Visual C# 2010 von Andreas Kühnel
Das umfassende Handbuch

Visual C# 2010
geb., mit DVD
1295 S., 49,90 Euro
Rheinwerk Computing
ISBN 978-3-8362-1552-7

▲

17.4 Einführung in XAML ▼

XAML ist eine deklarative Programmiersprache, deren Wurzeln auf XML zurückzuführen sind. XAML unterliegt damit auch denselben strengen Regeln wie XML:

- Elemente werden durch Tags beschrieben.
- Auf jedes Start-Tag muss zwingend ein End-Tag folgen.
- Zwischen dem Start- und dem End-Tag wird das Element beschrieben.
- Tags können ineinander verschachtelt werden. Eine diagonale Verschachtelung ist nicht erlaubt.
- Die Groß-/Kleinschreibung muss beachtet werden.

Anmerkung

Auf eine detaillierte Einführung in XML soll an dieser Stelle verzichtet werden. Sollten Sie dennoch noch nicht mit XML Bekanntschaft gemacht haben, sollten Sie Abschnitt 16.1, »XML-Dokumente«, über wohlgeformten XML-Code lesen.

▲

17.4.1 Struktur einer XAML-Datei ▼▲

Sehen wir uns zunächst noch einmal den XAML-Code an, den Visual Studio 2010 beim Erstellen eines neuen Fensters erzeugt.

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
  <Grid>
  </Grid>
</Window>
```

Jede XAML-Datei besitzt ein Wurzelelement, das alle anderen Elemente einschließt. Bei einer WPF-Anwendung handelt es sich um `Window`, bei einer WPF-Browseranwendung um `Page`. Der Parameter `x:Class` gibt die Klassendefinition an, die vom XAML-Code beschrieben wird. Natürlich spielt hier intern das Konzept partieller Klasse eine tragende Rolle. Sie müssen beachten, dass die einfache Angabe des Klassenbezeichners nicht ausreicht. Zusätzlich ist die Angabe des Namespace erforderlich.

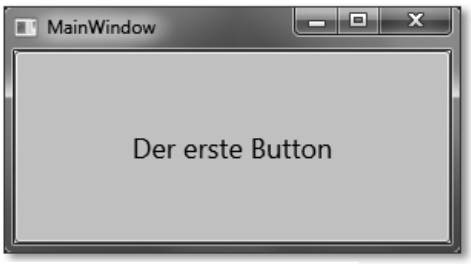
Damit XAML einwandfrei funktioniert, werden zwei Namespaces angeführt die, zur deklarativen Festlegung der Objekte benötigt werden. Mit `Title` wird die Zeichenfolge beschrieben, die in der Titelleiste des Fensters angezeigt wird. `Height` und `Width` legen die Gesamthöhe und -breite des Fensters fest.

Innerhalb des `Window`-Wurzelelements werden alle Elemente angegeben, aus denen sich das Fenster zusammensetzt. Üblicherweise fügt man aber zuerst ein Containerelement ein, in dem die anderen Elemente wie `Button` oder `TextBox` positioniert werden. Typische Containerelemente sind `Grid` oder `StackPanel`. Diese zeichnen sich dadurch aus, dass sie über eine Eigenschaft `Children` verfügen, die eine Collection von `UIElement`-Objekten verwaltet.

Im folgenden Codefragment sehen Sie das Stammelement `Window` nebst dem untergeordneten Container vom Typ `Grid`. Dieser enthält ein `Button`-Element.

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
        xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
        Title="MainWindow" Height="163" Width="300">
  <Grid>
    <Button FontSize="18" Background="LightGray"
          Name="btnButton1">Der erste Button</Button>
  </Grid>
</Window>
```

Die Abmessungen der Schaltfläche sind nicht angegeben. Es liegt in der Natur des umgebenden `Grid`-Containers, dass der `Button` dann den gesamten im Container zur Verfügung gestellten Platz für sich beansprucht.



Hier klicken, um das Bild zu vergrößern

Abbildung 17.3 Eine Schaltfläche in einem Grid-Container

▲

17.4.2 XAML-Elemente ▼▲

Mit XAML wird die Benutzeroberfläche beschrieben. Wie im Listing oben schon gezeigt wurde, lässt sich eine Schaltfläche wie folgt darstellen:

```
<Button FontSize="18" Background="LightGray" Name="btnButton1">
  Der erste Button </Button>
```

Das Element `Button` entspricht einer WPF-Klasse, die sich im Namespace `System.Windows.Control` befindet. Die Angabe im XAML-Code bewirkt die Instanziierung des Elements über den parameterlosen Konstruktor der Klasse. `Content` ist eine Eigenschaft der neuen Instanz und dient zur Beschriftung der Schaltfläche. Alternativ können Sie die Schaltfläche auch im Programmcode erzeugen:

```
Button btn = new Button();
btn.Content = "Button1";
this.AddChild(btn);
```

Mit der letzten Anweisung wird die Schaltfläche ihrem übergeordneten Container zugeordnet. Es sei hier davon ausgegangen, dass es sich dabei um das Window-Objekt handelt.

▲

17.4.3 Eigenschaften eines XAML-Elements festlegen ▼▲

Attribut-Schreibweise

Jedes Objekt hat Eigenschaften, die im XAML-Code durch Attribute angegeben werden. Beabsichtigen Sie, neben der Beschriftung auch die Breite und die Höhe einer Schaltfläche festzulegen, müssen Sie die Eigenschaften `Height` und `Width` als Attribute angeben:

```
<Button Height="50" Width="100">Mein erster Button</Button>
```

Allen Attributen werden Werte grundsätzlich immer als `String` übergeben. Der Zeichenfolgenwert wird von einem Konverter anschließend in den von der Eigenschaft beschriebenen Datentyp umgewandelt. Bei `Height` und `Width` ist das die Umwandlung in den Datentyp `Double`.

Nicht immer muss der Zieldatentyp so einfach sein. Legen Sie zum Beispiel die Hintergrundfarbe `Background` fest, verbirgt sich dahinter die Konvertierung in den schon sehr komplexen Typ `Brush`.

Eigenschaft-Element-Syntax

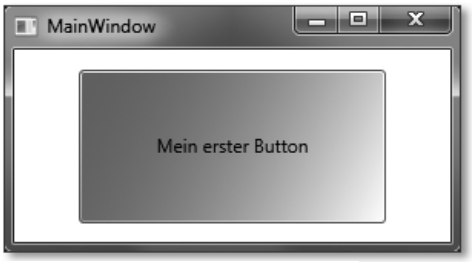
Die Attribut-Schreibweise ist einerseits kompakt, birgt aber den Nachteil, dass einem Attribut nur Zeichenfolgen zugewiesen werden können. Eigenschaften können aber auch komplexer sein. Nehmen wir exemplarisch die Eigenschaft `Background`, die die Hintergrundfarbe beschreibt. Soll diese Blau sein, ist die Attributschreibweise vollkommen ausreichend:

```
<Button Background="Blue"></Button>
```

Was ist aber, wenn die Hintergrundfarbe nicht durch eine konkrete Farbe, sondern einen komplexen Farbverlauf beschrieben werden soll? Dafür sind entsprechende Objekte notwendig, die von WPF bereitgestellt werden. In solchen Fällen kommt die sogenannte *Eigenschaft-Element-Syntax* ins Spiel. Bei dieser Schreibweise wird zuerst der Typ des Elements genannt und dahinter, durch einen Punkt getrennt, die Eigenschaft.

```
<Button Height="100" Width="200">
  <Button.Background>
    <LinearGradientBrush>
      <GradientStop Color="Red" Offset="0.0" />
      <GradientStop Color="White" Offset="1.0" />
    </LinearGradientBrush>
  </Button.Background>
    Mein erster Button
</Button>
```

Das Codefragment beschreibt mit einem Objekt vom Typ `LinearGradientBrush` den linearen Farbverlauf des Hintergrunds einer Schaltfläche. Die beiden `GradientStop`-Objekte geben die Position und die Farben des Farbverlaufs an.



Hier klicken, um das Bild zu vergrößern

Abbildung 17.4 Button mit linearem Farbverlauf

Die Eigenschaft-Element-Schreibweise können Sie auch bei einfachen Eigenschaften wie zum Beispiel `Height` und `Width` verwenden. Sie geben dann den gewünschten Wert zwischen den beiden Tags an:

```
<Button>
  <Button.Height>
    50
  </Button.Height>
  <Button.Width>
    100
  </Button.Width>
  Mein erster Button
</Button>
```

Inhaltseigenschaften

Sie kennen nun die Attribut-Schreibweise und die Eigenschaft-Element-Syntax. Doch wie verhält es sich mit der Beschriftung einer Schaltfläche? Zur Erinnerung, die Beschriftung wird zwischen den ein- und ausleitenden Element-Tags angegeben, beispielsweise so:

```
<Button>Mein erster Button</Button>
```

Tatsächlich wird natürlich auch die Beschriftung durch eine spezifische Eigenschaft vertreten. Beim `Button` handelt es sich um `Content`. Sie können die Beschriftung daher auch als Attribut angeben:

```
<Button Content="Mein erster Button"></Button>
```

Es steht jetzt die Frage im Raum, warum wir in der erstgenannten Form nicht ausdrücklich die Eigenschaft angeben müssen, also:

```
<Button>
  <Button.Content>
    Mein erster Button
  </Button.Content>
</Button>
```

Die Antwort lautet: Weil `Content` als sogenannte *Inhaltseigenschaft* definiert ist. Bei Inhaltseigenschaften kann auf die explizite Angabe in der Eigenschaft-Element-Schreibweise verzichtet werden. Natürlich ist eine Inhaltseigenschaft nicht zwangsläufig an die Bezeichnung `Content` gebunden, sondern kann beliebig sein.

Inhaltseigenschaften sind bei allen Steuerelementen möglich, die von der Klasse `ContentControl` abgeleitet sind. Hintergrund ist, dass `ContentControl` mit dem Attribut `DefaultProperty` verknüpft ist, mit dem die Standardeigenschaft einer Komponente angegeben wird.

```
[DefaultProperty("Content")]
public class ContentControl : Control, IAddChild
```

Die Klasse `Button` wird, wie viele andere übrigens auch, von `ContentControl` abgeleitet und erbt damit auch die Vorgabe des Attributs.

Das Besondere einer Inhaltseigenschaft ist der Datentyp, denn sie beschreibt: `Object`. Dadurch lassen sich beliebige Objekte der Inhaltseigenschaft eines Elements zuordnen. Hinsichtlich unseres `Buttons` bedeutet das, dass Sie weitergehende Möglichkeiten erhalten, der Eigenschaft auch beliebige Objekte zu übergeben.

```
<Button>
   <Rectangle Height="30" Width="100" Fill="DarkBlue"></Rectangle>
</Button>
```

Ein anderes Beispiel bietet uns die Klasse `StackPanel`. Dieses Element beschreibt ein Containersteuerelement, ähnlich wie auch `Grid`. Die Klasse `StackPanel` erbt nicht direkt von `ContentControl`, ist aber von der Basis `Panel` mit der Inhaltseigenschaft `Children` ausgestattet. Im Gegensatz zur Eigenschaft `Content`, die nur einmal pro Steuerelement genutzt werden kann, lassen sich mit `Children` auch mehrere Inhaltselemente hinzufügen.

```
<StackPanel>
  <Button>OK</Button>
  <Button>Übernehmen</Button>
  <Button>Abbrechen</Button>
</StackPanel>
```

Entwickeln Sie eine WPF-Anwendung, wird Ihre Haupttätigkeit die XAML-Codierung sein. Nichtsdestotrotz können Sie alles, was Sie im XAML-Code schreiben, auch mit C#-Code erreichen. Um beispielsweise den XAML-Code nur durch C#-Code abzubilden, sind die folgenden Anweisungen notwendig:

```
StackPanel stackPanel = new StackPanel();
Button btnOK = new Button();
btnOK.Content = "OK";
Button btnUebernehmen = new Button();
btnUebernehmen.Content = "Übernehmen";
Button btnAbbrechen = new Button();
btnAbbrechen.Content = "Abbrechen";
stackPanel.Children.Add(btnOK);
stackPanel.Children.Add(btnUebernehmen);
stackPanel.Children.Add(btnAbbrechen);
this.AddChild(stackPanel);
```

Dieses Beispiel zeigt deutlich, dass XAML deutlich kürzer und kompakter erscheint als die Beschreibung der Oberfläche mittels Programmcode.

▲

17.4.4 Typkonvertierung ▼▲

Die im XAML-Code definierten Elemente entsprechen einer Klasse, die Attribute einer Eigenschaft. Dabei wird den Attributen ein Wert immer als Zeichenfolge übergeben. Die meisten Eigenschaften sind aber nicht vom Datentyp `String`, und müssen daher in den tatsächlichen Datentyp konvertiert werden.

Betrachten wir dazu das einfache Beispiel der Eigenschaften `Width` und `Height` eines Buttons.

```
<Button Height="100" Width="200">Beenden</Button>
```

Beide Eigenschaften sind vom Typ `Double`. Die Folge ist, dass die angegebenen Werte in die Fließkommazahlen 100,0 bzw. 200,0 konvertiert werden müssen. In XAML sind viele solcher Konvertierungen definiert. Meist werden Sie aber von diesem intern ablaufenden Vorgang keine Notiz nehmen.

Das obige Beispiel zeigt dabei nur einen recht einfachen Fall. Einen komplexeren Fall haben wir bereits weiter oben gesehen, als wir der Eigenschaft `Background` eine Farbe zugeordnet haben.

```
<Button Background="Blue"></Button>
```

Tatsächlich erwartet `Background` die Angabe eines Objekts vom Typ `Brush`. Da die Klasse `Brush` abstrakt definiert ist, muss in eine der Ableitungen konvertiert werden. Da wir es mit einer monochromen Farbe zu tun haben, wird es sich um eine Konvertierung in den Typ `SolidColorBrush` handeln. Dieser Sachverhalt wird deutlich, wenn wir uns die Eigenschaft-Element-Schreibweise ansehen:

```
<Button>
  <Button.Background>
    <SolidColorBrush Color="Blue" />
  </Button.Background>
</Button>
```

Hier muss natürlich auch noch der Farbwert `Blue` konvertiert werden, da der Typ einer Farbe nicht vom Typ `String`, sondern vom Typ `Color` ist.

Die Technik der Typkonvertierung wird in XAML auch benutzt, um mehrere ähnliche Eigenschaften zusammenzufassen, um auf diese Weise den XAML-Code kompakter zu gestalten. `Margin` gehört zu dieser Gruppe von Eigenschaften. `Margin` ist vom Typ `Thickness` und legt den äußeren Rand eines Elements fest. In der Eigenschaft-Element-Schreibweise können Sie `Margin` wie folgt bei einem `StackPanel` einsetzen:

```
<StackPanel>
  <StackPanel.Margin>
    <Thickness Left="100" Top="30" Right="50" Bottom="10" />
  </StackPanel.Margin>
  <Button>Button1</Button>
</StackPanel>
```

Typkonvertierung ist hierbei noch nicht im Spiel. Sie können aber die Eigenschaft `Margin` auch wie folgt festlegen:

```
<StackPanel Margin="100,30,50,10">
  <Button>Button1</Button>
</StackPanel>
```

Diese Form des Einsatzes von `Margin` setzt voraus, dass die Zeichenfolge von einem Typkonvertierer passend umgesetzt wird. Natürlich ist die Eigenschaft mit der erforderlichen Verhaltensweise ausgestattet. Der Nachteil dieser syntaktischen Umsetzung ist, dass solche Zeichenfolgen zu einem schwer lesbaren Code führen. Im Zweifelsfall ist daher eher zu empfehlen, auf die etwas aufwendigere Eigenschaft-Element-Schreibweise zurückzugreifen.

▲

17.4.5 Markup-Erweiterungen (Markup Extensions) ▼▲

Die Eigenschaften eines Elements werden in XAML durch Attribute beschrieben, die als Zeichenfolge angegeben und passend ausgewertet werden. Dieses Konzept wird auch in XML benutzt, hat aber seine Grenzen, wenn ein Eigenschaftswert durch die Referenz auf ein Objekt beschrieben werden muss. An dieser Stelle kommen Markup-Erweiterungen ins Spiel, um eine Lösung für diese Problematik zu bieten. Im Grunde genommen stellen auch Typkonvertierer einen Weg dar, um an Objekte zu binden. Der Unterschied zwischen einem Typkonvertierer und einer Markup-Erweiterung ist aber, dass Typkonvertierer implizit im Hintergrund agieren und nach einer festgelegten Regel arbeiten, während Markup-Erweiterungen allgemein verwendbar sind.

Markup-Erweiterungen sind Attributwerte, die in geschweiften Klammen eingeschlossen angegeben sind. Im folgenden Beispielcode wird das Konzept der Markup-Erweiterung dazu benutzt, den Inhalt der Textbox `txtUnten` an den Inhalt der `TextBox txtOben` zu binden. Das hat zur Folge, dass zur Laufzeit eine Eingabe in die obere `TextBox` sofort von der unteren `TextBox` übernommen wird.

```
<StackPanel>
  <TextBox Name="txtOben"></TextBox>
  <TextBox Name="txtUnten"
           Text="{Binding ElementName=txtOben, Path=Text}">
  </TextBox>
</StackPanel>
```

Der XAML-Reader erkennt an den geschweiften Klammern, dass der Attributwert weder ein Literal noch ein über einen Typkonvertierer umwandelbarer Wert ist. Die Markup-Erweiterung, die im Beispielcode verwendet wird, ist `Binding`. Dieser Typ gehört zum Namespace `System.Windows.Data`. Der Parameter `ElementName` gibt das Element an, an das gebunden wird, `Path` beschreibt die Eigenschaft des Quellelements, aus dem der Wert bezogen werden soll. Beachten Sie, dass zwischen den Parametern ein Komma gesetzt werden muss.

Werden den Parametern einer Markup-Erweiterung mit dem Zuweisungsoperator `=` Werte übergeben (siehe `ElementName` und `Path` im Beispiel), wird die Markup-Erweiterung mit dem parameterlosen Konstruktor instanziiert. Die Parameterwerte werden den gleichnamigen Eigenschaften des Objekts übergeben.

Enthalten die Parameter hingegen kein `=`-Zeichen, werden sie an den Konstruktor der Markup-Erweiterung weitergeleitet. Selbstverständlich müssen dann Anzahl und Typ mit den entsprechenden Werten des Konstruktors übereinstimmen.

Markup-Erweiterungen werden nicht nur durch Binding beschrieben. `StaticResource`, `DynamicResource` oder auch `x:` sind einige weitere wichtige Erweiterungen, die uns noch beschäftigen werden. Zudem werden Sie noch sehen, dass auch mehrfach verschachtelte Markup-Erweiterungen möglich sind.

Eigenschaft-Element-Schreibweise

Die weiter oben beschriebene Eigenschaft-Element-Schreibweise ist auch im Zusammenhang mit Markup-Erweiterungen möglich. Das folgende Listing zeigt das Beispiel von oben noch einmal, ist aber in die Elementschreibweise umgeschrieben.

```
<StackPanel>
  <TextBox Name="txtOben"></TextBox>
  <TextBox Name="txtUnten">
    <TextBox.Text>
      <Binding ElementName="txtOben" Path="Text" />
    </TextBox.Text>
  </TextBox>
</StackPanel>
```

Markup-Erweiterungen mittels C#-Code

Markup-Erweiterungen lassen sich nicht nur deklarativ festlegen, sondern auch durch Code beschreiben. Wir wollen uns das am Beispiel der beiden

Textboxen ansehen.

```
// ----------------------------------------------------------
// Beispiel: ...\Kapitel 17\MarkupExtension
// ----------------------------------------------------------
public partial class Window1 : Window {
  public Window1() {
    InitializeComponent();
    // StackPanel
    StackPanel panel = new StackPanel();
    this.AddChild(panel);
    // Textbox oben
    TextBox txtOben = new TextBox();
    panel.Children.Add(txtOben);
    // Textbox unten
    TextBox txtUnten = new TextBox();
    panel.Children.Add(txtUnten);
    // Bindung als Markup-Erweiterung
    Binding binding = new Binding("Text");
    binding.Source = txtOben;
    txtUnten.SetBinding(TextBox.TextProperty, binding);
  }
}
```

Der Code ist im Konstruktor des Fensters nach dem Aufruf der Methode `InitializeComponent` implementiert. Zuerst wird das `StackPanel`-Objekt erzeugt und dem `Window` durch Aufruf von `AddChild` als untergeordnetes Element übergeben. Die beiden Eingabefelder hingegen werden nach der Instanziierung zu untergeordneten Elementen des `StackPanels`. Dieser Container liefert durch Aufruf der Eigenschaft `Children` die Referenz auf ein `UIElementCollection`-Objekt, dem mit der Methode `Add` die Textboxen hinzugefügt werden.

Die Bindung der unteren an die obere `TextBox` erfordert ein `Binding`-Objekt, dessen Konstruktor Sie die Eigenschaft des Quellelements bekanntgeben, an die gebunden werden soll. Das Quellelement selbst wird der Eigenschaft `Source` genannt. Aktiviert wird die Bindung der unteren an die obere `TextBox` mit der Methode `SetBinding`. Während das zweite Argument keiner besonderen Erklärung bedarf, erscheint das erste ungewöhnlich. Hierbei handelt es sich um die Angabe einer abhängigen Eigenschaft (*Dependency Property*). Später in diesem Kapitel werden wir uns diesem Thema noch widmen.

▲

17.4.6 Namespaces ▼▲

XAML-Namespaces

Bei den XAML-Namespaces handelt es sich um eine Technologie, um Elemente eindeutig zuordnen zu können. Beispielsweise könnte das Element `<Button>` in einem XAML-Dokument zwei unterschiedliche Schaltflächen beschreiben. Erst die Zuordnung zu einem Namespace gestattet die eindeutige Identifizierbarkeit. Prinzipiell kommt den XAML-Namespaces somit die gleiche Bedeutung zu wie den Namespaces in .NET. Im Wurzelelement `Window` einer XAML-Datei werden sofort zwei Namespaces verfügbar gemacht:

```
xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
```

Der erste Namespace ist als Standard-Namespace definiert. Standard-Namespaces weisen kein Präfix auf. Alle diesem Namespace zugeordneten Klassen können direkt ohne Präfixangabe in XAML verwendet werden. Dazu gehören beispielsweise die Elemente `<StackPanel>`, `<Button>` oder `<TextBox>`.

Da nur ein Namespace ohne Präfix angegeben werden darf, muss allen anderen Namespaces ein Präfix zugeordnet werden. Die zweite Namespace-Angabe definiert ein solches mit `x`. Dieser Namespace dient den XAML-Spracherweiterungen. Greifen Sie auf ein Element dieses Namespace zu, müssen Sie vor dem Element das Namespace-Präfix, gefolgt von einem Doppelpunkt, angeben, z. B. `x:Class`.

CLR-Namespaces

In vielen WPF-Anwendungen müssen Sie über die Standardangabe hinaus noch weitere Namespaces angeben: Manchmal ist es ein Namespace der *Common Language Runtime* (CLR), manchmal auch ein Namespace, der in der aktuellen Anwendung beschrieben ist.

Angenommen, wir möchten in einer WPF-Anwendung mit XAML auf die Klasse `Circle` der Assembly *GeometricObjects.dll* zugreifen. Die Klasse `Circle` sei im Namespace `GeometricObjects` definiert.

```
namespace GeometricObjects {
  public class Circle {
    private int radius;
    public int XCoordinate  {get; set;}
    public int YCoordinate  {get; set;}
    public int Radius {
      get { return radius; }
      set { radius = value; }
    }
  }
}
```

Nachdem Sie in der WPF-Anwendung einen Verweis auf die Bibliothek angelegt haben, geben Sie den Namespace wie folgt im Wurzelelement `Window` an:

```
<Window x:Class="WpfApplication1.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:geo="clr-namespace:GeometricObjects;assembly=GeometricObjects"
  Title="Window1" Height="300" Width="300">
</Window>
```

Das Präfix, hier `geo`, ist frei wählbar, muss aber eindeutig sein. Danach folgen zwei Name-Wert-Paare, die durch ein Semikolon voneinander getrennt sind. Das erste Paar wird mit `clr-namespace` eingeleitet. Dahinter folgt ein Doppelpunkt, und anschließend kommt der CLR-Namespace, dem die Klasse zugeordnet ist. Das zweite Name-Wert-Paar wird mit `assembly` eingeleitet. Nach dem `=`-Zeichen wird die Assembly angegeben, jedoch ohne die Dateiendung DLL.

Jetzt können Sie die Klasse `Circle` im XAML-Code verwenden und den Eigenschaften die gewünschten Werte mitteilen. Der Elementangabe müssen Sie dabei das gewählte Präfix, hier `geo`, voranstellen.

```
<geo:Circle Radius="77" XCoordinate="100" YCoordinate="-250">
</geo:Circle>
```

Sie sollten bei der Entwicklung von Klassen, die unter XAML Verwendung finden sollen, beachten, dass nur die Eigenschaften, die durch eine Eigenschaftsmethode (*Property*) beschrieben werden, auch als Attribute erscheinen. Öffentliche Felder sind nicht deklarierbar.

Etwas einfacher wird die Bekanntgabe eines Namespace, sollte dieser zum aktuellen Projekt gehören. Sie können in diesem Fall auf die Angabe der Assembly verzichten, z. B. so:

```
xmlns:geo="clr-namespace:WpfApplication1"
```

Mehrere CLR-Namespaces zusammenfassen

Entwickeln Sie Klassenbibliotheken, sollten Sie daran denken, die Klassen für den Einsatz in XAML passend vorzubereiten. Dazu gehört neben der Bereitstellung von Propertys mit den Accessoren `get` und `set` auch die Berücksichtigung der Namespaces. Das gilt insbesondere für den Fall, dass in einer Klassenbibliothek mehrere CLR-Namespaces festgelegt sind. Im

folgenden Codefragment sind die beiden Klassen `Rectangle` und `Circle` jeweils unterschiedlichen Namespaces zugeordnet.

```
namespace Namespace1 {
  public class Circle { }
}
namespace Namespace2 {
  public class Rectangle { }
}
```

Ohne weitere Maßnahmen zu ergreifen, können Sie beide Typen nur dann im XAML-Code nutzen, wenn Sie beide Namespaces im Wurzelelement angeben, zum Beispiel:

```
xmlns:geo1="clr-namespace:Namespace1;assembly=GeometricObjects"
xmlns:geo2="clr-namespace:Namespace2;assembly=GeometricObjects"
```

Beide Namespaces lassen sich aber auch auf einen gemeinsamen XML-Namespace mappen. Dazu muss die entsprechende Vorkehrung bereits in der Klassenbibliothek erfolgen. Die notwendigen Angaben machen Sie in der Datei *AssemblyInfo.cs* der Klassenbibliothek. Ergänzen Sie diese dazu wie hier gezeigt um zwei Attributangaben:

```
[assembly: XmlnsDefinition("http://www.tollsoft.de", "Namespace1")]
[assembly: XmlnsDefinition("http://www.tollsoft.de", "Namespace2")]
```

Um auf das Attribut `XmlnsDefinition` zugreifen zu können, müssen Sie zuerst einen Verweis auf die Bibliothek *WindowsBase.dll* legen und sollten darüber hinaus den Namespace `System.Windows.Markup` bekanntgeben. Das Attribut beschreibt zwei Parameter. Dem ersten übergeben Sie den gewünschten XML-Namespace, dem zweiten Parameter teilen Sie mit, welcher CLR-Namespace auf diesen XML-Namespace gemappt werden soll. Der XAML-Code reduziert sich daraufhin auf eine Namespace-Bekanntgabe im Wurzelelement, beispielsweise:

```
xmlns:geo="http://www.tollsoft.de"
```

Sie können anschließend mit dem Präfix `geo` Elemente sowohl vom Typ `Circle` als auch vom Typ `Rectangle` in Ihren XAML-Code einbetten.

```
<geo:Circle><geo:Circle>
```

▲

17.4.7 XAML-Spracherweiterungen ▼▲

XAML definiert eine Reihe von Attributen, die besondere Aspekte bei der Entwicklung berücksichtigen und keine Entsprechungen in einer Klasse besitzen. Hiermit werden nur Zusatzinformationen geliefert, die einer besonderen Verabeitung bedürfen. Tabelle 17.1 stellt Ihnen einige davon vor. In den folgenden Kapiteln werden Sie in den Beispielen auf einige dieser Schlüsselwörter stoßen.

Schlüsselwort	Bedeutung
`x:Class`	Dieses Attribut stellt die Beziehung zwischen dem Wurzelelement im XAML-Code und der Code-Behind-Datei her.
`x:Code`	Die Trennung von Code und Oberflächenbeschreibung ist keine strikte Vorgabe. Sie können auch Code innerhalb einer

	XAML-Datei implementieren. Mit `x:Code` wird ein Codebereich im XAML-Code definiert.
`x:Key`	Gibt den eindeutigen Namen eines Elements in einer Ressource an.
`x:Name`	Mit diesem Attribut können Sie einem Element einen Namen geben, wenn das Element selbst nicht über eine Eigenschaft `Name` verfügt.

Tabelle 17.1 Schlüsselwörter von XAML (Auszug)

17.4.8 Markup-Erweiterungen

Es gibt mehrere Markup-Erweiterungen, die nicht spezifisch für die WPF-Anwendung sind, sondern Implementierungen für Funktionen von XAML als Sprache sind. Auch diese Markup-Erweiterungen sind durch das `x:`-Präfix in der Verwendung identifizierbar.

Erweiterung	Beschreibung

`x:Array`	Hiermit lassen sich Arrays in XAML definieren. Beispiel:
	`<x:Array Type="clr:Int32">`
	`  <clr:Int32>36</clr:Int32>`
	`  <clr:Int32>1270</clr:Int32>`
	`  <clr:Int32>5</clr:Int32>`
	`</x:Array>`

`x:Null`	Wird verwendet, um einem Element null zuzuweisen. Beispiel: `<Button Background="{x:Null}" />`
`x:Static`	Referenziert eine statische Variable oder Eigenschaft eines Objekts oder eine Konstante oder einen Enumerationswert. Beispiel: `<Button Background="{x:Static Brushes.Red}" />`
`x:Type`	Wird beispielsweise in Stildefinitionen benutzt, um einen Typ anzugeben. Beispiel: `<Style TargetType="{x:Type TextBox}" />`

Tabelle 17.2 Markup-Erweiterungen von XAML

<< zurück <top> vor >>