeitpunkt noch nicht gibt, erscheint eine Warnung, die Sie aber ignorieren können.

#### C#

Mit dem Klick auf die Schaltfläche wollen wir eine Ressource zuordnen:

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

Neuen *ImageBrush* erzeugen:

```
    ImageBrush brush = new ImageBrush();
```

Eine Bitmap zuweisen:

```
    brush.ImageSource = new BitmapImage(new
        Uri("pack://application:,,,/Images/back2.jpg"));
```

Die Ressource erzeugen (Window-Ressource):

```
        Resources.Add("bck2", brush);
    }
```

#### Ergebnis



**Bild 11.3**

Nach Klick hat auch der obere Button eine Ressource zugewiesen.



**HINWEIS:** Und warum verwenden wir nicht einfach immer dynamische Ressourcen? Da die Verwaltung dynamischer Ressourcen deutlich aufwendiger ist, würden hier unnötigerweise Systemressourcen verschwendet werden.

### 11.2.5 Wie werden Ressourcen adressiert?

Möchten Sie auf eingebundene Ressourcen (z.B. Grafiken) Ihrer Anwendung zugreifen, müssen Sie beispielsweise bei der Zuordnung von Grafiken per Code eine bestimmte Syntax einhalten, andernfalls wird die Ressource an der falschen Stelle gesucht und dann wohl auch nicht gefunden.

Die dazu erforderliche URI können Sie absolut, d.h. mit voller Pfadangabe, inklusive der aktuellen Assembly oder relativ zur aktuellen Assembly angeben.

**Beispiel 11.5:** Absolute Pfadangabe (Bild liegt in der Projekt-Root, Buildvorgang=*Resource*)

C#

```
Uri uri = new Uri("pack://application:,,,/Bild.jpg");
```

**Beispiel 11.6:** Absolute Pfadangabe (das Bild liegt im Unterverzeichnis *Images* des aktuellen Projekts, Buildvorgang=*Resource*)

C#

```
Uri uri = new Uri("pack://application:,,,/Images/Bild.jpg");
```

**Beispiel 11.7:** Relative Pfadangabe (das Bild liegt im gleichen Verzeichnis wie die Assembly)

C#

```
Uri uri = new Uri(@"\Back3.jpg", UriKind.Relative);
```

**Beispiel 11.8:** Relative Pfadangabe (das Bild liegt im relativen Unterverzeichnis *Images* der Assembly)

C#

```
Uri uri = new Uri(@"\Images\Back3.jpg", UriKind.Relative)
```



**HINWEIS:** Vielleicht ist Ihnen aufgefallen, dass in WPF noch immer viele *using*-Anweisungen zum Import von Namespaces gebraucht werden. Dies liegt daran, dass die implizite globale Verwendung in WPF-Projekten nicht standardmäßig aktiviert ist. Fügen Sie einfach `<ImplicitUsings>enable</ImplicitUsings>` in die Projektdatei ein.

Neben den gezeigten Möglichkeiten können Sie unter anderem auch auf eingebundene Assemblies zugreifen.

## 11.2.6 Systemressourcen einbinden

Neben den in Ihrer Anwendung definierten Ressourcen können Sie auch Systemressourcen verwenden. Die Einbindung erfolgt entweder statisch oder dynamisch. Im ersten Fall reagiert die Anwendung jedoch nicht auf aktuelle Änderungen an den Systemeinstellungen.

**Beispiel 11.9:** Einbinden von Systemressourcen

### XAML

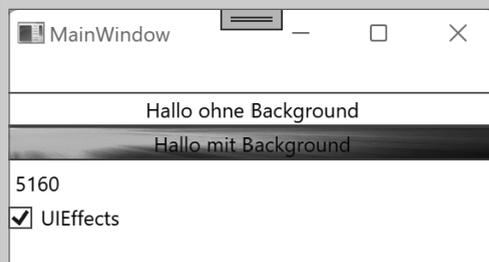
Anzeige der *VirtualScreenWidth* in einem *Label* (Ressourcenabfrage):

```
<Label Content="{StaticResource {x:Static
    SystemParameters.VirtualScreenWidthKey}}"/>
```

Anzeige, ob User-Interface-Effekte aktiviert sind (Wertabfrage):

```
<CheckBox IsChecked="{x:Static SystemParameters.UIEffects}"
    Content="UIEffects" />
```

### Ergebnis



**Bild 11.4**

Anzeige von Systemressourcen

## 11.3 Das WPF-Ereignismodell

Nachdem wir in den bisherigen Beispielen recht sparsam mit der Verwendung von Event-Handlern umgegangen sind, wollen wir uns jetzt diesem Thema etwas intensiver widmen.

### 11.3.1 Einführung

Sicher haben Sie auch schon mehr oder weniger unbewusst von den Ereignissen in WPF Gebrauch gemacht. Nach dem Klick, z.B. auf eine Schaltfläche, wird der entsprechende Ereignishandler von Visual Studio bereitgestellt.

**Beispiel 11.10:** *Button* mit zugehöriger Ereignisbehandlung in C#.

#### XAML

```
<StackPanel>
  <Button Content="Klick mich!" Click="Button_Click"/>
</StackPanel>
```

#### C#

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Hallo");
}
```

So weit, so gut – das kennen Sie sicher auch schon von der Programmierung mit Win32 oder Windows Forms. Doch was passiert, wenn Sie in WPF eine Schaltfläche aus einzelnen Elementen „zusammenbasteln“?

**Beispiel 11.11:** *Button* mit Text und Grafik

#### XAML

```
<Button Click="Button_Click">
  <StackPanel Orientation="Horizontal" Margin="10">
    <Image Source="Images/Flash.png" Width="56" Height="46" />
    <TextBlock VerticalAlignment="Center">Klick mich!</TextBlock>
  </StackPanel>
</Button>
```

#### Ergebnis



**Bild 11.5**

Button mit eingebettetem *Image* und *TextBlock*

Rein intuitiv haben Sie sicher auch das *Click*-Ereignis dem *Button* zugeordnet. Doch warum sollte das eigentlich funktionieren? Was passiert, wenn der Nutzer auf das *Image* oder den *TextBlock* klickt, zusätzlich befindet sich „darunter“ ja noch das *StackPanel*.

Das Zauberwort heißt hier „Routed Events“, ein Verfahren, um auftretende Ereignisse in der Element-Hierarchie weiterzugeben.

### 11.3.2 Routed Events

WPF unterscheidet bei den Routed Events zwei verschiedene Varianten, die Sie als Programmierer auch auseinanderhalten sollten:

- **Tunneling Events:** Ausgehend vom Wurzelement (*Window/Page*) werden die Ereignisse bis zum auslösenden Element weitergereicht. Diese Events werden **vor** den zugehörigen Bubbling Events ausgelöst.

- **Bubbling Events:** Ausgehend vom aktivierten Element werden die Ereignisse zum jeweils übergeordneten Element weitergereicht, d. h. im Endeffekt bis zum *Window* oder zur *Page*.



**HINWEIS:** Tunneling Events sind durch das einleitende „Preview...“ gekennzeichnet, Bubbling Events verzichten darauf.

**Beispiel 11.12:** Ablauf der Ereigniskette für einen Mausklick mit der linken Taste

#### XAML

```
<Window x:Class="EreignisseSample.MainWindow"
...
  Title="MainWindow" MouseLeftButtonDown="Window_MouseLeftButtonDown"
    PreviewMouseLeftButtonDown="Window_PreviewMouseLeftButtonDown">
  <StackPanel>
    <Button MouseLeftButtonDown="Button_MouseLeftButtonDown"
      PreviewMouseLeftButtonDown="Button_PreviewMouseLeftButtonDown">
      <StackPanel Orientation="Horizontal" Margin="10"
        MouseLeftButtonDown="StackPanel_MouseLeftButtonDown"
        PreviewMouseLeftButtonDown="StackPanel_PreviewMouseLeftButtonDown">
        <Image Source="Images/Flash.png" Width="56" Height="46"
          MouseLeftButtonDown="Image_MouseLeftButtonDown"
          PreviewMouseLeftButtonDown="Image_PreviewMouseLeftButtonDown" />
        <TextBlock VerticalAlignment="Center"
          MouseLeftButtonDown="TextBlock_MouseLeftButtonDown"
          PreviewMouseLeftButtonDown="TextBlock_PreviewMouseLeftButtonDown">
          Klick mich!
        </TextBlock>
      </StackPanel>
    </Button>
    <ListBox Name="listBox1" />
  </StackPanel>
</Window>
```

#### Ergebnis

Die Ereigniskette nach einem Klick auf das *Image*:

1. Window: *PreviewMouseLeftButtonDown*
2. Button: *PreviewMouseLeftButtonDown*
3. StackPanel: *PreviewMouseLeftButtonDown*
4. Image: *PreviewMouseLeftButtonDown*
5. Image: *MouseLeftButtonDown*
6. StackPanel: *MouseLeftButtonDown*

Wie Sie sehen, wird zunächst die komplette Tunneling-Ereigniskette durchlaufen, nachfolgend die Bubbling-Events<sup>1</sup>.

Im Normalfall werden Sie wohl nur Bubbling-Events verwenden. Tunneling-Events nutzen Sie beispielsweise, um Ereignisse bzw. deren Weiterleitung zu blockieren.

<sup>1</sup> Der Button löst ein *Click*-Ereignis aus. Die dazu nötige Logik verhindert das Auslösen entsprechender *MouseLeftButtonDown*-Ereignisse, deshalb ist hier die Ereigniskette zu Ende.

**Beispiel 11.13:** Das Weiterleiten der Ereignisse verhindern.

**C#**

Wir werten gleich das erste Ereignis aus (das Tunneling-Event beginnt beim Wurzelobjekt, d. h. dem Window):

```
private void Window_PreviewMouseLeftButtonDown(object sender,
        MouseButtonEventArgs e)
{
```

Ereignis behandelt

```
    e.Handled = true;
    listBox1.Items.Add(nameof(Window_PreviewMouseLeftButtonDown));
}
```

**Beispiel 11.14:** Das auslösende Element bestimmen

**C#**

Den Ursprung für ein Ereignis können Sie mit dem ...*EventArgs.Source*-Parameter bestimmen:

```
private void Window_PreviewMouseLeftButtonDown(object sender,
        MouseButtonEventArgs e)
{
    MessageBox.Show(e.Source.ToString());
    listBox1.Items.Add(nameof(Window_PreviewMouseLeftButtonDown));
}
```

**Ergebnis**

Beim Klick auf das Image wird auch dieses als *Source* übermittelt.



**Bild 11.6**  
Klick auf das Image

### 11.3.3 Direkte Events

Auch diese Form der Ereignisse ist nach wie vor präsent. Hierbei handelt es sich um die ganz normalen .NET-Ereignisse, wie Sie auch in Windows-Forms-Anwendungen auftreten.

Direkte Events werden eingesetzt, wenn die Verwendung von Bubbling oder Tunneling keinen Sinn macht, beispielsweise beim *MouseLeave*-Ereignis, das sehr objektbezogen ausgelöst wird.

Sie erkennen diese Ereignisse an einem fehlenden *Preview...*-Pendant.

## ■ 11.4 Verwendung von Commands

Im Zusammenhang mit der Entwicklung von Anwendungen ist es Ihnen sicher auch schon passiert, dass Sie eine Menüfunktion zum x-ten Male neu programmiert haben. Das Gleiche trifft sicher ebenfalls auf die Toolbar zu. Das Prozedere ist doch immer gleich:

1. Methode mit der eigentlichen Logik erstellen
2. Menüpunkt erstellen
3. Menüpunkt Tastenkürzel zuweisen, Tastaturabfrage implementieren
4. Menüpunkt Ereignismethode zuweisen und Methode von 1. aufrufen
5. Toolbar-Button bereitstellen
6. Toolbar-Button Ereignismethode zuweisen und Methode von 1. aufrufen
7. Logik für das Sperren von 2. und 5. erstellen, wenn die Funktion nicht zur Verfügung steht

### 11.4.1 Einführung zu Commands

In WPF-Anwendungen können Sie diesen Aufwand wesentlich verringern, indem Sie entweder die von WPF vordefinierten Commands verwenden, oder indem Sie selbst eigene Commands implementieren.

Die neue Vorgehensweise (vordefinierte Commands, z. B. Einfügen in die Zwischenablage) im Vergleich zum bisherigen Vorgehen:

1. Entfällt, da für viele Controls bereits implementiert
2. **Menüpunkt erstellen und Command zuweisen**
3. Entfällt, da per Command automatisch zugewiesen
4. Entfällt, da per Command automatisch zugewiesen
5. **Toolbar-Button bereitstellen und Command zuweisen**
6. Entfällt, da per Command automatisch zugewiesen
7. Entfällt, da Command diese Logik bereitstellt

Das sieht doch schon wesentlich freundlicher aus als die mühsame und fehleranfällige erste Variante. Ähnlich einfach gestaltet sich die Wiederverwendung von selbst erstellten Kommandos, wenn Sie diese bereits einmal erstellt haben.

## 11.4.2 Verwendung vordefinierter Commands

Statt vieler Worte möchten wir Ihnen zunächst an einem Beispiel die Vorgehensweise bei der Verwendung von Commands demonstrieren.

**Beispiel 11.15:** Programmieren der Funktionen „Kopieren und Einfügen“ für ein Formular mit zwei TextBoxen

### XAML

Der XAML-Code der Oberfläche:

```
<Window x:Class="CommandsSample.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:CommandsSample"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="400">
  <DockPanel>
```

Die Menüdefinition:

```
<Menu DockPanel.Dock="Top" Height="22" Name="menu1">
  <MenuItem Header="_Bearbeiten">
```

Hier werden die beiden Menüpunkte „Kopieren“ und „Einfügen“ definiert:

```
<MenuItem Command="ApplicationCommands.Copy"/>
<MenuItem Command="ApplicationCommands.Paste"/>
```

Sie können auf die Angabe des Headers sowie der Tastenkürzel verzichten, diese stellt die obige *Command*-Definition automatisch zur Verfügung.

```
</MenuItem>
</Menu>
<ToolBarTray DockPanel.Dock="Top">
  <ToolBar>
```

Hier findet sich bereits die Definition für die *ToolBar*-Buttons. Auch in diesem Fall genügt die Zuordnung der *Command*-Eigenschaft:

```
<Button Width="30" Height="30" Command="ApplicationCommands.Copy">
  <Image Source="Images/editcopy.png"/>
</Button>
<Button Width="30" Height="30" Command="ApplicationCommands.Paste">
  <Image Source="Images/editpaste.png"/>
```

```

        </Button>
    </ToolBar>
</ToolBarTray>
<StackPanel>

```

Hier noch die beiden *TextBox*en, für die die Funktionen implementiert werden:

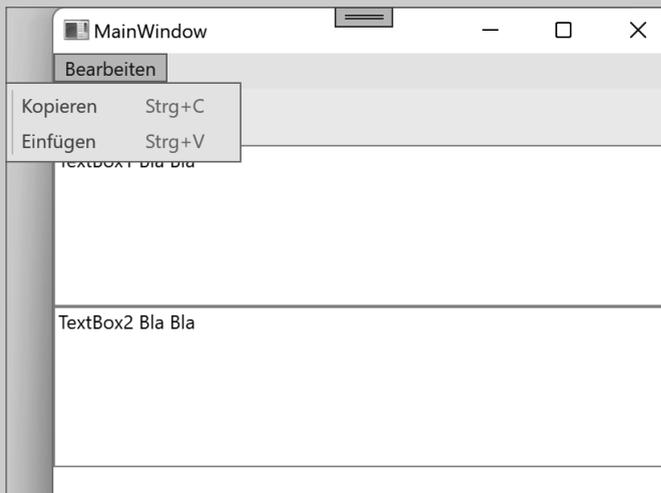
```

        <TextBox Name="txt1" Height="100">TextBox1 Bla Bla</TextBox>
        <TextBox Name="txt2" Height="100">TextBox2 Bla Bla</TextBox>
    </StackPanel>
</DockPanel>
</Window>

```

### Ergebnis

Und wo bleibt der Code? Kurze Antwort: Ihr Programm ist an dieser Stelle bereits fertig. Nach dem Start können Sie sich von der Funktionstüchtigkeit überzeugen.



**Bild 11.7** Fertige Menübefehle dank Commands

Beide Menüpunkte verfügen über eine Beschriftung und ein Tastenkürzel. Befindet sich Text in der Zwischenablage, ist das Menü *Einfügen* aktiviert, andernfalls ist der Menüpunkt gesperrt. Der Menüpunkt *Kopieren* ist nur aktiv, wenn Sie in einer der *TextBox*en Text markieren und die *TextBox* den Eingabefokus besitzt.

Nicht schlecht, wenn Sie dies mit der konventionellen Variante vergleichen, bei der Sie sicher wesentlich mehr Code produziert hätten.<sup>2</sup>

<sup>2</sup> Geben Sie trotzdem einen Wert an, überschreibt dies die vom Command bereitgestellten Werte.

### 11.4.3 Das Ziel des Commands

Doch gerade bei obigem Beispiel wird sicher bei manchem die Frage aufkommen, wohin denn eigentlich der Text eingefügt wird, haben wir doch zwei Textfelder. Die Frage ist sicher berechtigt, aber im obigen Beispiel noch recht einfach lösbar: Ziel des jeweils gewählten Commands ist standardmäßig immer das gerade aktive Control, d. h. die *TextBox*, die den Eingabefokus besitzt.

Schwieriger wird es, wenn die Zwischenablageinhalte gezielt in ein bestimmtes Control eingefügt werden sollen. In diesem Fall müssen Sie neben der *Command*- auch die *CommandTarget*-Eigenschaft definieren. Allerdings genügt in diesem Fall nicht die reine Angabe des Elementnamens, sondern Sie müssen die Binding-Syntax verwenden.

**Beispiel 11.16:** Zwischenablageinhalte sollen immer in *TextBox2* eingefügt werden

#### XAML

Wir fügen dem betreffenden Menüpunkt ein *CommandTarget* hinzu:

```
<MenuItem Header="_Bearbeiten">
  <MenuItem Command="ApplicationCommands.Copy"/>
  <MenuItem Command="ApplicationCommands.Paste"
    CommandTarget="{Binding ElementName=txt2}"/>
</MenuItem>
...
```

#### Ergebnis

Starten Sie jetzt das Programm, ist es egal, welches Control den Fokus besitzt. Drücken Sie die Tastenkombination **Strg+V** oder wählen Sie den entsprechenden Menüpunkt, wird der Text immer in die zweite *TextBox* eingefügt.



**HINWEIS:** Diese Vorgehensweise müssen Sie auch wählen, wenn Sie Commands von einzelnen Schaltflächen aus starten wollen. Da diese den Fokus erhalten können, würde nie klar sein, welches Ziel die Aktion haben soll.

**Beispiel 11.17:** „Freistehender“ *Button* für das Einfügen des Zwischenablageinhalts

#### XAML

```
<Button Command="ApplicationCommands.Paste"
  CommandTarget="{Binding ElementName=txt2}">
  Paste (TextBox 2)
</Button>
```

### 11.4.4 Vordefinierte Commands

WPF bietet bereits „ab Werk“ eine ganze Reihe von Commands, die in der täglichen Programmierpraxis immer wieder anfallen. Diese Commands gliedern sich in die folgenden Gruppen:

- *ApplicationCommands* (z. B. *Copy*, *Cut*, *Paste*, *Help*, *New*, *Print*, *Save*, *Stop*, *Undo*)
- *ComponentCommands* (z. B. *ExtendSelectionLeft*, *MoveLeft*)
- *NavigationCommands* (z. B. *FirstPage*, *GotoPage*, *Refresh*, *Search*)
- *MediaCommands* (z. B. *Play*, *Pause*, *FastForward*, *IncreaseVolume*)
- *EditingCommands* (z. B. *Delete*, *MoveUpByLine*, *ToggleBold*)

Ob und, wenn ja, welche Commands implementiert sind, müssen Sie von Fall zu Fall ausprobieren. Sehr umfassend ist z. B. die Unterstützung bei *TextBox* und *RichTextBox* sowie beim *MediaElement* (siehe Onlinekapitel).

### 11.4.5 Commands an Ereignismethoden binden

Wie schon erwähnt, implementiert nicht jedes Control alle verfügbaren Commands. So steht zwar ein *ApplicationCommand.Open* (d. h. Beschriftung und Tastaturkürzel) zur Verfügung, im Normalfall passiert allerdings nichts, da kein Control eine entsprechende Logik implementiert hat, was bei derart komplexen Abläufen sicher auch nicht möglich ist.

Aus diesem Grund besteht die Möglichkeit, einem Command eine entsprechende Ereignisbehandlung per *CommandBinding* zuzuordnen. Zwei Ereignisse sind hier von zentraler Bedeutung:

- *Execute* (die eigentlich auszuführende Logik) und
- *CanExecute* (eine Abfrage, ob die Funktion überhaupt zur Verfügung steht)

**Beispiel 11.18:** Implementieren des *ApplicationCommand.Open*

#### XAML

Nutzen Sie für die Zuordnung der beiden Ereignismethoden am besten das Wurzelement (*Window*):

```
<Window x:Class="CommandSample.MainWindow"
...
    Title="MainWindow" Height="350" Width="400">
    <Window.CommandBindings>
        <CommandBinding Command="ApplicationCommands.Open"
                        Executed="OpenCmdExecuted"
                        CanExecute="OpenCmdCanExecute"/>
    </Window.CommandBindings>
```

Vergessen Sie nicht, noch einen zusätzlichen Menüeintrag zu erstellen:

```
<MenuItem Header="_Datei">
    <MenuItem Command="ApplicationCommands.Open"/>
</MenuItem>
```

**C#**

Die beiden zugehörigen Ereignismethoden aus der Klassendefinition des Windows:

```
void OpenCmdExecuted(object target, ExecutedRoutedEventArgs e)
{
    MessageBox.Show("Was soll ich öffnen?");
    // hier steht die eigentliche Logik
}
```

Nur wenn die erste *TextBox* auch leer ist, ist das Öffnen neuer Dateien möglich:

```
void OpenCmdCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    if (string.IsNullOrEmpty(txt1.Text))
    {
        e.CanExecute = true;
    }
    else
    {
        e.CanExecute = false;
    }
}
```

**Ergebnis**

Ein Test zur Laufzeit zeigt, dass die entsprechende Schaltfläche nur freigegeben ist, wenn die *TextBox* leer ist.

Klicken Sie auf den obigen Menüpunkt oder verwenden Sie die Tastenkombination **Strg+O**, wird unsere *MessageBox* aus der Methode *OpenCmdExecuted* ausgeführt.



**HINWEIS:** Selbstverständlich können Sie *CommandBinding* auch per Code realisieren, wie das folgende Beispiel zeigt.

**Beispiel 11.19:** Ereignismethode per Code zuweisen**C#**

```
public MainWindow()
{
    InitializeComponent();
    CommandBinding cmdOpen =
        new CommandBinding(ApplicationCommands.Open);
    cmdOpen.Executed += new ExecutedRoutedEventHandler(OpenCmdExecuted);
}
```

**11.4.6 Wie kann ich ein Command per Code auslösen?**

Neben der bereits beschriebenen Variante, Commands per Zuordnung im XAML-Code auszulösen, besteht auch die Möglichkeit, diese direkt mit der *Execute*-Methode aufzurufen.

**Beispiel 11.20:** Direkter Aufruf eines Commands in einer Ereignismethode

```
C#
private void Button_Click(object sender, RoutedEventArgs e)
{
    ApplicationCommands.Paste.Execute(null, txt2);
}
```

Im zweiten Parameter übergeben Sie das *CommandTarget*.

### 11.4.7 Command-Ausführung verhindern

Nicht immer und zu jeder Zeit können Kommandos einfach ausgeführt werden. Unter bestimmten Bedingungen steht eine Funktion nicht zur Verfügung, in diesem Fall sollen die Menüpunkte/Schaltflächen abgeblendet sein, um den Anwender nicht unnötig zu verwirren.

Die Lösungsmöglichkeit haben wir bereits beim Zuordnen von Ereignismethoden gezeigt. Im ... *CanExecute*-Ereignis wird mit dem Parameter *e.CanExecute* entschieden, ob eine Aktion ausführbar ist oder nicht.

**Beispiel 11.21:** Command-Ausführung verhindern

```
C#
private void OpenCmdCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = false;
}
```

## ■ 11.5 Das WPF-Style-System

Mit den WPF-Styles kommen wir jetzt zu einem Thema, das sicher von ganz zentraler Bedeutung für die oberflächenorientierten WPF-Anwendungen ist und auch einen der wesentlichsten Unterschiede zu den konventionellen Windows-Anwendungen darstellt.

### 11.5.1 Übersicht

Doch worum geht es eigentlich? Sicher haben Sie nach der Lektüre der beiden vorhergehenden Kapitel festgestellt, dass WPF-Controls mit Bergen von Eigenschaften ausgestattet sind, die es möglich machen, fast alle Aspekte der Darstellung zu beeinflussen.

Doch gerade für aufwendige Oberflächen ergeben sich einige Fragen:

- Wie kann ich mehr als einem Control ein spezifisches Aussehen zuweisen?
- Wie kann ich das Aussehen unter bestimmten Bedingungen ändern?
- Wie kann ich das grundsätzliche Aussehen eines Controls komplett ändern?
- Wie kann ich einfache Animationen realisieren?

Für alle diese Fragen bietet WPF eine Antwort:

- Verwendung von benannten oder Typ-Styles
- Verwendung von Triggern
- Verwendung von Templates
- Verwendung von StoryBoards



**HINWEIS:** Im Rahmen dieses Abschnitts werden wir die oben genannten Themen nur recht oberflächlich streifen, da dies eigentlich ein Hauptarbeitsgebiet für den Designer und nicht für den Programmierer ist. Außerdem ist Visual Studio für einige der obigen Aufgaben das falsche Tool. Für WPF-Designer gibt es das Tool *Blend for Visual Studio*, mit dem man komfortabel XAML-Dateien erstellen kann.

### 11.5.2 Benannte Styles

In den bisherigen Beispielen haben wir die Eigenschaften jedes Controls einzeln gesetzt, was bei größeren Ansammlungen recht schnell zum Geduldspiel ausarten kann. Denken Sie nur an unseren Taschenrechner aus dem WPF-Einführungskapitel, wo ca. zwanzig einzelne Tasten zu konfigurieren sind (Schrift, Randabstände, Farben etc.). Viel schöner wäre doch hier eine zentrale Vorschrift, wie ein derartiger Button auszusehen hat.

Genau diese Aufgabe übernimmt ein *benannter Style*. Dieser wird einmal definiert und kann dann per Key den einzelnen Elementen zugewiesen werden.

**Beispiel 11.22:** Alle Schaltflächen für den Taschenrechner sollen einen Randabstand von 2, eine fette Schrift und eine gelbe Hintergrundfarbe bekommen.

#### XAML

```
<Window x:Class="StyleSample.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:StyleSample"
  mc:Ignorable="d"
  Title="MainWindow" Height="250" Width="400">
```

Jetzt wird auch klar, warum wir uns in diesem Kapitel bereits mit Ressourcen beschäftigt haben. Hoffentlich haben Sie diesen Abschnitt nicht gelangweilt überblättert!

```
<Window.Resources>
```

Wir definieren einen neuen Style und legen dessen *Key* fest (über diesen können wir später auf den Style zugreifen bzw. auf diesen verweisen):

```
<Style x:Key="myBtnStyle">
```

Innerhalb des *Style*-Elements können Sie per *Setter* die gewünschten Eigenschaften beeinflussen:

```
<Setter Property="Control.Margin" Value="2" />
<Setter Property="Control.Background" Value="Yellow" />
<Setter Property="Control.FontWeight" Value="UltraBold" />
```

Legen Sie jeweils *Property* und *Value* fest.

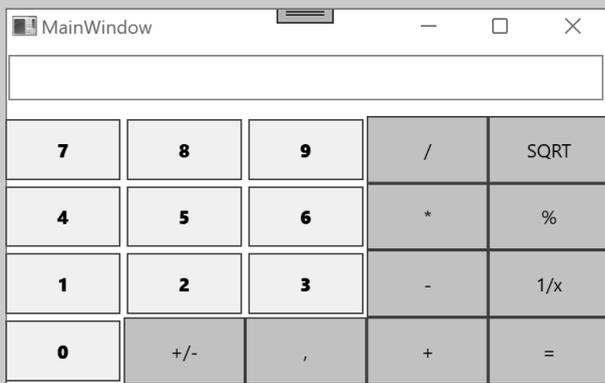
```
</Style>
</Window.Resources>
```

Und jetzt kommt unser neuer Style zum Einsatz (setzen Sie diesen nur bei den Zifferntasten):

```
<UniformGrid Name="uniformGrid1" Columns="5" Rows="4" Grid.Row="1">
  <Button Name="button1" Style="{StaticResource myBtnStyle}">7</Button>
  <Button Name="button2" Style="{StaticResource myBtnStyle}">8</Button>
  <Button Name="button3" Style="{StaticResource myBtnStyle}">9</Button>
  ...
```

## Ergebnis

Schon im Designer dürfte sich jetzt etwas getan haben. Bild 11.8 zeigt die App während der Laufzeit.



**Bild 11.8** Button mit Zahlen mit einem einheitlichen Style

Was passiert eigentlich, wenn wir den Style einer *TextBox* zuweisen? Probieren Sie es ruhig aus, es kann nichts passieren. Vielleicht sind Sie überrascht, aber auch die *TextBox* wird nach dieser Aktion im gelben Outfit erscheinen und eine fette Schrift anzeigen.

Warum dies so ist? Ganz einfach, auch die *TextBox* verfügt über die aufgeführten Eigenschaften und übernimmt diese automatisch vom Style.

### 11.5.3 Typ-Styles

Ein „fauler“ Programmierer (sind wir das nicht alle?) wird immer einen einfacheren Weg suchen, und so ist es sicher mühsam, den Style jedem einzelnen Button explizit zuzuweisen, zumal die Syntax auch recht umfangreich ist. Aus diesem Grund gibt es noch eine zweite Art von Styles, die nicht über einen Key, sondern indirekt über den Klassennamen zugeordnet werden.

Der Vorteil: Alle WPF-Elemente, die Instanzen dieser Klasse sind, übernehmen automatisch diesen Style, ohne dass dies explizit angegeben werden muss.

**Beispiel 11.23:** (Fortsetzung) Mit Ausnahme der Zifferntaste sollen alle Tasten einen grünen Hintergrund und eine weiße Schrift bekommen.

#### XAML

```
<Window.Resources>
```

Hier der benannte Style für die Zifferntasten:

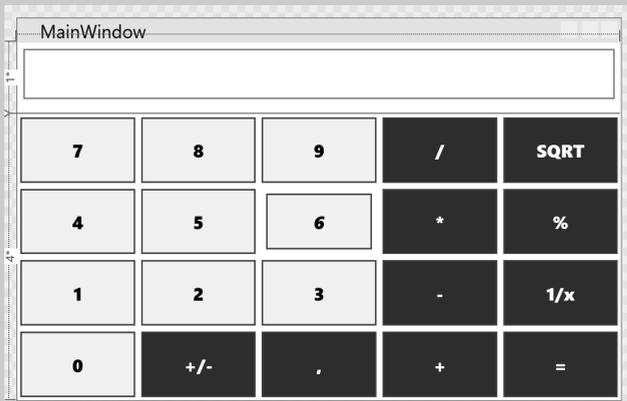
```
<Style x:Key="myBtnStyle">
...
</Style>
```

Und hier der Default-Style für alle Elemente vom Typ *Button*:

```
<Style TargetType="{x:Type Button}">
  <Setter Property="Margin" Value="2" />
  <Setter Property="Background" Value="Green" />
  <Setter Property="FontWeight" Value="UltraBold" />
  <Setter Property="Foreground" Value="White" />
</Style>
```

#### Ergebnis

Ohne weitere Änderungen im XAML-Code dürfte sich jetzt bereits folgender Anblick im Designer bieten (Bild 11.9).



**Bild 11.9** Buttons mit zwei unterschiedlichen Style-Varianten



**HINWEIS:** Hätten Sie die Style-Definition in der Datei *App.xaml* eingefügt, würde es sich um einen anwendungsweiten Style handeln – alle Windows bzw. die enthaltenen Schaltflächen würden dieses Outfit bekommen.

Doch was ist, wenn sich auf Ihrem Formular zum Beispiel ein paar *ToggleButton*-Controls befinden? Diese sind von obiger Style-Definition nicht betroffen, handelt es sich doch um eine andere Klasse.

### 11.5.4 Styles anpassen und vererben

Es gibt immer wieder Ausnahmen von der Regel, und so kommen Sie meist nicht darum herum, den Style einzelner Controls speziell anzupassen. Drei Varianten bieten sich dazu an:

- Sie überschreiben einzelne Attribute direkt im Element.
- Sie ersetzen den Style auf einer niedrigeren Ebene (statt *Application* z. B. in einem *StackPanel*).
- Sie vererben den Style und passen ihn unter neuem Namen an.

#### Styles anpassen (überschreiben)

Das Überschreiben von Styles ist eigentlich ganz intuitiv möglich. Geben Sie den Key des Styles an oder nutzen Sie einen Typ-Style wie bisher. Gleichzeitig erweitern Sie die Attributliste des betreffenden Elements, um die gewünschten Änderungen vorzunehmen.

**Beispiel 11.24:** (Fortsetzung) Die Taste „0“ soll rot hinterlegt werden (der Style gibt gelb vor).

#### XAML

```
<Button Name="button16" Style="{StaticResource myBtnStyle}"
        Background="Red" >0</Button>
```

#### Style ersetzen

Fällt ein komplettes Formular aus dem Rahmen oder möchten Sie einzelne Elemente einer Gruppe (*StackPanel*, *Grid* etc.) mit einem angepassten Style versehen, können Sie auch den zentral gültigen Style ersetzen. Definieren Sie dazu einen neuen *Resources*-Abschnitt und fügen Sie die neue Style-Definition in diesen ein.

**Beispiel 11.25:** (Fortsetzung) Ersetzen des zentralen Button-Styles

#### XAML

```
<Window x:Class="StyleSample.MainWindow"
...

```

Hier die übergreifende Definition:

```
<Window.Resources>
  <Style TargetType="{x:Type Button}">
```

```

...
</Style>
</Window.Resources>
...
<UniformGrid Name="uniformGrid1" Columns="5" Rows="4" Grid.Row="1">

```

Hier wird der Style **ersetzt**, d. h. alle obigen Einstellungen gehen verloren:

```

<UniformGrid.Resources>
  <Style TargetType="{x:Type Button}">
    <Setter Property="Margin" Value="2" />
    <Setter Property="Background" Value="Blue" />
    <Setter Property="FontWeight" Value="UltraBold" />
    <Setter Property="Foreground" Value="White" />
  </Style>
</UniformGrid.Resources>
...

```



**HINWEIS:** Elemente, die sich in der Hierarchie oberhalb des *UniformGrids* befinden, sind nicht von dieser Anpassung betroffen, hier gilt wieder die zentrale Version.

## Styles vererben

Wer schreibfaul ist und beispielsweise nicht den kompletten Style austauschen will, kann diesen auch einfach vererben. Dazu wird in die Definition des Styles das Attribut *BasedOn* aufgenommen, das per Binding auf den Basisstyle verweist.

Ob Sie in diesem neuen Style bestehende Eigenschaften überschreiben (einfach erneut definieren) oder neue Eigenschaften hinzufügen, bleibt Ihnen überlassen.

### Beispiel 11.26: Vererben eines Styles

#### XAML

```
<Window.Resources>
```

Das Original:

```

<Style x:Key="myBtnStyle">
  <Setter Property="Control.Margin" Value="2" />
  <Setter Property="Control.Background" Value="Yellow" />
  <Setter Property="Control.FontWeight" Value="UltraBold" />
</Style>

```

Der „Erbe“ mit geringfügiger Änderung:

```
<Style x:Key="myBtnStyle2" BasedOn="{StaticResource myBtnStyle}">
```

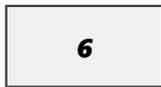
Hier wird eine Eigenschaft überschrieben:

```
<Setter Property="Control.Margin" Value="5" />
```

Hier wird eine neue Eigenschaft gesetzt:

```
<Setter Property="Control.FontStyle" Value="Italic" />
</Style>
</Window.Resources>
```

#### Ergebnis



**Bild 11.10**

Links der Basis-Style, rechts der neue Style

## Styleänderung per Code

Ihr Programm ist nicht darauf angewiesen, immer den gleichen Style zu verwenden. So ist es problemlos möglich, auch zur Laufzeit einen Style per Code neu zu setzen (z. B. unter bestimmten Bedingungen).

**Beispiel 11.27:** Der Style von *button10* wird geändert

C#

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Ziffer 3
    button13.Style = (Style)FindResource("myBtnStyle2");
}
```



**HINWEIS:** Den Style bzw. dessen Instanz finden Sie mit der Methode *FindResource* über dessen Key.



**HINWEIS:** Bevor Sie sich weiter in diese Thematik vertiefen, werfen Sie zunächst einen Blick auf den folgenden Abschnitt; vielleicht können Sie damit bestimmte Aufgaben eleganter lösen.

## ■ 11.6 Verwenden von Triggern

In unseren bisherigen Experimenten waren die vom Style vorgenommenen Änderungen immer statischer Natur, d. h. einmal gesetzt, blieb die Optik immer gleich. Doch dies kann sicher nicht der Weisheit letzter Schluss sein.

Wenn jetzt in Ihnen der Programmierer wieder durchkommt und Sie an C# und Ereignisprozeduren denken, vergessen Sie es gleich wieder. Für (fast) alle Aufgabenstellungen ist auch hier XAML die beste Lösung.

Für die Reaktion auf Eigenschaftsänderungen, Ereignisse, Datenänderungen etc. können Sie in WPF-Anwendungen sogenannte Trigger verwenden und damit zum Beispiel den Style ändern. Dabei sind Sie nicht auf einen einzelnen Trigger angewiesen, sondern Sie können der Trigger-Collection auch mehrere Ereignisse mit unterschiedlichen Bedingungen zuweisen.

Im Folgenden wollen wir uns die verschiedenen Triggerarten näher ansehen.

### 11.6.1 Eigenschaften-Trigger (Property Triggers)

Sicher kennen Sie auch das eine oder andere Programm, das exzessiv Gebrauch von diversen optischen Spielereien macht. Wird beispielsweise mit dem Mauscursor auf ein Control gezeigt, ändert sich dessen Rahmen oder die Hintergrundfarbe. Gleiches gilt für Eingabefelder, die den Fokus erhalten, etc. In all diesen Fällen ändern sich Eigenschaften (*IsMouseOver*, *IsFocused*), auf die Sie bei der konventionellen Programmierung mit Ereignismethoden reagieren können. Mit Eigenschaften-Trigger können Sie Ihren C#-Quellcode von derartigem Ballast befreien und direkt per XAML-Code Änderungen am Control vornehmen.

**Beispiel 11.28:** Eine *TextBox* soll auf *IsMouseOver* und *IsFocused* mit Farbänderungen reagieren.

#### XAML

```
<Window x:Class="TriggerSample.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:TriggerSample"
  mc:Ignorable="d"
  Title="MainWindow" Height="300" Width="300">
```

Einen Style für die *TextBox* erzeugen:

```
<Window.Resources>
  <Style x:Key="myStyle" TargetType="{x:Type TextBox}">
```

Der Außenabstand soll immer 2 betragen:

```
<Setter Property="Margin" Value="2" />
```

Hier werden die Trigger definiert:

```
<Style.Triggers>
```

Unter der Bedingung ...

```
<Trigger Property="IsMouseOver" Value="True">
```

... wird die folgende Eigenschaft gesetzt:

```
<Setter Property="Background" Value="Yellow" />
</Trigger>
```

Unter der Bedingung ...

```
<Trigger Property="IsFocused" Value="True">
```

... werden die folgenden Eigenschaften gesetzt:

```
    <Setter Property="Background" Value="Blue" />
    <Setter Property="Foreground" Value="White" />
  </Trigger>
</Style.Triggers>
</Style>
</Window.Resources>
<StackPanel>
```

Hier verwenden wir den Style:

```
  <TextBox Style="{StaticResource myStyle}" >Hallo</TextBox>
  <TextBox Style="{StaticResource myStyle}" >Hallo</TextBox>
  <TextBox Style="{StaticResource myStyle}" >Hallo</TextBox>
</StackPanel>
</Window>
```

Änderungen, die durch einen Trigger vorgenommen wurden, werden automatisch wieder rückgängig gemacht, wenn die Bedingung nicht mehr eingehalten wird (automatisches Wiederherstellen des Standardwertes).

### Ergebnis

Bild 11.11 zeigt die Laufzeitansicht. Die erste TextBox hat den Eingabefokus, die zweite erfüllt die Bedingung *IsMouseOver=True* und die dritte TextBox ist im Standardzustand.



**Bild 11.11**  
Trigger im Einsatz

Doch was ist, wenn Sie mehr als eine Bedingung benötigen? Auch das ist kein Problem, in diesem Fall erzeugen Sie einfach einen „multi-condition property trigger“.

**Beispiel 11.29:** Der *TextBox*-Hintergrund soll rot werden, wenn die *TextBox* den Eingabefokus besitzt und wenn kein Text enthalten ist.

### XAML

```
<Window.Resources>
  <Style x:Key="myStyle" TargetType="{x:Type TextBox}">
  ...
```

Einen Multi-Condition-Trigger definieren:

```
<MultiTrigger>
  <MultiTrigger.Conditions>
```

Die beiden folgenden Bedingungen müssen zutreffen:

```
<Condition Property="IsFocused" Value="True"/>
<Condition Property="Text" Value="" />
</MultiTrigger.Conditions>
```

Hier die Aktion:

```
<Setter Property="Background" Value="Red" />
</MultiTrigger>
</Style.Triggers>
</Style>
</Window.Resources>
```

## 11.6.2 Ereignis-Trigger

Ereignis-Trigger werden durch bestimmte Ereignisse (vom Typ *RoutedEvent*) ausgelöst. Im Gegensatz zu den Eigenschaften-Trigger können Sie über derartige Trigger jedoch direkt keine Eigenschaften ändern, Sie können „lediglich“ Animationen starten, die sich wiederum auf Eigenschaften auswirken.



**HINWEIS:** Ereignis-Trigger setzen geänderte Eigenschaften nicht wieder zurück, dafür sind **Sie** als Programmierer verantwortlich (z. B. im Pendant des betreffenden Ereignisses).

**Beispiel 11.30:** Wird der Mauscursor über einen Button bewegt, wird dieser transparent.

### XAML

Zunächst die Style-Definition:

```
<Style TargetType="{x:Type Button}">
<Style.Triggers>
```

Hier kommt unser Ereignis-Trigger:

```
<EventTrigger RoutedEvent="Button.MouseEnter">
```

Wir reagieren mit einer Aktion über die Hintergründe:

```
<EventTrigger.Actions>
<BeginStoryboard>
```

Hier die eigentliche Aktion – die Eigenschaft *Opacity* soll in 4 Sekunden von 1 auf 0.25 verringert werden:

```
<Storyboard>
<DoubleAnimation From="1" To="0.25"
Duration="0:0:4"
```

```

        Storyboard.TargetProperty="(Opacity)"/>
    </Storyboard>
</BeginStoryboard>
</EventTrigger.Actions>
</EventTrigger>

```

Beim *MouseLeave*-Ereignis gehen wir den umgekehrten Weg und stellen die ursprüngliche Transparenz wieder her:

```

    <EventTrigger RoutedEvent="Button.MouseLeave">
        <EventTrigger.Actions>
            <BeginStoryboard>
                <Storyboard>
                    <DoubleAnimation From="0.25" To="1"
Duration="0:0:4"
                    Storyboard.TargetProperty="(Opacity)"/>
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger.Actions>
    </EventTrigger>
</Style.Triggers>
</Style>

```

### 11.6.3 Daten-Trigger

Mit diesen Triggern können Sie auf das Ändern beliebiger Eigenschaften reagieren. Die Verbindung zu den entsprechenden Eigenschaften stellen Sie per Bindung her. Als Reaktion auf eine Eigenschaftsänderung können Sie, wie auch bei den Eigenschaften-Triggern, mit einem *Setter*-Element bestimmte Eigenschaften ändern.



**HINWEIS:** Die durch den Trigger geänderten Eigenschaften werden automatisch zurückgesetzt, wenn die Bedingung nicht mehr erfüllt ist.

**Beispiel 11.31:** Enthält die *TextBox* mehr als zehn Zeichen, wird sie grün eingefärbt.

#### XAML

```

<Window.Resources>
    <Style x:Key="myStyle" TargetType="{x:Type TextBox}">
    ...
        <DataTrigger Binding="{Binding RelativeSource={RelativeSource Self},
            Path=Text.Length}" Value="10">
            <Setter Property="Background" Value="Green" />
        </DataTrigger>
    </Style.Triggers>
</Style>
</Window.Resources>

```

## ■ 11.7 Einsatz von Templates

Im Folgenden möchten wir Sie zunächst mit ein paar Grundaussagen konfrontieren, bevor wir uns der Thematik „Templates“ bzw. „Vorlagen“ widmen:

- WPF-Controls haben prinzipiell keine Zeichenlogik, es handelt sich um Lookless Controls.
- WPF-Controls stellen lediglich eine Sammlung von Verhalten (Behaviors) dar.

Spinnen denn die Römer? Wo kommen denn sonst die ganzen optischen Spielereien her? Die Antwort auf dieses Paradoxon: Die Zeichenlogik eines Controls wird nur vom Layout/Styling bestimmt. Jedes Control besitzt ein Standardaussehen (Default-Template), das komplett ersetzt werden kann.

Hier haben Sie es mit der Spielwiese der Designer zu tun. Aus dem guten alten viereckigen Button kann ein gänzlich anderes Objekt werden, das jedoch nach wie vor das wesentliche Verhalten eines Buttons (Klick) besitzt. Aus Sicht des Programmierers kann dieser neue Button wie der Standard-Button verwendet werden. Damit ersparen Templates uns vielfach die Mühe, Controls umständlich abzuleiten und deren Zeichenlogik per C#-Code komplett neu zu implementieren.



**HINWEIS:** Im Gegensatz zu den Styles können Sie bei den Templates nicht nur die vorhandenen Eigenschaften beeinflussen, sondern das Control auch von Grund auf neu zusammenbauen.

### 11.7.1 Neues Template erstellen

Ein etwas komplexeres Beispiel soll die prinzipielle Vorgehensweise verdeutlichen. Eine Komplettübersicht dieses Themas können wir Ihnen an dieser Stelle leider nicht geben.

**Beispiel 11.32:** Erzeugen und Verwenden eines Templates

#### XAML

Unsere Schaltflächen sollen ellipsenförmig sein und einen Farbverlauf aufweisen. Ist die Maus über dem Control, soll sich die Schriftstärke ändern. Ein Niederdrücken der Schaltfläche führt zu einem umgekehrten Farbverlauf im Control.

```
<Window.Resources>
```

Zunächst erstellen wir einen neuen Type-Style für *Button*-Elemente (Sie können auch einen benannten Style verwenden):

```
<Style TargetType="{x:Type Button}">
```

Hier greifen wir erstmals auf das Template zu. Die Definition ist etwas verschachtelt, da es sich um eine recht komplexe Zuweisung handelt:

```
<Setter Property="Template">
  <Setter.Value>
```

Der Eigenschaft *Template* wird ein *ControlTemplate* zugewiesen:

```
<ControlTemplate TargetType="{x:Type Button}">
```

Für die innere Ausrichtung nutzen wir zunächst ein *Grid*:

```
<Grid HorizontalAlignment="Stretch"
      VerticalAlignment="Stretch" ClipToBounds="False">
```

Und hier haben wir es schon mit der Optik zu tun. Wir erzeugen eine Ellipse mit einem Farbverlauf (dies ist der Defaultzustand):

```
<Ellipse>
  <Ellipse.Fill>
    <LinearGradientBrush StartPoint="0,0" EndPoint="0,1" >
      <LinearGradientBrush.GradientStops>
        <GradientStop Color="#fff399" Offset="0.1"/>
        <GradientStop Color="#ffe100" Offset="0.5"/>
        <GradientStop Color="#feca00" Offset="0.9"/>
      </LinearGradientBrush.GradientStops>
    </LinearGradientBrush>
  </Ellipse.Fill>
</Ellipse>
```

In unserem Button soll auch etwas angezeigt werden, dafür ist das *ContentPresenter*-Element verantwortlich:

```
<ContentPresenter x:Name="PrimaryContent"
  HorizontalAlignment="Center" VerticalAlignment="Center"
```

Den eigentlichen Inhalt holen wir uns wiederum vom ursprünglichen Element (dem *Button*), deshalb auch die etwas umständliche Bindung:

```
Content="{Binding Path=Content,
  RelativeSource={RelativeSource TemplatedParent}}" />
</Grid>
```

Nicht nur die Standardanzeige unseres Buttons wollen wir beeinflussen, sondern auch die Reaktion auf Maus und Klicken:

```
<ControlTemplate.Triggers>
```

Es wird geklickt, d. h. wir zeichnen einen neuen Hintergrund. Dazu benötigen wir allerdings den Namen der Ellipse:

```
<Trigger Property="Button.IsPressed" Value="True">
  <Setter Property="Fill" TargetName="elli" >
```

Achtung: Diese Namen sind nur innerhalb des Templates verwendbar!

```
<Setter.Value>
  <LinearGradientBrush StartPoint="0,1" EndPoint="0,0" >
    <LinearGradientBrush.GradientStops>
      <GradientStop Color="#fff399" Offset="0.1" />
      <GradientStop Color="#ffe100" Offset="0.5" />
      <GradientStop Color="#feca00" Offset="0.9" />
    </LinearGradientBrush.GradientStops>
  </LinearGradientBrush>
</Setter.Value>
</Setter>
</Trigger>
```

Die Maus wird über das Control bewegt. In diesem Fall wird lediglich die Schriftstärke geändert:

```
<Trigger Property="Button.IsMouseOver" Value="True">
  <Setter Property="FontWeight" Value="Bold" />
</Trigger>
```

Hier könnten Sie noch auf weitere Eigenschaftsänderungen reagieren ...

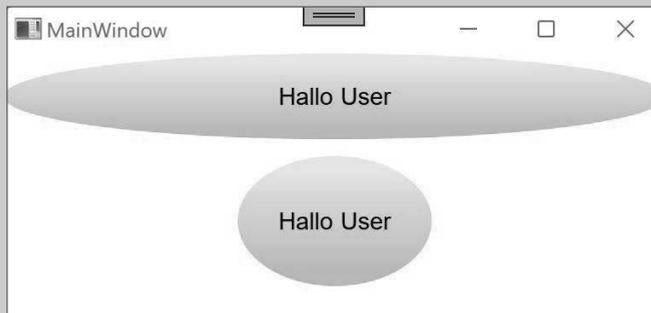
```
</ControlTemplate.Triggers>
</ControlTemplate>
</Setter.Value>
</Setter>
```

Last, but not least, setzen wir noch ein paar Eigenschaften:

```
<Setter Property="Foreground" Value="Black" />
<Setter Property="FontFamily" Value="Arial" />
<Setter Property="FontSize" Value="14" />
</Style>
</Window.Resources>
<StackPanel>
  <Button Height="50">Hallo User</Button>
  <Button Height="76" Margin="10" Width="113">Hallo User
</Button>
</StackPanel>
```

## Ergebnis

Und wie sieht nun unsere neue Schaltfläche aus? Die Antwort sehen Sie in Bild 11.12.



**Bild 11.12**  
Buttons mit geändertem  
Template

Doch wie können Sie innerhalb des Templates auf die ursprüngliche Definition von Eigenschaften zugreifen? Hier hilft Ihnen ebenfalls Binding weiter.

**Beispiel 11.33:** Verwendung der *Button.Background*-Eigenschaft

#### XAML

```
<ControlTemplate TargetType="{x:Type Button}">
  <Grid HorizontalAlignment="Stretch" VerticalAlignment="vStretch"
        ClipToBounds="False">
    <Rectangle Fill="{TemplateBinding Property=Background}"/>
    <Ellipse Name="elli">
```

#### Ergebnis

Starten Sie das Programm mit dieser Änderung, taucht im Hintergrund zunächst der Default-Farbverlauf eines Buttons auf (so ist *Background* für einen *Button* auch gesetzt).



**Bild 11.13** Der graue Hintergrund ist in den Ecken sichtbar.

Werfen wir noch einen Blick auf das *ContentPresenter*-Element in unserem obigen Template. Dessen Content stammt vom ursprünglichen Element ab und bietet damit ebenfalls die Möglichkeit, komplexe Controls „zusammenzubasteln“.

**Beispiel 11.34:** Wir definieren den zweiten Button mit einer Grafik, der Button übernimmt den vorliegenden Style.

#### XAML

```
<Button Height="76" Margin="10" Width="113">
  <StackPanel Orientation="Horizontal">
    <Image Source="Images/flash.png" Width="26" Height="26"
          Margin="0,0,10,0"/>
    <TextBlock VerticalAlignment="Center">Action</TextBlock>
  </StackPanel>
</Button>
```

#### Ergebnis

Das zusammengewürfelte Endergebnis (Grafik und Text per *Content*, Grundlayout per *Template*) präsentiert sich nach wie vor als vollwertiger Button:



**Bild 11.14**  
Button mit Image und TextBox

### 11.7.2 Template abrufen und verändern

Möchten Sie von einem vorhandenen Control das Template abrufen und eventuell verändern, ist dies seit Visual Studio 2012 kein Problem mehr<sup>3</sup>.

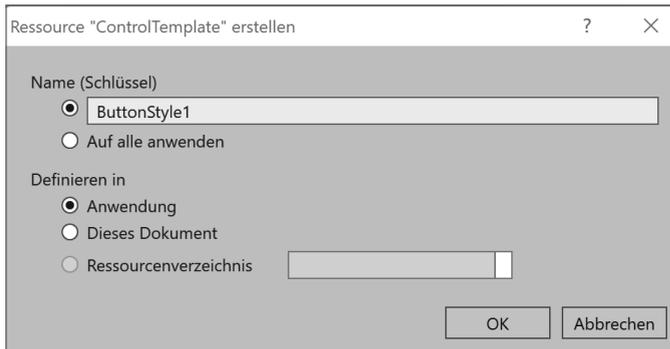
Sie können von jedem Control direkt aus dem Designer heraus eine Kopie des Default-Templates editieren. Markieren Sie das betreffende Control im WPF-Editor-Fenster und klicken Sie mit der rechten Maustaste. Über das Kontextmenü rufen Sie *Vorlage bearbeiten/Kopie bearbeiten* auf:



**Bild 11.15** Kopie eines Templates erstellen

Im folgenden Dialog (Bild 11.16) bestimmen Sie, ob das Template per Name (benannt) oder per Typ zugeordnet werden soll und wo das Template erstellt wird. Auf das Speichern im aktuellen Window (DIESES DOKUMENT) sollten Sie aus Gründen der Übersichtlichkeit verzichten. Mit der Option *Anwendung* wird das Template in der *app.xaml* erstellt.

<sup>3</sup> In früheren Versionen waren Sie auf die Anwendung *Expression Blend* angewiesen, mittlerweile ist der WPF-Editor in Visual Studio massiv aufgerüstet worden.



**Bild 11.16** ControlTemplate erstellen

**Beispiel 11.35:** Die derart erstellte Kopie sieht wie folgt aus (Beispiel *Button*):

#### XAML

```
<Application.Resources>
```

Als Erstes wird ein *FocusVisualStyle* erstellt:

```
<Style x:Key="FocusVisual">
  <Setter Property="Control.Template">
    <Setter.Value>
      <ControlTemplate>
        <Rectangle Margin="2" StrokeDashArray="1 2"
          Stroke="{DynamicResource {x:Static SystemColors.ControlTextBrushKey}}"
          SnapsToDevicePixels="true" StrokeThickness="1"/>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

Dann folgen Farbdefinitionen:

```
<SolidColorBrush x:Key="Button.Static.Background" Color="#FFDDDDDD"/>
<SolidColorBrush x:Key="Button.Static.Border" Color="#FF707070"/>
<SolidColorBrush x:Key="Button.MouseOver.Background" Color="#FFBEE6FD"/>
<SolidColorBrush x:Key="Button.MouseOver.Border" Color="#FF3C7FB1"/>
<SolidColorBrush x:Key="Button.Pressed.Background" Color="#FFC4E5F6"/>
<SolidColorBrush x:Key="Button.Pressed.Border" Color="#FF2C628B"/>
<SolidColorBrush x:Key="Button.Disabled.Background" Color="#FFF4F4F4"/>
<SolidColorBrush x:Key="Button.Disabled.Border" Color="#FFADB2B5"/>
<SolidColorBrush x:Key="Button.Disabled.Foreground" Color="#FF838383"/>
```

Hier geht es mit der eigentlichen Definition des Buttons los, bei der zunächst einige Einstellungen zugewiesen werden:

```
<Style x:Key="ButtonStyle1" TargetType="{x:Type Button}">
  <Setter Property="FocusVisualStyle" Value="{StaticResource FocusVisual}"/>
  <Setter Property="Background"
    Value="{StaticResource Button.Static.Background}"/>
  <Setter Property="BorderBrush"
    Value="{StaticResource Button.Static.Border}"/>
```

Hier wird direkt eine Systemressource genutzt. Die Anpassung erfolgt dynamisch, d. h. bei Änderungen in der Systemsteuerung:

```
<Setter Property="Foreground"
    Value="{DynamicResource {x:Static SystemColors.ControlTextBrushKey}}"/>
```

Dann werden weitere Properties gesetzt:

```
<Setter Property="BorderThickness" Value="1"/>
<Setter Property="HorizontalAlignment" Value="Center"/>
<Setter Property="VerticalContentAlignment" Value="Center"/>
<Setter Property="Padding" Value="1"/>
```

Das Default-Template wird als Kopie über einen weiteren *Setter* definiert:

```
<Setter Property="Template">
    <Setter.Value>
        <ControlTemplate TargetType="{x:Type Button}">
            <Border x:Name="border" Background="{TemplateBinding
Background}"
                BorderBrush="{TemplateBinding BorderBrush}"
                BorderThickness="{TemplateBinding BorderThickness}"
                SnapsToDevicePixels="true">
                <ContentPresenter x:Name="contentPresenter"
                    Focusable="False"
                    HorizontalAlignment=
                        "{TemplateBinding HorizontalContentAlignment}"
                    Margin="{TemplateBinding Padding}"
                    RecognizesAccessKey="True"
                    SnapsToDevicePixels=
                        "{TemplateBinding SnapsToDevicePixels}"
                    VerticalAlignment=
                        "{TemplateBinding VerticalContentAlignment}"/>
            </Border>
```

Die Reaktion auf Eigenschaften-Trigger und Mausbewegungen:

```
<ControlTemplate.Triggers>
    <Trigger Property="IsDefaulted" Value="true">
        <Setter Property="BorderBrush" TargetName="border"
Value="{DynamicResource {x:Static SystemColors.HighlightBrushKey}}"/>
    </Trigger>
    <Trigger Property="IsMouseOver" Value="true">
        <Setter Property="Background" TargetName="border"
Value="{StaticResource Button.MouseOver.Background}"/>
        <Setter Property="BorderBrush" TargetName="border"
Value="{StaticResource Button.MouseOver.Border}"/>
    </Trigger>
    <Trigger Property="IsPressed" Value="true">
        <Setter Property="Background" TargetName="border"
Value="{StaticResource Button.Pressed.Background}"/>
        <Setter Property="BorderBrush" TargetName="border"
Value="{StaticResource Button.Pressed.Border}"/>
    </Trigger>
    <Trigger Property="IsEnabled" Value="false">
        <Setter Property="Background" TargetName="border"
```

```

Value="{StaticResource Button.Disabled.Background}"/>
    <Setter Property="BorderBrush" TargetName="border"
Value="{StaticResource Button.Disabled.Border}"/>
    <Setter Property="TextElement.Foreground"
        TargetName="contentPresenter"
Value="{StaticResource Button.Disabled.Foreground}"/>
    </Trigger>
</ControlTemplate.Triggers>
</ControlTemplate>
</Setter.Value>
</Setter>
</Style>
</Application.Resources>

```



**HINWEIS:** Ändern Sie in diesem Template beispielsweise die Farben im *IsMouseOver*-Trigger (siehe fett hervorgehobene Zeilen), dann wird sich das Aussehen aller Schaltflächen in Ihrer Anwendung ändern.

Hauptzielgruppe dieses Features dürfte jedoch nicht der Programmierer, sondern der Designer der Anwendung sein. Gleiches trifft ebenfalls auf das im folgenden Abschnitt behandelte Storyboard zu.

## ■ 11.8 Transformationen, Animationen, StoryBoards

Im Folgenden wollen wir in einem „Schnelldurchlauf für Programmierer“ noch einen Blick auf WPF-typische Features werfen, auch wenn diese im Allgemeinen nicht zum Hauptarbeitsgebiet des Entwicklers gehören.

### 11.8.1 Transformationen

Mithilfe von Transformationen können Sie in WPF das optische Standardverhalten problemlos verändern, ohne sich um Templates oder Styles kümmern zu müssen. Sie können die Größe, Position, Drehung und Verzerrung der betroffenen Controls über die einfache Zuweisung einer entsprechenden Transformation ändern (statische Änderung).



**HINWEIS:** Damit sind diese Operationen auch die Vorstufe für einfache Animationen (dynamische Änderungen in Abhängigkeit von der Zeit).

Folgende Möglichkeiten stehen Ihnen zur Verfügung:

Transformation	Beschreibung
<i>RotateTransform</i>	Element um einen bestimmten Winkel drehen
<i>ScaleTransform</i>	Element vergrößern/verkleinern
<i>SkewTransform</i>	Element verformen
<i>TranslateTransform</i>	Element verschieben
<i>MatrixTransform</i>	Zusammenfassen der obigen Transformationen per 3x3-Transformationsmatrix

Lassen Sie uns nun an einfachen Beispielen die Wirkung der jeweiligen Transformation demonstrieren.

### Drehen mit RotateTransform

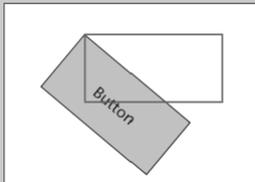
Mit *RotateTransform* realisieren Sie eine Drehung des Elements im Uhrzeigersinn. Den Drehpunkt können Sie optional festlegen, per Default ist dies die linke obere Ecke.

**Beispiel 11.36:** Button um 40° drehen

#### XAML

```
<Canvas>
  <Button Canvas.Left="100" Canvas.Top="100" Content="Button"
    Height="50" Width="100" >
    <Button.RenderTransform>
      <RotateTransform Angle="40"/>
    </Button.RenderTransform>
  </Button>
  <Rectangle Canvas.Left="100" Canvas.Top="100" Height="50"
    Stroke="Red" Width="100" />
</Canvas>
```

#### Ergebnis



**Bild 11.17**  
Um 40° gedrehter Button

Das *Angle*-Attribut bestimmt den Drehwinkel um den per *CenterX*- und *CenterY*-Attribut festgelegten Drehpunkt. Natürlich können Sie für eine andere Drehrichtung auch negative Werte übergeben.



**HINWEIS:** Durch die Rotation wird das Koordinatensystem des gedrehten Elements verändert!

## Skalieren mit ScaleTransform

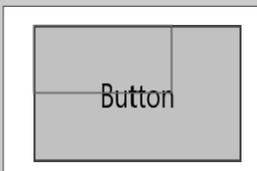
Soll ein Element skaliert werden, nutzen Sie eine *ScaleTransform*.

### Beispiel 11.37: Button skalieren

#### XAML

```
<Canvas>
  <Button Canvas.Left="100" Canvas.Top="300" Content="Button"
    Height="50" Width="100" >
    <Button.RenderTransform>
      <ScaleTransform ScaleX="1.5" ScaleY="2" />
    </Button.RenderTransform>
  </Button>
  <Rectangle Canvas.Left="100" Canvas.Top="300" Height="50"
    Stroke="Red" Width="100" />
</Canvas>
```

#### Ergebnis



**Bild 11.18**  
Button skalieren

Mit den *ScaleX*- und *ScaleY*-Attributen bestimmen Sie den Skalierungsfaktor für die X- bzw. Y-Achse. Mit *CenterX* und *CenterY* bestimmen Sie den Fixpunkt, von dem aus die Skalierung gestartet wird. Dies ist per Default die linke obere Ecke.

## Verformen mit SkewTransform

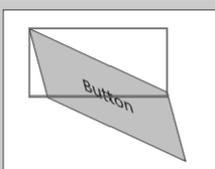
Mit *SkewTransform* verformen Sie das Koordinatensystem um bestimmte Winkel.

### Beispiel 11.38: Verformen des Koordinatensystems

#### XAML

```
<Canvas>
  <Button Canvas.Left="100" Canvas.Top="100" Content="Button"
    Height="50" Width="100" >
    <Button.RenderTransform>
      <SkewTransform AngleY="25" AngleX="15" />
    </Button.RenderTransform>
  </Button>
  <Rectangle Canvas.Left="100" Canvas.Top="100" Height="50"
    Stroke="Red" Width="100" />
</Canvas>
```

#### Ergebnis



**Bild 11.19**  
Verformung des Koordinatensystems

Den Verformungsgrad bestimmen Sie mit den Attributen *AngleX* und *AngleY*. *CenterX* und *CenterY* legen den Ursprungspunkt fest.

### Verschieben mit `TranslateTransform`

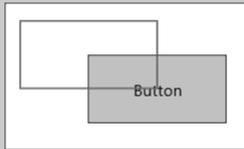
Auch die Verschiebung eines Elements ist mit *TranslateTransform* kein Problem. Sie können mit den X- und Y-Attributen die horizontale bzw. vertikale Verschiebung bestimmen.

#### Beispiel 11.39: Button verschieben

##### XAML

```
<Canvas>
  <Button Canvas.Left="300" Canvas.Top="300" Content="Button"
    Height="50" Width="100" >
    <Button.RenderTransform>
      <TranslateTransform X="50" Y="25"/>
    </Button.RenderTransform>
  </Button>
  <Rectangle Canvas.Left="300" Canvas.Top="300" Height="50"
    Stroke="Red" Width="100" />
</Canvas>
```

##### Ergebnis



**Bild 11.20**  
Button verschieben

### Und alles zusammen mit `TransformGroup`

Sicher sind Sie auch schon versucht gewesen, mehrere der obigen Effekte gleichzeitig zu realisieren. Allerdings dürfte Ihnen die Syntaxprüfung des XAML-Editors dabei einen Strich durch die Rechnung gemacht haben.



**HINWEIS:** Um mehrere Transformationen gleichzeitig zuzuweisen, müssen Sie eine *TransformGroup* verwenden.

#### Beispiel 11.40: Mehrere Transformationen gleichzeitig anwenden

##### XAML

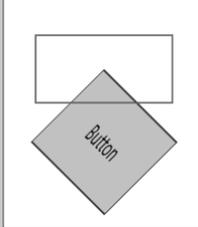
```
<Canvas>
  <Button Canvas.Left="500" Canvas.Top="100" Content="Button"
    Height="50" Width="100" >
    <Button.RenderTransform>
      <TransformGroup>
        <ScaleTransform ScaleX=".75" ScaleY="1.5"/>
        <RotateTransform Angle="45"/></RotateTransform>
        <TranslateTransform X="50" Y="25"/>
      </TransformGroup>
    </Button.RenderTransform>
  </Button>
</Canvas>
```

```

    </Button.RenderTransform>
  </Button>
  <Rectangle Canvas.Left="500" Canvas.Top="100" Height="50"
    Stroke="Red" Width="100" />
</Canvas>

```

### Ergebnis



**Bild 11.21**  
Transformationsgruppen

Doch Achtung: Hier spielt die Reihenfolge in der Gruppe eine bedeutende Rolle, wie folgende kleine Änderung (erst Drehung, dann Skalierung) zeigt. Ursache ist die Veränderung des Koordinatensystems des betroffenen Controls.

### Beispiel 11.41: Änderung der Transformationsreihenfolge

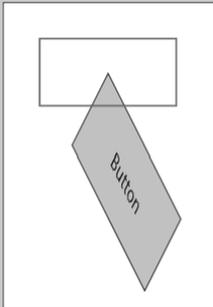
#### XAML

```

<TransformGroup>
  <RotateTransform Angle="45"></RotateTransform>
  <ScaleTransform ScaleX=".75" ScaleY="1.5"/>
  <TranslateTransform X="50" Y="25"/>
</TransformGroup>

```

### Ergebnis

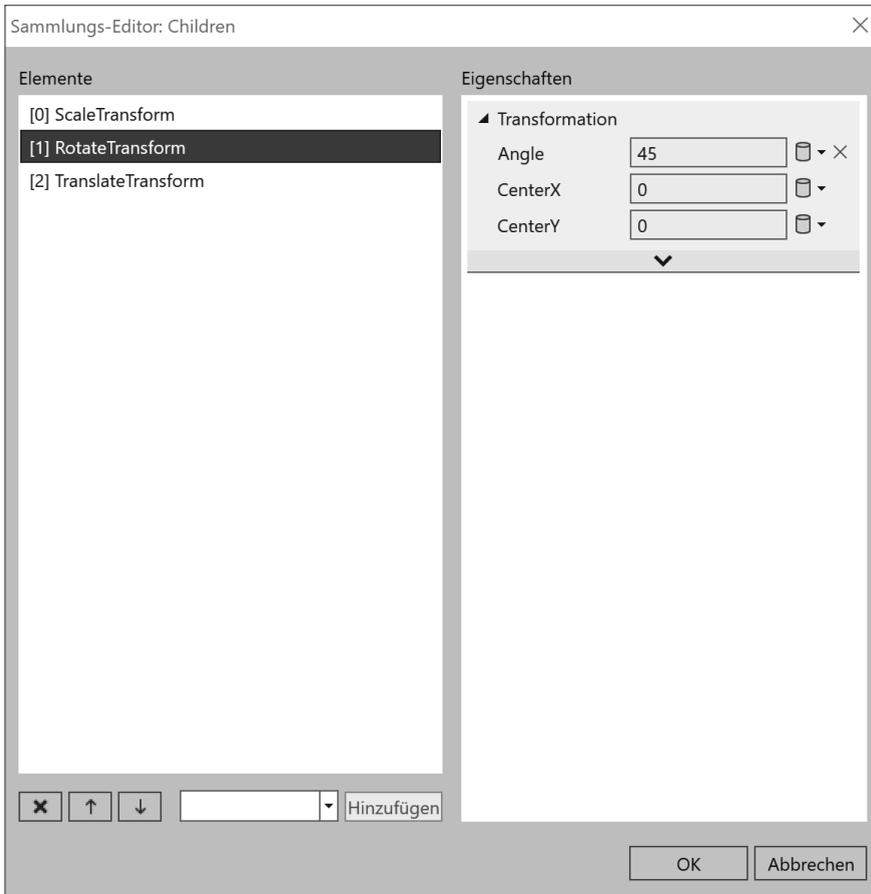


**Bild 11.22**  
Transformationsgruppe mit anderer Reihenfolge

### Hilfe durch den WPF-Editor

Wem die obigen Ausführungen zu kompliziert und wenig intuitiv waren, der kann es auch einfacher haben. Nutzen Sie einfach den in Visual Studio integrierten WPF-Editor, dieser ist mittlerweile sogar ganz brauchbar und bietet unter anderem auch die Möglichkeit, die *Rotate-Transformation* per Maus umzusetzen.

Im Eigenschaftenfenster können Sie zusätzlich die anderen Transformationsarten getrennt parametrieren:



**Bild 11.23** Editor für RotateTransform

## 11.8.2 Animationen mit dem StoryBoard realisieren

Für die Realisierung von Animationen werden in WPF sogenannte Storyboards verwendet, die wiederum einzelne oder mehrere Animationen (zeitliche Veränderungen von Eigenschaften) enthalten können. Storyboards können wiederum über bestimmte Ereignis-Trigger ausgelöst, angehalten, fortgesetzt oder auch beendet werden (alternativ natürlich auch per Code).

Ein erstes einfaches Beispiel soll die prinzipielle Vorgehensweise beim Animieren einer Eigenschaft (in diesem Fall der Transparenz) demonstrieren.

**Beispiel 11.42:** Ausblenden eines Buttons, wenn die Maus darüber bewegt wird

#### XAML

```
<Canvas>
```

Zunächst den Button definieren:

```
<Button Canvas.Left="700" Canvas.Top="100" Content="Button"
        Height="50" Width="100" Name="button1" >
```

Wir reagieren mit einem *Trigger* auf das Hineinbewegen der Maus:

```
<Button.Triggers>
  <EventTrigger RoutedEvent="Button.MouseEnter">
```

Aufgrund des ausgelösten Trigger-Ereignisses wird das folgende *Storyboard* ausgeführt:

```
<BeginStoryboard>
  <Storyboard x:Name="Storyboard1">
```

Das *Storyboard* enthält eine *DoubleAnimation* (Verändern einer *Double*-Eigenschaft) mit einer Zeitdauer (*Duration*) von 4 Sekunden:

```
<DoubleAnimation Duration="0:0:4"
```

Ziel der Eigenschaftsänderung ist *button1*:

```
Storyboard.TargetName="button1"
```

Die zu ändernde Eigenschaft ist *Opacity*:

```
Storyboard.TargetProperty="Opacity"
```

Die Eigenschaft wird in der oben genannten Zeitdauer von 1 auf 0 geändert:

```
    From="1" To="0" />
  </Storyboard>
</BeginStoryboard>
</EventTrigger>
</Button.Triggers>
</Button>
<Rectangle Canvas.Left="700" Canvas.Top="100" Height="50"
           Stroke="Red" Width="100" />
</Canvas>
```

Das war hoffentlich nicht allzu abschreckend. Es geht teilweise auch einfacher und alternativ könnten Sie auch mit sinnvollen Werkzeugen den obigen XAML-Code erstellen.

### Animation per C#-Code realisieren

Im obigen Fall müssen Sie immer ein *Storyboard* einsetzen, um die Animation(en) zu kapseln. Etwas einfacher geht es, wenn Sie lediglich eine Animation per C#-Code realisieren wollen. In diesem Fall erstellen Sie einfach eine Instanz der gewünschten Animation (davon gibt es

je nach Zieleigenschaft unterschiedliche), parametrieren diese und starten die Animation, indem Sie diese an die *BeginAnimation*-Methode des gewünschten Controls übergeben.

**Beispiel 11.43:** Eine einfache Animation per Code definieren und ausführen

```
C#
using System.Windows;
using System.Windows.Media.Animation;
...
private void Button2_Click(object sender, RoutedEventArgs e)
{
    Instanz erstellen:
        DoubleAnimation animation = new DoubleAnimation();

    Parametrieren:
        animation.From = 1;
        animation.To = 0;

    Starten (Transparenz ändern):
        button2.BeginAnimation(OpacityProperty, animation);
}
```

Das war doch gar nicht so schwierig, oder?

### Animation per Code steuern

Vielleicht dämmert es Ihnen schon, komplexe Animationen bzw. die Zusammenfassung mehrerer Animationen als Storyboard sind kaum für die tägliche Praxis des C#-Entwicklers geeignet. Abgesehen davon, dass sie Unmengen von C#-Code erzeugen, fehlt bei vielen Animationen einfach die Vorstellungskraft. Dauernde Programmstarts zum Ausprobieren der Effekte zehren auch an den Nerven und kosten Zeit. Ganz nebenbei ist auch die Parametrierung vieler Eigenschaften mit C# eine Pein – hier kann XAML seine Vorteile deutlich ausspielen.

Viel besser ist es, die Storyboards mit einem Programm wie Microsoft Blend zu erstellen und nachträglich in die Anwendung einzufügen. Zum Starten der Animation können Sie entweder, wie bereits gezeigt, einen Trigger verwenden, oder Sie nutzen das *Storyboard* per Code. Dazu stellt die *Storyboard*-Klasse mehrere Methoden bereit, mit denen Sie die Animation gezielt kontrollieren können:

Methode	Beschreibung
<i>Begin</i>	Animationen des <i>Storyboards</i> starten.
<i>Pause</i>	Wiedergabe anhalten.
<i>Resume</i>	Wiedergabe fortsetzen.
<i>Seek</i>	Bei der Wiedergabe zu einer Position im <i>Storyboard</i> springen. Verwenden Sie einen <i>TimeSpan</i> -Wert.
<i>Stop</i>	Wiedergabe anhalten und Wiedergabeposition zurücksetzen.



**HINWEIS:** Auf das Ende der Animationen im *Storyboard* können Sie mit dessen *Completed*-Ereignis reagieren.

**Beispiel 11.44:** Als Ressource definierte Animation per Code starten

#### XAML

```
<Window x:Class="TransformationenSample.MainWindow"
...
    Title="MainWindow" Height="600" Width="1200">
```

Als Window-Ressource definieren wir ein *Storyboard*:

```
<Window.Resources>
```

Achten Sie darauf, einen *Key* zu vergeben:

```
    <Storyboard x:Key="storyboard2">
        <DoubleAnimation Duration="0:0:4" Storyboard.TargetName="button3"
                        Storyboard.TargetProperty="Width" To="300" />
    </Storyboard>
</Window.Resources>
<Canvas>
<Button Canvas.Left="900" Canvas.Top="100" Content="Button"
        Height="50" Name="button3" Width="100"
        Click="Button3_Click">
</Button>
...

```

#### C#

Zunächst den Namespace importieren:

```
...
using System.Windows.Media.Animation;
...

```

Zur Laufzeit können wir unser *Storyboard* suchen und mit der *Begin*-Methode starten:

```
private void Button3_Click(object sender, RoutedEventArgs e)
{
    Storyboard? sb = FindResource("storyboard2") as Storyboard;
    sb?.Begin();
}
```

Ist für das *Storyboard* kein *TargetName* vorgegeben, können Sie das *Storyboard* auch auf jedes andere Control anwenden, wenn Sie dieses an die *Begin*-Methode übergeben:

```
    sb?.Begin(button3);
}
```



**HINWEIS:** Ein Klick auf eine andere Taste könnte beispielsweise mit *Storyboard.Stop* die Animation anhalten.

Selbstverständlich können Sie ein in den Ressourcen abgelegtes Storyboard auch per XAML einbinden bzw. starten. In diesem Fall benötigen Sie nicht eine Zeile C#-Code, können aber die Animationen (bzw. die übergeordneten Storyboards) zentral verwalten.

**Beispiel 11.45:** Alternative zum vorhergehenden Beispiel

#### XAML

```
<Window x:Class="Animation_Bsp.MainWindow"
...
  <Window.Resources>
    <Storyboard x:Key="storyboard2">
      <DoubleAnimation Duration="0:0:4" Storyboard.TargetName="button3"
        Storyboard.TargetProperty="Width" To="300" />
    </Storyboard>
  </Window.Resources>
...
  <Button Canvas.Left="900" Canvas.Top="300" Content="Button"
    Height="23" Name="button4" Width="75" >
```

Per Trigger starten wir ein *Storyboard*:

```
<Button.Triggers>
  <EventTrigger RoutedEvent="Button.MouseEnter">
```

Hier weisen wir die Ressource zu:

```
    <BeginStoryboard Storyboard="{StaticResource storyboard2}" />
  </EventTrigger>
</Button.Triggers>
</Button>
</Canvas>
</Window>
```

Haben Sie ein *Storyboard* ohne *TargetName* (universelle Verwendung), müssen Sie diesen beim Einbinden der Ressource angeben, um auch das Ziel der Animation zu bestimmen:

**Beispiel 11.46:** Ziel bestimmen

#### XAML

```
...
  <EventTrigger RoutedEvent="Button.MouseEnter">
    <BeginStoryboard Storyboard="{StaticResource storyboard4}"
      Storyboard.TargetName="button4" />
  </EventTrigger>
...

```

### Mehrere Animationen zusammenfassen

Dass eine Animation nur auf der linearen Änderung einer Eigenschaft basiert, dürfte wohl selten der Fall sein. Meist werden mehrere Eigenschaften gleichzeitig geändert. Auch das ist mit dem *Storyboard* kein Problem, wie es das folgende Beispiel zeigt:

**Beispiel 11.47:** *Storyboard* mit drei Animationen**XAML**

```
<Window x:Class="TransformationenSample.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:TransformationenSample"
  mc:Ignorable="d"
  Title="MainWindow" Height="600" Width="1200">
  <Window.Resources>
    <Storyboard x:Key="storyboard3">
      <DoubleAnimation Duration="0:0:4"
        Storyboard.TargetProperty="Width" To="300" />
      <DoubleAnimation Duration="0:0:4"
        Storyboard.TargetProperty="RenderTransform.Angle" To="360" />
      <ColorAnimation Duration="0:0:3"
        Storyboard.TargetProperty="Foreground.Color"
        From="Red" To="Blue" />
    </Storyboard>
  </Window.Resources>
```

Das soll zu diesem Thema genügen. Auch wenn WPF im Bereich „Animationen“ fast unbegrenzte Möglichkeiten bietet, so sprechen diese doch kaum den Programmierer, sondern eher den Designer der Anwendung an. Machen Sie sich also nicht die Mühe, komplexe Storyboards per XAML oder gar C#-Quellcode zu erstellen, sondern nutzen Sie die Vorteile von Blend für Visual Studio. Erstellen Sie damit interaktiv den entsprechenden XAML-Code und fügen Sie diesen in die Ressourcen Ihrer Anwendung ein. Damit ersparen Sie sich viele graue Haare und haben mehr Zeit für die eigentliche Anwendungsentwicklung.



# Index

## Symbole

<ErrorBoundary> 975  
<PageTitle> 947  
[ApiController] 903, 915  
[Authorize] 886  
.BAML 439  
@bind 953  
[BindProperty] 846  
@code 947  
.cshtml 833, 855  
.G.CS 439  
[HttpGet] 904  
[HttpPost] 904  
[IInvokable] 966  
@model 843, 857  
.NET CLI 831, 851, 885  
.NET Core 3  
.NET-Framework 6  
.NET MAUI 16, 637, 937  
– Blazor 937  
.NET Standard 15  
??-Operator 65  
.razor 946  
@ (Razor-Syntax) 834  
[Route] 903

## A

Abbruchbedingung 751  
Abhängige Eigenschaften 537  
Abort 654  
Abs 270  
abstract 176, 177  
Abstraktion 5  
AcceptReturn 483  
Access-Key 479  
Accessor 142

Action 316, 317  
ActionResult 909  
Activated 466  
ActualHeight 479  
ActualWidth 479  
AddCors 931  
AddDays 265  
AddDefaultPolicy 931  
AddHours 265  
AddMinutes 265  
AddMonths 265  
AddRange 300  
AddYears 265  
AllowsTransparency 469  
Anfangswerte 60  
Angehängte Eigenschaften 539  
AngleX 576  
AngleY 576  
Animationen 578  
Anonyme Methoden 312, 336  
Anonyme Typen 343  
API 893  
App 432  
– Klasse 463  
App.config 759  
App.Current 432  
ApplicationCommands 553  
Application Programming Interface *Siehe* API  
appsettings.json 872, 888, 913  
App.xaml.cs 432  
Array 117, 241, 253, 375, 380  
ArrayList 299, 306  
as-Operator 69  
ASP.NET Core 825  
– MVC *Siehe* ASP.NET Core MVC  
– Projektvorlagen 828  
– Razor Pages *Siehe* Razor Pages

- ASP.NET Core Data Protection 961
- ASP.NET Core Identity 885
- ASP.NET Core MVC 842, 850
  - Action-Methoden 852
  - CRUD 869
  - Konzept 850
  - Layout-Vorlagen 863
  - Projektaufbau 851
  - Routing 853
  - Sessions *Siehe* Sessions
  - Zustandsmanagement 864
- ASP.NET Web API 893
  - Blazor 955
  - Controller 903
  - CORS 930
  - Daten aktualisieren 918
  - Daten einfügen 916
  - Daten lesen 903, 915
  - Fehlerzustand 909
  - HTTP-Statuscodes 909
  - minimale API 921
  - Paginierung 925
  - Routing 903
  - Vorlagen 897
  - XML 927
- Assemblierung 10, 26, 42
- async 684
- Asynchrone Programmierentwurfsmuster 677
- Atn 270
- Attached Properties 444, 539
- Attribute 11
- Auflistung 296
- Aufzählungszeichen 517
- Ausgabefenster 745
- Ausnahmen 769
- Ausnahmenfilter 402
- Authentifizierung 885
- AutoGenerateColumns 626
- AutoProperties 145, 400
- AutoProperty 138, 146
- Autorisierung 885
- AutoToolTipPlacement 500
- AutoToolTipPrecision 500
- await 684
  
- B**
- Background 457, 477
- BandIndex 510
- Barrier 734
- base 165
- Beep 652
- Befehlsfenster 742
- Begin 580
- BeginAnimation 580
- BeginInit 497
- BeginInvoke 672
- BeginningEdit 632
- Benannte Styles 556
- Bereichsoperator 420
- BigInteger 324
- Binary Application Markup Language 439
- BinaryFormatter 223
- BinarySearch 253
- Binding 587
- BindingBase.StringFormat 625
- BindingSource 224
- Bindungsarten 588
- BitmapFrame 497
- BitmapImage 497
- BlackoutDates 532
- Blazor 16, 933
  - <ErrorBoundary> 975
  - APIs 955
  - Datenbindung 953
  - Event-Handling 949
  - Fehlerbehandlung 974
  - File-Uploads 972
  - Hosting-Modelle 934
  - JavaScript-Interoperabilität 963
  - Komponenten 946
  - Layout-Template 938
  - .NET MAUI 937
  - Online-Status 967
  - Projektvorlagen 934, 937
  - Protected Browser Storage 961
  - Routing 946
  - Seitentitel 947
  - Tastaturreignisse 950
  - Zustandsmanagement 961
- Blazor Server 935, 937, 959
- Blazor WebAssembly 936, 940, 955
- BlockingCollection 734
- bool 58
- Boolesche Operatoren 79
- Border 515
- BorderBrush 477
- BorderThickness 515
- Boxing 73
- break 89, 116
- Breakpoints 748
- BringIntoView 528

Bubbling Events 547  
 BulletDecorator 517  
 Button 480  
 byte 57

## C

Calendar 530  
 Callback 680  
 CallerFilePath 761  
 Caller Information 760  
 CallerLineNumber 761  
 CallerMemberName 761  
 CamelCase-Notation 128  
 CancellationToken 725  
 CancellationTokenSource 725  
 CanExecute 555  
 Canvas 441, 446  
 CaretBrush 484  
 CaretIndex 483  
 case 83, 116  
 C#-Compiler 21  
 CenterOwner 468  
 CenterScreen 468  
 CenterX 575  
 CenterY 575  
 ChangeTracker 805  
 char 58  
 CheckAccess 673  
 CheckBox 486  
 class 26, 118, 127  
 ClassLoader 9  
 Clear 253  
 Client-Server-Prinzip 826  
 Clone 252, 261, 262  
 CLR 6, 9  
 CLR-Threadpool 705  
 CLS 6  
 Codefenster 35  
 Code Manager 9  
 Collapsed 478  
 Collection 296, 382, 599  
 CollectionView 610  
 CollectionViewSource 602  
 ColumnDefinitions 453  
 ColumnSpan 457  
 ComboBox 492  
 COM-Komponenten 10  
 Command 549, 550  
 CommandTarget 552  
 COM-Marshaller 9

Common Language Runtime 4, 9  
 Common Language Specification 6, 7  
 Common Type System 6, 8  
 Completed 581  
 Complex 326  
 ComponentCommands 553  
 ConcurrentBag 734  
 ConcurrentDictionary 734  
 ConcurrentQueue 734  
 ConcurrentStack 734  
 const 64  
 Constraint 308  
 Content Negotiation 927  
 ContentPresenter 567  
 ContextMenu 508  
 continue 86  
 Controller 850, 852, 898, 903  
 ControllerBase 903  
 ControlTemplate 567  
 Convert 71, 623  
 ConvertBack 623  
 Cookies 864  
 CopyTo 252, 261, 262  
 CornerRadius 515  
 CORS 930, 955  
 Cos 270  
 CountdownEvent 734  
 Cross-Origin Resource Sharing *Siehe* CORS  
 Cross-Site Request Forgery 877  
 CRUD 869, 881  
 C#-Source-Datei 23  
 CSRF *Siehe* Cross-Site Request Forgery  
 CTS 6  
 Current 297  
 CurrentItem 602, 610  
 CurrentPosition 610  
 Cursor 477

## D

DataBindings 227  
 DataGrid 529, 626  
 DataGridCheckBoxColumn 628  
 DataGridTextColumn 628  
 DataGridView 224  
 DataTemplate 604  
 Datenbankkontext 913  
 Datenstrukturen 734  
 Daten-Trigger 565  
 Datentypen 57, 114  
 Datenzugriff 92

DateOnly 269  
 DatePicker 530  
 DateTime 264  
 Datumsformatierung 275  
 Datumsfunktionen 263  
 Day 264  
 DayOfWeek 264  
 DayOfYear 264  
 DaysInMonth 266  
 Deactivated 466  
 Deadlocks 649  
 Debug 754  
   - Write 755  
   - Writelf 755  
   - WriteLinelf 755  
 Debugger 741  
 decimal 58  
 DecodePixelWidth 497  
 default 86  
 DefaultView 601  
 deferred execution 390  
 Dekrement 76  
 Delay 591  
 delegate 119, 151  
 Delegate 309, 336  
   - instanziiieren 310  
 DELETE 896, 918  
 Dependency Injection 180  
 DependencyObject 538  
 Dependency Properties 537  
 Designer 34  
 Destruktor 157, 161  
 Diagnostics 652  
 Dictionary 307  
 Dimensionsgrenzen 247  
 DirectX 428  
 Direkte Events 549  
 DispatcherUnhandledException 466  
 DisplayDate 531  
 DisplayMemberPath 604  
 DisplayMode 530  
 Dispose 163  
 Distinct 373  
 do 87, 88  
 DockPanel 441, 449  
 DockPanel.Dock 449  
 Document Object Model (DOM) 949  
 double 57  
 Duplikate 373  
 Dynamische Programmierung 319

## E

EditingCommands 553  
 EF Core Provider 778  
 Eigenschaften 28, 137  
 Eigenschaften-Fenster 35  
 Eigenschaften-Trigger 562  
 Eigenschaftsmethoden 233  
 Einzelschrittmodus 752  
 Ellipse 534  
 else 115  
 else if 82  
 Emscripten 936  
 Endeoperator 420  
 EndInvoke 672  
 EndsWith 256  
 Enter 662  
 Entity Framework 775  
 Entity Framework Core 871, 885  
 Entwicklungsumgebung 30  
 enum 90, 117  
 Enumerable 375, 380  
 Enumerationen 117  
 Ereignis 28, 119, 150, 151  
   - auslösen 153  
 Ereignismethoden 553  
 Ereignis-Trigger 564  
 Erweiterungsmethoden 344, 363  
 event 119, 151  
 EventLog 759  
 EventLogTraceListener 759  
 Events 28  
 Exception 770  
 ExceptionManager 9  
 Exit 466, 662  
 Exp 270  
 ExpandDirection 519  
 Expander 519  
 Exponentialfunktion 271  
 eXtensible Application Markup Language 430  
 Extension-Method-Syntax 347, 363

## F

Fehlerbehandlung 762  
 Fehlerklassen 764, 771  
 Fill 458  
 Filter 612  
 FindResource 593  
 float 57  
 Fluent-API 798

FontFamily 477  
 FontSize 477  
 FontStyle 477  
 FontWeight 477  
 for 87, 116  
 foreach 117, 174, 245, 307  
 Foreground 477  
 Form1.cs 34  
 Format 275, 623  
 Formulare 28  
 FromCurrentSynchronizationContext 730  
 Func 316, 317  
 Funktionen 118

## G

Garbage Collector 161  
 Generics 303, 304  
 get 142  
 GET 896  
 GetDefaultView 601  
 GetEnumerator 297, 306  
 GetLength 252, 262  
 Getter-only Auto-Property 146  
 global usings 293  
 goto 86  
 Grafikausgabe 428  
 Grafikskalierung 499  
 Grid 441, 453  
 Grid.Column 455  
 Grid.Row 455  
 GridSplitter 457  
 GroupBox 516  
 GroupName 488

## H

Haltepunkte 750  
 Hardwarebeschleunigung 428  
 Hashtable 300  
 HasValue 65  
 Header 519  
 Hidden 478  
 HorizontalAlignment 446, 456, 477  
 HorizontalContentAlignment 478  
 HorizontalOffset 523  
 HorizontalScrollBarVisibility 502  
 Hot Reload 17, 831, 940  
 Hour 264  
 HSTS *Siehe* HTTP Strict Transport Security  
 HTML-Formulare 843

HTML Helper 847  
 HTTP 826  
 - Accept-Header 927  
 - Content Negotiation 927  
 - Methoden 827, 896  
 - Statuscode 828, 909  
 HttpClient 956  
 HttpContext 844, 866  
 HttpDelete 862  
 HttpPost 862  
 HttpPut 862  
 HTTP Strict Transport Security 831

## I

IActionResult 854  
 IAsyncEnumerable<T> 691  
 IAsyncResult 679, 680, 682  
 ICollection 298  
 IComparable 218  
 IComparer 218  
 Icon 468  
 IEnumerable 297, 375  
 IEnumerator 298  
 if 82, 115  
 IIS Express 830, 832  
 JSRuntime 964  
 Image 496  
 immutable 183, 280  
 ImplicitUsings 37  
 IndentLevel 756  
 IndentSize 756  
 Index 241  
 Indexer 233, 295, 303, 334  
 IndexOf 253, 256  
 Indexprüfung 244  
 Initialisierer 375  
 Initialisierung 129  
 Initialize 252, 262  
 InitializeComponent 432  
 Init-only Setter 422  
 Inkrement 76  
 INotifyCollectionChanged 599  
 INotifyPropertyChanged 596  
 InputGestureText 505  
 Insert 256  
 Instanz 124  
 Instanzieren 128  
 int 57  
 Int16 57  
 Int32 57

Int64 57  
 IntelliSense 134  
 internal 126  
 internal protected 126  
 Interrupt 653  
 InvalidOperationException 603  
 Invoke 671, 682  
 InvokeRequired 673  
 IsAlive 655  
 IsBackGround 655  
 IsCheckable 507  
 IsChecked 486, 488, 507  
 IsCompleted 679, 712  
 IsCurrentAfterLast 601, 610  
 IsCurrentBeforeFirst 602, 610  
 IsDirectionReversed 500  
 IsEditable 493  
 IsExpanded 521, 528  
 IsIndeterminate 514  
 IsLeapYear 266  
 IsLocked 510  
 IsOpen 523  
 IsReadOnly 483  
 IsSelected 528  
 IsSelectionRangeEnabled 501  
 IsSnapToTickEnabled 501  
 IsSynchronizedWithCurrentItem 603  
 IsThreeState 486  
 Iterator 306, 713  
 IValueConverter 623

## J

JavaScript Object Notation  
   *Siehe* JSON  
 JIT-Compiler 5  
 Join 653  
 JSON 860, 868, 900, 915  
 JsonResult 860

## K

Kapselung 5, 124  
 Kartenspiel 228  
 Kartesische Koordinaten 210  
 Kestrel 830, 832  
 Kind-Elemente 455  
 Klasse 124  
   - statische 179  
 Klassendefinition 118  
 Klassenmethode 149

Kommentare 56  
 Komplexe Zahlen 210  
 Konsolenanwendung 102  
 Konstante Felder 144  
 Konstanten 57, 64  
 Konstruktor 157  
   - überladen 233  
 Kontravarianz 323  
 Kovarianz 323  
 Kurz-Operatoren 77  
 Kurzschlussausrwertung 80

## L

Label 479  
 Lambda-Ausdruck 313, 363, 369  
 Lambda Expression 336  
 Language Integrated Query 341  
 LastChildFill 450  
 launchSettings.json 832, 854  
 Layout 441  
 LazyLoading 621  
 Leerzeichen 462  
 Length 252, 256, 261, 262  
 Line 536  
 LineBreak 462  
 LINQ 341, 369, 371, 380, 382  
   - Abfrageoperatoren 348  
   - Aggregat-Operatoren 356  
   - AsEnumerable 359  
   - Count 357  
   - GroupBy 353  
   - Grundlagen 341  
   - Gruppierungsoperator 353  
   - Join 356  
   - Konvertierungsmethoden 359  
   - OrderBy 352  
   - OrderByDescending 352  
   - Projektionsoperatoren 350  
   - Restriktionsoperator 351  
   - Reverse 353  
   - Select 350  
   - SelectMany 350  
   - Sortierungsoperatoren 352  
   - Sum 357  
   - ThenBy 352  
   - ToArray 359  
   - ToDictionary 359  
   - ToList 359  
   - ToLookup 359  
   - Where 351

LINQ-Abfrageoperatoren 346  
 LINQ-Architektur 341  
 LINQ-Provider 342  
 LINQ-Syntax 346  
 LINQ to Objects 341  
 List 304, 306  
 ListBox 489  
 List-Klasse 306  
 ListView 529, 607  
 Live Shaping 613  
 Live Share 412  
 Local Storage 961  
 lock 659  
 Log 270  
 Log10 270  
 Logarithmus 271  
 Logische Operatoren 78  
 Lokale Variablen 66  
 Lokal-Fenster 743  
 long 57  
 LongRunning 730  
 LowestBreakIteration 712

## M

MainWindow.xaml 433  
 MainWindow.xaml.cs 433  
 ManualResetEventSlim 734  
 Margin 445  
 Mass Assignment 876, 916  
 Matrix 233  
 MatrixTransform 574  
 Matrizen 238  
 Max 270  
 MaxHeight 445, 478  
 MaxLength 483  
 MaxLines 483  
 MaxWidth 445, 478  
 Media 652  
 MediaCommands 553  
 Menu 504  
 Menü  
 – Grafiken 506  
 – Tastenkürzel 505  
 MenuItem 504  
 Menüleiste 503  
 Messwertliste 366  
 Metadaten 11  
 Methoden 28, 93, 118, 147  
 – generische 305  
 – statische 149

– überladen 110, 233  
 – überladene 148  
 Methodenzeiger 309  
 MethodImpl 667  
 Methods 28  
 Microsoft Intermediate Language Code 5  
 Min 270  
 MinHeight 445, 478  
 MinLines 483  
 Minute 264  
 MinWidth 445, 478  
 Modale Dialoge 480  
 Model 850, 857  
 Model Binding 846, 907  
 Monitor 662  
 Mono Runtime 638  
 Month 264  
 MoveCurrentTo 611  
 MoveCurrentToFirst 602, 611  
 MoveCurrentToLast 601, 611  
 MoveCurrentToNext 601, 611  
 MoveCurrentToPosition 611  
 MoveCurrentToPrevious 602, 611  
 MoveNext 297  
 MSIL-Code 5  
 Multi-Platform App UI 637  
 MultipleRange 531  
 Multitasking 648  
 Multithreading 13, 648  
 Mutex 666  
 MVC *Siehe* ASP.NET Core MVC  
 MVVM 428

## N

Name-Attribut 435  
 Namespace 9, 129  
 NavigationCommands 553  
 new 119, 157, 242  
 Next 377  
 non-destructive mutation 188  
 Now 266  
 null 64, 131  
 Nullable Type 64  
 null-coalescing operator 65  
 Null-Coalescing-Zuweisungsoperator 418  
 NULL-Zusammenführungsoperator 65  
 Nutzeradministration 889

## O

object 58  
   – Datentyp 63  
 Objekt 124  
 Objektbaum 223  
 Objekte 119  
 Objektinitialisierer 160, 343, 344  
 ObservableCollection 599  
 ODER 80  
 OnAfterRenderAsync 970  
 OneTime 588  
 OneWay 588  
 OneWayToSource 588  
 OnExplicitShutdown 466  
 OnGet 844, 846  
 OnInitializedAsync 958, 964  
 OnLastWindowClose 466  
 Online-Status 967  
 OnMainWindowClose 466  
 OnPost 844, 846  
 OnStartUp 465  
 OOP 228  
 Opacity 469, 478  
 OpenAPI 898, 924  
 OpenFileDialog 498  
 Open Source 3  
 Operatoren 74, 115  
   – arithmetische 75  
   – boolesche 79  
   – logische 78  
 Operatorenüberladung 210  
 Optionale Parameter 322  
 orderby 381  
 Orientation 447, 500, 502  
 out 98  
 OverflowMode 511  
 Overposting *Siehe* Mass Assignment  
 override 165

## P

Padding 445, 478  
 PadLeft 256  
 PadRight 256  
 Paket-Manager-Konsole 616  
 PAP 102  
 Parallel.For 709  
 Parallel.ForEach 713  
 Parallel.Invoke 706  
 Parallel LINQ 735

ParallelLoopResult 712  
 Parallel-Programmierung 701  
 Parameterübergabe 97, 98  
 Parent 478  
 Parse 71, 266  
 Pascal-Notation 128  
 PasswordBox 483, 485  
 PATCH 896  
 Pattern Matching 405  
 Pause 580  
 Pi 270  
 Placement 523  
 PlacementRectangle 523  
 PlacementTarget 523  
 PLINQ 360, 735  
 Polarkoordinaten 210  
 Polling 678  
 Polymorphes Verhalten 172  
 Polymorphie 5, 125, 164, 174  
 Popup 522  
 Portieren 113  
 POST 896, 916  
 Postman 902  
 Potenz 271  
 Pow 270  
 PreferFairness 730  
 Priority 655  
 private 126  
 private protected 410  
 Procedure-Step 748  
 Process 694, 698  
 Process.Start 699  
 ProcessThread 694  
 Program.cs 830, 898, 914, 921, 937  
 Programm starten 697  
 ProgressBar 514  
 Progressive Web Application *Siehe* PWA  
 Projektmappen-Explorer 33  
 Projekttyp 33  
 Properties 28, 190  
 Property-Accessoren 142  
 PropertyChanged 590  
 protected 126  
 Prozeduren 118  
 Prozedurschritt 753  
 Prozesse 694  
 public 126  
 Pulse 662, 663  
 PulseAll 662, 663  
 PUT 896, 918  
 PWA 940, 941

## Q

Query-Expression-Syntax 347, 363  
 Queue 304, 307  
 QueueUserWorkItem 656

## R

Racing 650  
 RadioButton 488  
 Rahmenbreite 515  
 Random 230, 272, 273, 377  
 Range 376, 420  
 Rank 252, 261, 262  
 Razor-Komponenten *Siehe* Blazor:Komponenten  
 Razor Pages 829, 833
 

- anlegen 833
- CRUD 881
- foreach 836
- Formulare 843
- Layout-Vorlagen 838
- Modelle 842

 Razor-Syntax 834, 947  
 ReadLine 26  
 Records 183, 422  
 Rectangle 535  
 ref 97  
 Referenzieren 128  
 Referenztyp 63, 255  
 Reflexion 11  
 ReleaseMutex 666  
 Remove 256  
 RenderBody 840  
 RenderSectionAsync 840  
 Repeat 377  
 RepeatButton 480  
 Replace 256  
 Representational State Transfer *Siehe* REST  
 Reset 297  
 Resources 478  
 Ressourcen 539  
 REST 894
 

- HTTP-Methoden 896
- Prinzipien 894
- URIs 895

 Resume 580  
 return 86, 118, 306  
 RotateTransform 574  
 Round 270  
 Routed Events 546  
 RowDefinitions 453

RowDetailsTemplate 630  
 RowDetailsVisibilityMode 630, 632  
 RowSpan 457  
 Rückrufmethode 678  
 Rücksprung 753

## S

Same-Origin Policy 930  
 ScaleTransform 574  
 ScaleX 575  
 ScaleY 575  
 Schaltjahr 266  
 Schleifen 116  
 Schleifenabbruch 71  
 Schleifenanweisungen 87  
 Schlüsselwörter 55, 292  
 ScrollBar 502  
 ScrollView 502  
 sealed 177  
 Second 264  
 Security Engine 9  
 Seek 580  
 select 381  
 SelectedDate 531  
 SelectedItem 528  
 SelectedItemChanged 528  
 SelectedItems 491  
 SelectionBrush 484  
 SelectionMode 489  
 Semaphore 668  
 SemaphoreSlim 734  
 Separator 504  
 Serialisierung 12  
 Serializable 224  
 SessionEnding 466  
 Sessions 865
 

- JSON 868
- lesen 866
- schreiben 866

 Session Storage 961, 964  
 set 138, 142  
 Shared-Methoden 233  
 short 57  
 Show 469  
 ShowDialog 469  
 Shutdown 466  
 Sign 270  
 SignalR 936  
 Sin 270  
 Single-Page Application 827, 893

SingleRange 531  
 Single-Step 748  
 Skalieren mit ScaleTransform 575  
 SkewTransform 574  
 Sleep 654  
 Slider 500  
 Sort 218, 253  
 SortDescriptions 612  
 SortedList 307  
 SortedSet 328  
 Sortieren 380  
 Source 496  
 SPA *Siehe* Single-Page Application  
 SpellCheck.IsEnabled 483  
 Sperrmechanismen 657  
 SpinLock 734  
 SpinWait 734  
 Split 256  
 Sqr 270  
 Stack 304  
 StackPanel 441, 447  
 StartInfo 698  
 StartsWidth 256  
 Startup 465, 466  
 StartupEventArgs 465  
 StartupUri 432, 463  
 StateHasChanged 970  
 static 118, 179  
 StaticResource 542, 559  
 Statische Klassen 179  
 Statische Methode 149  
 Statischer Konstruktor 160  
 StatusBar 512  
 StatusBarItem 512  
 Steuerelemente 28  
 Stop 580  
 StoryBoard 578  
 Stretch 499  
 StretchDirection 499  
 string 58  
 String 255  
 StringAddition 280  
 struct 91, 117  
 Strukturen 117  
 Strukturvariable 93  
 Style 478, 558  
 Style anpassen 559, 561  
 Style ersetzen 559  
 Styles vererben 560  
 Subklassen 165, 166  
 SubString 256

Swagger UI 900, 924  
 switch 83, 116  
 System 57  
 System.Collections.Concurrent 734  
 System.Nullable 65  
 System.Object 175  
 Systemressourcen 545  
 System.Threading 651, 704  
 System.Threading.Tasks 704

## T

TabControl 521  
 TabIndex 478  
 TabPanel 441  
 Tag 478  
 Tag Helper 849  
 Tan 270  
 Target 479  
 Task
 

- Canceled 729
- ContinueWith 721, 730
- Created 729
- Datenübergabe 717
- Faulted 729
- Fehlerbehandlung 728
- IsCanceled 729
- IsCompleted 729
- IsFaulted 729
- Klasse 714
- RanToCompletion 729
- Result 721
- return 724
- Rückgabewerte 720
- Running 729
- starten 715
- Status 729
- TaskCreationOptions 730
- Task-Ende 730
- Task-Id 729
- User Interface 730
- Verarbeitung abbrechen 723
- Wait 718
- WaitAll 719
- WaitingForActivation 729
- WaitingForChildrenToComplete 729
- WaitingToRun 729
- weitere Eigenschaften 729

 Task<> 688  
 Task.Factory.StartNew 714  
 Task Parallel Library 701

TaskScheduler 730  
 Tastaturereignisse 950  
 Template 566  
 Textausrichtung 463  
 TextBlock 459  
 TextBox 483  
 Textformatierungen 460  
 TextFormattingMode 472  
 TextWriterTraceListener 758  
 Thin Client 194  
 Thread 651, 653  
 ThreadInterruptedException 653  
 ThreadPool 656  
 Threads 694  
 Thread Service 9  
 Threadsicher 670  
 Threadsichere Collections 734  
 ThreadState 655  
 Throw 765, 771  
 ThrowIfCancellationRequested 725  
 TickFrequency 501  
 TickPlacement 501  
 TimeOnly 269  
 Timer-Threads 675  
 TimeSpan-Klasse 280  
 Title 468  
 ToArray 373, 381  
 ToCharArray 256  
 Today 266  
 ToggleButton 480  
 ToLongDateString 265  
 ToLongTimeString 265  
 ToLower 256  
 Toolbar 509  
 ToolbarTray 509, 510  
 Toolbox 34  
 ToolTip 478  
 ToShortDateString 265  
 ToShortTimeString 265  
 ToString 70, 274  
 ToUpper 256  
 Trace 754, 758  
 TraceListener 758  
 TrackBar 280  
 Transformationen 573  
 TransformGroup 576  
 TranslateTransform 574  
 Transparenz 469  
 TreeView 525  
 Trefferanzahl 752  
 Trigger 561

Trim 256  
 try 116  
 try-catch 763  
 TryEnter 662, 665  
 try-finally 767  
 Tunneling Events 546  
 Tuple 327  
 TwoWay 588  
 Typecasting 332  
 Typinferenz 66, 343, 363  
 Typ-Styles 558  
 Typsuffixe 60

## U

Überladene Methoden 148  
 Überwachungsfenster 744  
 Uhr anzeigen 267  
 UI-Virtualisierung 628  
 Unboxing 73  
 UND 80  
 Unicode 61  
 Uniform 458  
 UniformGrid 441, 452  
 UniformToFill 459  
 UnIndent 756  
 UpdateSourceTrigger 590  
 Uri 544  
 URI 895  
 UseCors 931  
 using 25, 129, 292

## V

Value 500  
 var 66  
 Variablen 57  
 Variablentypen 57  
 VB 113  
 Verarbeitungsstatus 712  
 Vererbung 125  
 Verformen mit SkewTransform 575  
 Vergleichsoperatoren 78  
 Verschieben mit TranslateTransform 576  
 VerticalAlignment 446, 456, 477  
 VerticalContentAlignment 478  
 VerticalOffset 523  
 VerticalScrollBarVisibility 502  
 Verweistypen 58  
 Verzweigungen 115  
 View 850, 855

ViewBox 441, 458  
 ViewData 839, 863, 867  
 ViewResult 860  
 VirtualizingStackPanel 628  
 Visibility 478  
 Visual Studio  
 - ASP.NET Core 828  
 - Blazor 934  
 Visual Studio 20  
 Visual Studio Enterprise 20  
 Visual Studio Professional 20  
 void 95

## W

Wait 662, 663  
 WaitOne 666, 668  
 WASM *Siehe* WebAssembly  
 Web API *Siehe* API  
 WebAssembly 936  
 WebSockets 936  
 Werkzeugkasten 34  
 Wertetypen 58  
 where 308  
 while 87, 88, 116  
 Wiederholmuster 379  
 Wiederverwendbarkeit 125  
 Windows Presentation Foundation  
 427  
 WindowStartupLocation 468  
 WindowStyle 468, 469  
 Winkel 271  
 with 189  
 work stealing 705  
 WPF 427, 673  
 - Anwendung beenden 466  
 - Applikationstypen 437  
 - Eigenschaften 477  
 - Ereignishandler 434  
 - Ereignismodell 545  
 - Height 444

- Kommandozeilenparameter 465  
 - Left 444  
 - Maßangaben 444  
 - Startobjekt festlegen 463  
 - Style-System 555  
 - Top 444  
 - Width 444  
 - Window-Klasse 467  
 - Zielplattformen 437  
 WPF-Programm 463  
 WPF-Wertkonvertierer 623  
 Wrap 460  
 WrapPanel 441, 451  
 WrapWithOverflow 460  
 Writelf 754  
 WriteLine 26, 754  
 Wurzel 271

## X

Xamarin 16  
 XAML 430  
 x:Class 433  
 XML 927  
 xml:space 462  
 XOR 80

## Y

Year 264  
 yield 306, 335

## Z

Zahlenformatierung 274  
 Zeilenumbrüche 462  
 Zeitfunktionen 263  
 Zeitmessung 281  
 Zufallszahlen 272, 273, 377  
 Zustandsmanagement 961  
 Zuweisungsoperatoren 77