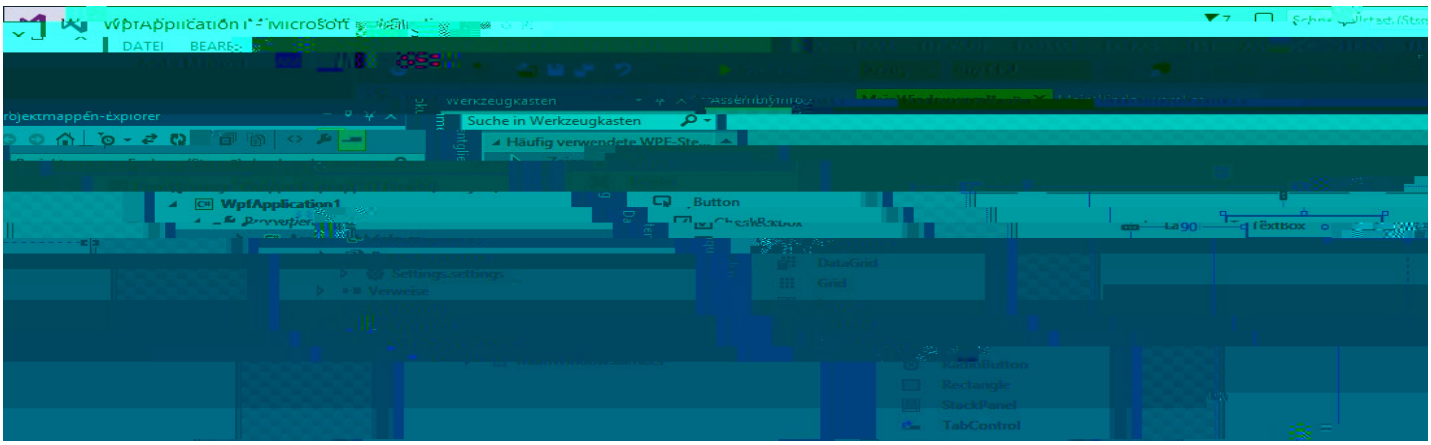
AUSGABE 02/2015

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.



TOP-THEMEN:

André Minhorst Verlag

Impressum

DATENBANKENTWICKLER
© André Minhorst Verlag
Borkhofer Str. 17
47137 Duisburg

Redaktion: Dipl.-Ing. André Minhorst

Mit Zeichenketten arbeiten

Schon VBA bietet eine Menge Funktionen für den Umgang mit Zeichenketten. Unter C# – wer hätte das gedacht – setzt Microsoft noch Einiges drauf. Einer der wichtigsten Unterschiede ist, dass eine Variable des Typs String nun nicht mehr einfach nur eine Zeichenkette aufnehmen kann, sondern sogar auf ein Objekt verweist, das den Text enthält und Methoden und Eigenschaften aufweist. Dieser Artikel liefert eine Einführung in den Umgang mit Zeichenketten unter C#.

Unter VBA ist eine Zeichenkette, also der Inhalt einer Variablen mit dem Datentyp **String**, ein Werttyp – genau wie die Zahlendatentypen oder Datum/Zeit. Unter C# (und den anderen .NET-Programmiersprachen) ist dies anders: Dort ist ein **String** ein Referenztyp, das heißt, dass die Objektvariable des Datentyps **String** lediglich die Speicheradresse enthält, hinter der sich der eigentliche Inhalt verbirgt. Außerdem bringt der **String** unter .NET die Eigenschaften eines Objekts mit, nämlich Methoden und Eigenschaften. Prinzipiell entspricht ein **String**-Objekt sogar einem Array, das Elemente des Datentyps **Char** enthält – der wiederum ein Werttyp ist.

Beispiele nachbauen

Wenn Sie die Beispiele dieses Artikels nachvollziehen möchten, legen Sie ein Projekt des Typs **Konsolenanwendung** mit der Programmiersprache **Visual C#** an. Die nachfolgenden Beispiele können Sie dann als Inhalt der Prozedur **Main** eintragen und mit **F5** ausführen.

Variable deklarieren und füllen

Für ein einfaches Beispiel, das einfach nur eine Variable des Typs **String** mit dem Namen **zeichenfolge** erstellt, mit einer Zeichenkette füllt und auf der Konsole ausgibt, sieht dies wie folgt aus:

```
namespace Zeichenketten {
    class Program     {
        static void Main(string[] args) {
            String zeichenfolge = "Beispieltext";
            Console.WriteLine(zeichenfolge);
            Console.ReadLine();
        }
    }
}
```

Die Anweisung zum Deklarieren und Füllen der Variablen können Sie, wie üblich, auch auf zwei Anweisungen aufteilen:

```
String zeichenfolge;
zeichenfolge = "Beispieltext";
```

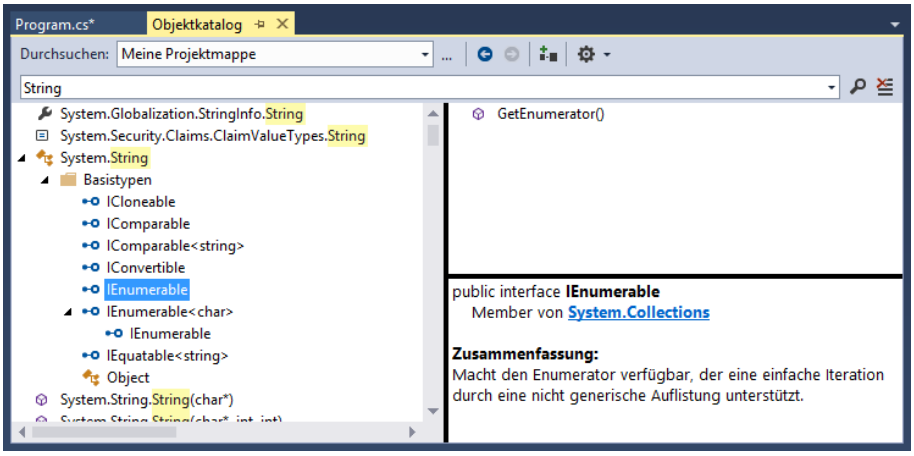


Bild 1: Ausgabe zweier **Write**-Anweisungen

Ein String ist ein Array?

Eingangs haben wir die Behauptung aufgestellt, dass ein **String**-Objekt Eigenschaften eines Array aufweisen würde. Dabei handelt es sich beispielsweise um die Möglichkeit, per Index auf die einzelnen Elemente des Typs **Char** zuzugreifen oder diese sogar per **foreach**-Schleife zu durchlaufen. Um diese Erkenntnis zu

gewinnen, gibt es zwei Möglichkeiten: Die erste ist es, sich einfach per IntelliSense die Eigenschaften und Methoden des **String**-Objekts anzeigen zu lassen. Oder Sie öffnen mit **Strg + Alt + J** den Objektkatalog und suchen dort nach der Zeichenkette **String**, was unter **System.String** schnell den richtigen Treffer liefert (siehe Bild 1).

Buchstaben per foreach-Schleife durchlaufen

Wenn Sie sich die Basistypen des **String**-Objekts ansehen, stoßen Sie schnell auf die **IEnumerable**-Schnittstelle. Ist diese vorhanden, können Sie die Elemente eines Objekts per **foreach**-Schleife durchlaufen.

Dies erledigen wir im folgenden Beispiel, wo wir eine **for-each**-Schleife über alle **Char**-Elemente des **String**-Objekts **zeichenfolge** durchlaufen und das jeweilige **Char**-Element mit der Variablen **buchstabe** referenzieren. Innerhalb der Schleife geben wir den Inhalt der **Char**-Variablen **buchstabe** zweimal aus – einmal unbehandelt und einmal zu einem Integer-Wert konvertiert:

```
String zeichenfolge = "Beispieltext";
foreach(Char buchstabe in zeichenfolge) {
    Console.WriteLine("{0} {1}", buchstabe,
        (int)buchstabe);
}
Console.ReadLine();
```

Das Ergebnis sieht wie in Bild 2 aus – neben dem Buchstaben erscheint der ASCII-Wert entsprechend der UTF-16-Kodierung für das jeweilige Zeichen.

Statt **int** hätten wir hier auch den Datentyp **byte** verwenden können:

```
(byte)buchstabe
```

Zeichenkette mit for durchlaufen

Das folgende Beispiel zeigt, wie Sie die Anzahl der enthaltenen Zeichen in einer **String**-Variablen ermitteln. Mit diesem Wert können wir dann eine **for**-Schleife definieren, die mit

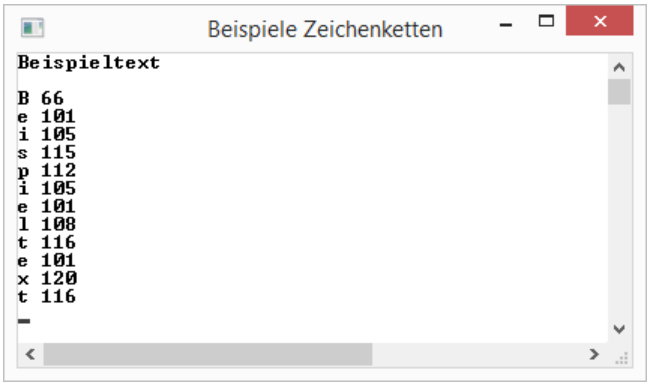


Bild 2: Ausgabe der **Char**-Elemente eines **String**-Objekts als Text und Integer

jedem Durchlauf das aktuelle **Char**-Element über den Index der **String**-Variablen **zeichenfolge** ermittelt und der Variablen **buchstabe** zuweist.

Auch hier landen der Buchstabe und der ASCII-Code in der Ausgabe:

```
String zeichenfolge = "Beispieltext";
int anzahl = zeichenfolge.Length;
for (int i = 0; i < anzahl; i++) {
    Char buchstabe = zeichenfolge[i];
    Console.WriteLine("{0} {1}", buchstabe,
        (int)buchstabe);
}
Console.ReadLine();
```

Dies ist schon einmal erheblich einfacher als unter VBA – dort hätten Sie erstens nur eine **For...Next**-Schleife verwenden können, um auf die einzelnen Zeichen zuzugreifen, und zweitens hätten wir für jeden einzelnen Zugriff die **Mid**-Funktion auf die Zeichenkette anwenden müssen:

```
For i = 0 To Len(strZeichenkette)
    Debug.Print Mid(strZeichenkette, i, 1)
Next i
```

Zur Ermittlung des ASCII-Codes hätten wir dort noch die Funktion **Asc** verwenden müssen, statt einfach nur den Buchstaben in den Zahlentyp **int** oder **byte** zu konvertieren.

Übrigens können Sie auch einfach über den Index auf ein Zeichen eines String-Objekts zugreifen – beispielsweise wie folgt auf das erste Zeichen:

```
string Vorname = "André";
char Buchstabe = Vorname[0];
```

Unterschied zwischen String und Char

Es gibt eine Reihe Unterschiede zwischen **string**- und **char**-Objekten. Einen haben Sie schon kennen gelernt: Ein Char ist ein Element eines Arrays, das in einer **string**-Variablen gespeichert wird. Ein anderer ist, dass nicht beide etwa beim Zuweisen in Anführungszeichen eingefasst werden.

Beim **string** verwenden Sie das übliche Anführungszeichen:

```
string zeichenfolge;
zeichenfolge = "Beispieltext";
```

Bei einer **Char**-Variable, die ja nur ein Zeichen aufnehmen kann, verwenden Sie hingegen einfache Anführungszeichen:

```
Char zeichen;
zeichen = 'A';
```

Dies nur, damit Sie sich nicht über Fehlermeldungen wundern, wenn Sie einmal einer **Char**-Variablen ein Zeichen in Anführungszeichen übergeben möchten (**Eine implizite Konvertierung von Typ 'string' in 'char' ist nicht möglich.**).

Zeichenfolge aus Char-Elementen zusammensetzen

Eine Zeichenfolge müssen Sie nicht immer direkt zuweisen; Sie können diese auch aus einem Array von **char**-Objekten zusammenstellen. Dazu brauchen Sie einfach nur das **char**-Array als Parameter beim Erstellen einer neuen **String**-Instanz anzugeben:

```
char[] chars = { 'A', 'n', 'd', 'r', 'é' };
String strChars = new String(chars);
Console.WriteLine("String aus Char-Array: {0}", strChars);
//Ausgabe:
//String aus Char-Array: André
```

Position einer Teilzeichenkette ermitteln

Unter VBA war hierzu die Funktion **InStr** angezeigt, wenn Sie das erste Auftreten einer Teilzeichenfolge ermitteln wollten, und **InStrRev**, wenn das erste Auftreten der Teilzeichenfolge von hinten gefragt war.

Das erste Auftreten einer Zeichenfolge in einer **String**-Variablen ermitteln Sie mit der Methode **IndexOf**, die als Parameter die zu suchende Teilzeichenkette erwartet:

```
String zeichenfolge;
zeichenfolge = "Beispieltext";
int position = zeichenfolge.IndexOf("spiel");
Console.WriteLine("Position von 'spiel': {0}", position);
//Ausgabe:
//3
```

Die Zeichenfolge **spiel** beginnt im Beispieltext mit dem vierten Zeichen, was dem Indexwert **3** (nullbasiert) entspricht.

Die VBA-Funktion **InStrRev** bilden Sie unter C# mit der Methode **LastIndexOf** ab. Diese sucht das erste Auftreten einer Zeichenfolge von hinten:

```
zeichenfolge = "blablabla";
position = zeichenfolge.LastIndexOf("bla");
Console.WriteLine("Position: {0}", position);
```

Zeichenfolge nicht gefunden

Findet **IndexOf** keine passende Teilzeichenkette, liefert es den Wert **-1** zurück:

```
zeichenfolge = "blablabla";
position = zeichenfolge.IndexOf("bla");
while (position != -1) {
    Console.WriteLine("Position: {0}", position);
    position = zeichenfolge.IndexOf("bla", position + 1);
}
```

```
//Ausgabe:
//Position: -1
```

Alle Auftreten einer Zeichenfolge finden

Unter VBA haben Sie hier eine **Do While**-Schleife genutzt, die so lange lief, bis der Wert für die gefundene Position den Wert **0** angenommen hat.

Hierzu ist anzumerken, dass **InStr** 1-basiert ist:

```
Dim intPosition As Integer
intPosition = InStr(1, "blablabla", "bla")
Do While Not intPosition = 0
    Debug.Print intPosition
    intPosition = InStr(intPosition + 1, "blablabla", "bla")
Loop
```

Unter C# hilft eine **while**-Schleife weiter sowie eine Überladung der Methode **IndexOf** – nämlich diejenige, die als zweiten Parameter die Startposition für die Suche erwartet:

```
zeichenfolge = "blablabla";
position = zeichenfolge.IndexOf("bla");
while (position != -1) {
    Console.WriteLine("Position: {0}", position);
    position = zeichenfolge.IndexOf("bla", position + 1);
}
//Ausgabe:
//0
//3
//6
```

Hier ist zu beachten, dass **IndexOf** – wie oben beschrieben – den Wert **-1** zurückliefert, wenn kein weiteres Auftreten mehr gefunden wird. Die Schleife verwendet dementsprechend den Ausdruck **position != -1** als Kriterium für die Fortsetzung.

Zeichenfolgen vergleichen

Zum Vergleichen zweier Zeichenfolgen liefert C# gleich mehrere Möglichkeiten, die wir uns nun ansehen.

Die erste Variante ist die Methode **Equals** des **string**-Objekts. Diese verwenden Sie als Methode des ersten zu vergleichenden **string**-Objekts und geben das zweite **string**-Objekt als Parameter an:

```
string str1 = "André";
string str2 = "André";
Console.WriteLine("{0} Equals {1}: {2}", str1.ToString(),
    str2.ToString(), str1.Equals(str2));
//Ausgabe:
//André Equals André: True
```

Diese Methode vergleicht tatsächlich die in den beiden Variablen gespeicherten Objekte. Aber wie können diese gleich sein – wir haben doch unabhängig voneinander zwei **string**-Objekte angelegt und gefüllt? Der Grund ist: Die Strings werden intern gespeichert, aber nur jeweils einmal. Haben Sie also bereits eine **string**-Variable mit dem Wert **André** gefüllt und legen eine weitere Variable mit diesem Wert an, dann verweisen beide **string**-Variablen auf das gleiche **string**-Objekt.

Anders funktioniert die Vergleichsmethode **Compare**. Diese ist eine Methode der **String**-Klasse und erwartet die beiden zu vergleichenden **string**-Variablen als Parameter. Sie wird beispielsweise so aufgerufen:

```
Console.WriteLine(String.Compare(str1, str2);
```

Sind beide Zeichenketten gleich, liefert **Compare** nicht den Wert **True**, sondern den Wert **0**. Andere Ergebnisse sind **-1**, wenn der erste Ausdruck kleiner als der zweite ist (zum Beispiel **a** und **b**) und **1**, wenn der erste Ausdruck kleiner als der zweite ist (**b** und **a**). Dabei bedeutet kleiner, dass die verglichenen Buchstaben sich im Alphabet weiter vorn befinden. Bezüglich kleiner und großer Buchstaben gilt, dass kleinere Buchstaben kleiner sind. **a** ist also kleiner als **A** und **A** ist kleiner als **b**.

Die Methode **CompareOrdinal** funktioniert ähnlich wie **Compare** und liefert bei Gleichheit den Wert **0** zurück. Wenn

jedoch beim zeichenweisen Durchlaufen der erste Unterschied festgestellt wird, ermittelt die Methode die Differenz bezüglich des ASCII-Codes der Zeichen.

Im folgenden Beispiel stößt die Methode also beim dritten Zeichen auf einen Unterschied. Sie ermittelt die ASCII-Codes der beiden Zeichen (**67** für **C** und **99** für **c**) und gibt als Ergebnis die Differenz aus, also **-32**:

```
str1 = "ABC";
str2 = "ABc";
Console.WriteLine("{0} CompareOrdinal {1}: {2}",
    str1.ToString(), str2.ToString(),
    String.CompareOrdinal(str1, str2));
//Ausgabe:
//ABC CompareOrdinal ABc: -32
```

Die Überladung dieser Methode konzentriert sich auf Teilzeichenketten der zu vergleichenden **string**-Objekte:

```
String.CompareOrdinal(str1, 3, str2, 3, 1));
```

Der erste und dritte Parameter nehmen wieder die zu untersuchenden Zeichenfolgen entgegen. Der zweite und vierte Parameter legen fest, ab welcher Position diese untersucht werden sollen, der fünfte, wie lang die zu vergleichenden Zeichenketten sein sollen.

Mid, Left, Right mit Substring

Diese drei VBA-Funktionen erlauben es, eine beliebige Teilzeichenfolge, eine Teilzeichenfolge beginnend mit dem ersten oder eine Teilzeichenfolge endend mit dem letzten Zeichen einer Zeichenkette zu ermitteln.

Unter C# ist das etwas komplizierter:

```
String zeichenfolge = "1234567890";
//erste vier Zeichen
Console.WriteLine(zeichenfolge.Substring(0, 4));
//letzte vier Zeichen
Console.WriteLine(zeichenfolge.Substring(6, 4));
//mittlere vier Zeichen
Console.WriteLine(zeichenfolge.Substring(3, 4));
```

Die Funktionen **Left**, **Right** und **Mid** gibt es also leider nicht. Sie könnten diese zwar nachbauen, aber wir sollten uns doch auf die vorhandenen Möglichkeiten konzentrieren – zum Beispiel die folgende.

Womit beginnt oder endet die Zeichenfolge?

Und wer **Left** oder **Right** nur dazu verwenden will, einen Teil der Zeichenkette von vorn oder hinten zu extrahieren, um diesen mit einer anderen Zeichenkette zu vergleichen, der wird mit den folgenden beiden Methoden ohnehin viel leichter glücklich:

- **StartsWith** ist eine Methode eines **string**-Objekts, das die ersten Zeichen mit der als Parameter übergebenen Zeichenkette vergleicht.

- **EndsWith** erledigt dies für das Ende der Zeichenkette. Beide berücksichtigen die Groß- und Kleinschreibung.

Im ersten Beispiel prüfen wir, ob die Zeichenkette aus **str1** mit der Zeichenkette **str2** beginnt:

```
string str1 = "André Minhorst";
string str2 = "And";
Console.WriteLine("'{0}' beginnt mit '{1}': {2}",
    str1, str2, str1.StartsWith(str2));
//Ausgabe:
//'André Minhorst' beginnt mit 'And': True
```

Das zweite Beispiel vergleicht das Zeichenkettenende mit dem Inhalt von **str3**:

```
string str1 = "André Minhorst";
string str3 = "rst";
Console.WriteLine("'{0}' endet mit '{1}': {2}",
    str1, str3, str1.EndsWith(str3));
//Ausgabe:
//'André Minhorst' endet mit 'rst': True
```

Schließlich noch ein Beispiel dafür, dass der Start nicht mit der Vergleichszeichenfolge übereinstimmt:

```
Console.WriteLine("'{0}' endet mit '{1}': {2}",
    str1, str2, str1.EndsWith(str2));
//Ausgabe:
//'André Minhorst' beginnt mit 'rst': False
```

Der Vorteil gegenüber der von VBA bekannten Methode, erst die zu vergleichende Teilzeichenkette des ersten Ausdrucks mit **Left** oder **Right** zu ermitteln und dann mit der Vergleichszeichenkette zu vergleichen, ist der, dass man sich nicht um die Länge der Vergleichszeichenkette kümmern muss. **StartsWith** und **EndsWith** vergleichen automatisch die entsprechende Teilzeichenkette.

Zeichen hinten oder mittendrin entfernen

Die Remove-Methode erlaubt es, den hinteren Teil einer Zeichenkette ab einem bestimmten Zeichen abzuschneiden – zum Beispiel ab dem Zeichen mit dem Index **5**:

```
str1 = "0123456789";
Console.WriteLine("Ab dem sechsten Zeichen abschneiden:
{0}", str1.Remove(5));
//Ausgabe:
//Ab dem sechsten Zeichen abschneiden: 01234
```

Alternativ trennen Sie eine Zeichenkette heraus, zum Beispiel die vom sechsten bis zum achten Zeichen:

```
Console.WriteLine("Vom sechsten bis achten Zeichen
    abschneiden: {0}", str1.Remove(5,3));
//Ausgabe:
//Vom sechsten bis zum achten Zeichen abschneiden: 0123489
```

Zeichenkette in Array-Elemente zerlegen

Unter VBA kann man eine Zeichenkette an definierten Stellen, also einem speziellen Zeichen, mit der **Split**-Funktion in ein Array mit je einem Element pro enthaltenem Wort zerlegen. Unter C# erledigen Sie das mit der gleichnamigen Methode, die in der einfachsten Version keine Parameter erwartet:

```
zeichenfolge = "wort1 wort2 wort3";
string[] woerter = zeichenfolge.Split();
foreach (string wort in woerter)
{
    Console.WriteLine(wort);
}
//Ausgabe
//Wort1
//Wort2
//Wort3
```

Die **Split**-Methode bietet einige Überladungen an. Eine davon erlaubt die Angabe eines Array von Trennzeichen. Wenn Sie etwa eine Zeichenfolge wie die folgende haben, können Sie alle Elemente ermitteln, die durch das Leerzeichen oder das Pipe-Zeichen voneinander getrennt werden:

```
zeichenfolge = "wort1 wort2|wort3 wort4|wort5";
```

Dazu definieren Sie folgendes **Char**-Array mit den Trennzeichen und übergeben es als Parameter der **Split**-Methode:

```
Char[] trennzeichen = { ' ', '|' };
woerter = zeichenfolge.Split(trennzeichen);
foreach (string wort in woerter)
{
    Console.WriteLine(wort);
}
//Ausgabe
//Wort1
//Wort2
//Wort3
//Wort4
//Wort5
```

Eine weitere Überladung erwartet als zweiten Parameter die Anzahl der zurückzuliefernden Elemente.

Wenn die Liste also etwa wie oben fünf Elemente enthält, Sie aber nur die ersten drei sehen wollen, ersetzen Sie den Aufruf der **Split**-Methode wie folgt:

Auflistungen mit der ArrayList

Von VBA kennen Sie als Auflistungsklassen zunächst die eingebaute Collection. Wer noch etwas weiter geht, bindet die Scripting Runtime Library ein und nutzt die Dictionary-Klasse zum Speichern von Auflistungen. Über den reinen Programmcode hinweg ließen sich diese Elemente unter VBA kaum nutzen – als Datenherkunft etwa für Formular oder Kombinationsfelder taugten sie nicht. Unter C# sieht dies anders aus. Es gibt eine ganze Reihe verschiedener Auflistungsklassen, die sich teilweise auch als Datenherkunft für Steuerelemente eignen. In diesem Artikel schauen wir uns die ArrayList als Vertreter der Auflistungsklassen an.

Die Auflistungsklasse **ArrayList** steckt im Namespace **System.Collections**. Diese fügen Sie mit der **using**-Anweisung zur Klasse hinzu – also so:

```
using System.Collections;
```

Auf diese Weise können Sie gleich auf das ArrayList-Objekt und seine Methoden und Eigenschaften zugreifen und müssen nicht immer den Namespace voranstellen. Ohne Angabe des Namespace mit der using-Anweisung wäre für die Deklaration beispielsweise diese Zeile nötig:

```
System.Collections.ArrayList arrayList;
```

Wenn Sie den Namespace zuvor wie oben beschrieben angegeben haben, reicht dies aus:

```
ArrayList arrayList;
```

Schnittstellen von Auflistungsklassen

Neben den eigentlichen **Collection**-Klassen enthalten die beiden Namespaces noch einige weitere Objekte. Einige davon sind Interfaces, also Schnittstellenklassen. Diese beschreiben die grundlegenden Methoden und Eigenschaften der **Collection**-Objekte. Die **Collection**-Objekte implementieren diese Schnittstellen, was kurz gefasst bedeutet, dass sie die für die Schnittstelle vorgegebenen Methoden, Eigenschaften und Ereignisse enthalten.

Diese Schnittstellenklassen sehen Sie sich am besten im Objektkatalog an (**Strg + Alt + J**). Wenn Sie dort unter **mscorlib** den Eintrag **System.Collections** auswählen, erhalten Sie die Übersicht aus Bild 1. Dort sehen Sie, dass zum Beispiel die Methode **Add** zum Hinzufügen eines Auflistungsobjekts aus der Schnittstelle **IList** stammt – genauso wie einige

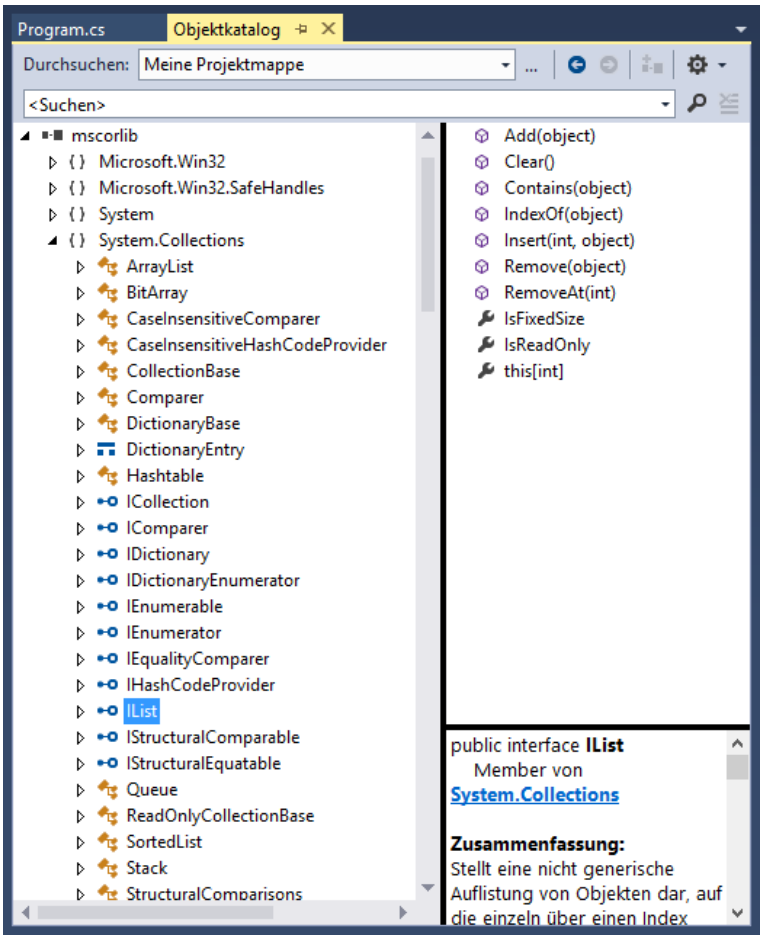


Bild 1: Schnittstelle einer Auflistungsklasse im Objektkatalog

andere Methoden. Die folgenden Schnittstellenklassen liefern die meisten Methoden und Eigenschaften, aus denen sich die verschiedenen Auflistungsklassen zusammensetzen:

- **ICollection**: Liefert neben anderen die **Count**-Eigenschaft zum Ermitteln der Anzahl der Elemente und die **CopyTo**-Methode, mit der Sie den Inhalt der Auflistungsklasse in ein Array kopieren können.

- **IList**: Liefert Methoden, um Listenelemente hinzuzufügen, zu entfernen oder deren Informationen zu ermitteln. Auflistungsklassen, die diese Schnittstelle implementieren, stellen ihre Elemente über einen Index bereit. Dies entspricht also etwa dem **Collection**-Objekt, das Sie von VBA kennen. Die Elemente der **IList**-Schnittstelle haben Sie bereits im Artikel **Von VBA Zu C#: Arrays** (**www.datenbankentwickler.de/6**) kennengelernt.

- **IDictionary**: Liefert ähnliche Methoden wie **IList**, aber ist für die Verwaltung von Name-Wert-Paaren ausgelegt. Dies entspricht dem **Dictionary**-Objekt der **Scripting Runtime Library**, das Sie möglicherweise unter VBA genutzt haben.

- **IEnumerable**: Steuert die Möglichkeit bei, die Elemente einer Auflistung per **foreach**-Schleife zu durchlaufen.

Diese Schnittstellenklassen liefern allerdings nicht alle Methoden und Eigenschaften etwa der **ArrayList**-Klasse – diese trägt selbst auch noch eigene Elemente bei.

ArrayList deklarieren und instanzieren

Ein **ArrayList**-Objekt können Sie in zwei Schritten deklarieren und instanzieren:

```
ArrayList arrayList;
arrayList = new ArrayList();
```

Wenn diese beiden Anweisungen ohnehin direkt nacheinander ausgeführt werden, können Sie dies auch in einer Zeile durchführen:

```
ArrayList arrayList = new ArrayList();
```

Elemente hinzufügen

Die einfachste Methode, ein Element zu einer **ArrayList** hinzuzufügen, ist die **Add**-Methode. Sie erwartet lediglich den hinzuzufügenden Wert beziehungsweise die Variable mit dem Objektverweis als Parameter. Die folgenden Anweisungen füllen das Array beispielsweise mit drei Zeichenketten:

```
arrayList.Add("Element 1");
arrayList.Add("Element 2");
arrayList.Add("Element 3");
```

Die **Add**-Methode fügt die Elemente immer hinten an die Liste an.

Index beim Hinzufügen ermitteln

Wenn Sie gleich beim Hinzufügen den Indexwert für das neue Element ermitteln wollen, speichern Sie das Ergebnis der **Add**-Methode in einer Integer-Variablen. Diesen können Sie dann wie folgt auf der Konsole ausgeben:

```
int indexNeuesElement = arrayList.Add("Element 4");
Console.WriteLine("Der Index des neuen Elements lautet:
{0}", indexNeuesElement);
```

Elemente direkt beim Initialisieren hinzufügen

Sie können auch direkt beim Initialisieren eine Liste der Elemente in geschweiften Klammern angeben:

```
arrayList = new ArrayList { "Element 1", "Element 2",
"Element 3" };
```

Anzahl der Elemente beim Erstellen angeben

Sie können beim Erstellen eines **ArrayList**-Objekts gleich die Anzahl der Einträge angeben – und zwar in Klammern hinter dem **ArrayList**-Schlüsselwort nach der **new**-Anweisung. Folgendes legt eine **ArrayList** mit drei Elementen an, aber nur die ersten beiden werden gefüllt. Bei der Ausgabe berücksichtigt die **foreach**-Schleife nur solche Elemente, die auch gefüllt wurden:

```
arrayList = new ArrayList(3);
arrayList.Add("Element 1");
arrayList.Add("Element 2");
foreach (string Element in arrayList) {
    Console.WriteLine("Element: {0}", Element);
}
```

Mit der Eigenschaft **Capacity** geben Sie die Anzahl der reservierten Elemente zurück oder legen diese fest – zum Beispiel so:

```
Console.WriteLine("Anzahl Elemente: {0}",
    arrayList.Capacity);
```

Element an bestimmter Stelle einfügen

Da die **Add**-Methode neue Elemente immer nur hinten an die Liste anhängt, benötigen wir noch eine Methode, die Elemente an einer bestimmten Position einfügt. Dies gelingt mit der Methode **Insert**. Diese erwartet als ersten Parameter den Index-Wert der Position, an der das neue Element eingefügt werden soll, als zweiten Parameter das einzufügende Element. Die folgenden Anweisungen legen zunächst zwei Elemente an und fügen dann ein weiteres Element an die Position des zweiten, sodass dieses an die dritte Stelle rutscht:

```
arrayList.Add("Element 1");
arrayList.Add("Element 2");
arrayList.Insert(1, "Element zwischen 1 und 2");
```

Da der Index der ArrayList 0-basiert ist, geben Sie als Position für den zweiten Eintrag den Wert **1** an.

Element entfernen

Wenn Sie gezielt ein Element aus einem Array entfernen möchten, benötigen Sie entweder dessen Inhalt oder dessen Index-Wert. Wenn Sie das Element kennen, können Sie es direkt über seinen Inhalt mit der **Remove**-Methode löschen:

```
arrayList.Add("Element 1");
arrayList.Add("Element 2");
arrayList.Add("Element 3");
arrayList.Remove("Element 2");
foreach (string eintrag in arrayList)
{
    Console.WriteLine(eintrag);
}
//Ausgabe:
//Element 1
//Element 2
```

Oder Sie kennen den Index des zu löschenden Elements. Diesen können Sie dann der **RemoveAt**-Methode als Parameter übergeben. Das folgende Beispiel entfernt beispielsweise das zweite von drei Elementen aus einer **ArrayList**:

```
arrayList.Add("Element 1");
arrayList.Add("Element 2");
arrayList.Add("Element 3");
arrayList.RemoveAt(1);
foreach (string eintrag in arrayList) {
    Console.WriteLine(eintrag);
}
//Ausgabe:
//Element 1
//Element 2
```

Mehrere Elemente entfernen

Wenn Sie mehrere zusammenhängende Elemente entfernen möchten, nutzen Sie die **RemoveRange**-Methode. Diese erwartet zwei Parameter: den Index des ersten und die Anzahl der zu entfernenden Elemente. Im folgenden Beispiel entfernt die Methode das zweite und dritte von vier Elementen:

```
arrayList = new ArrayList();
arrayList.Add("Element 1");
arrayList.Add("Element 2");
arrayList.Add("Element 3");
arrayList.Add("Element 4");
arrayList.RemoveRange(1, 2);
foreach (string eintrag in arrayList) {
    Console.WriteLine(eintrag);
}
```

```
//Ausgabe:
//Element 1
//Element 4
```

Elemente aus einer anderen ArrayList anfügen

Wenn Sie zwei **ArrayList**-Objekte verwenden und die Elemente des zweiten hinten an das erste Array anfügen möchten, verwenden Sie die Methode **AddRange**. Im folgenden Beispiel werden zwei **ArrayList**-Objekte mit je zwei Elementen gefüllt. Dann fügt die **AddRange**-Methode die Elemente von **arrayList2** hinten an **arrayList** an:

```
arrayList = new ArrayList();
arrayList.Add("Element 1");
arrayList.Add("Element 2");
arrayList2 = new ArrayList();
arrayList2.Add("Element 3");
arrayList2.Add("Element 4");
arrayList.AddRange(arrayList2);
```

Würden Sie nun alle Elemente von **arrayList** auf der Konsole ausgeben, erhielten Sie dieses Ergebnis:

```
Element 1
Element 2
Element 3
Element 4
```

Elemente aus einer anderen ArrayList einfügen

Auf ähnliche Weise fügen Sie die Elemente des zweiten **ArrayList**-Objekts an beliebiger Stelle in der ersten ArrayList ein. Dazu nutzen Sie diesmal jedoch die Methode **InsertRange**. Diese erwartet neben der Angabe der einzufügenden Liste noch den Index, an dessen Stelle die neuen Elemente eingefügt werden sollen:

```
arrayList = new ArrayList();
arrayList.Add("Element 1");
arrayList.Add("Element 2");
arrayList2 = new ArrayList();
arrayList2.Add("Element 3");
arrayList2.Add("Element 4");
arrayList.InsertRange(1,arrayList2);
foreach (String eintrag in arrayList) {
    Console.WriteLine(eintrag);
}
//Ausgabe:
//Element 1
//Element 3
//Element 4
//Element 2
```

Neue ArrayList aus Elementen einer ArrayList erstellen

Sie können auch aus einem oder mehreren Elementen einer ArrayList eine neue ArrayList erstellen. Dies geschieht mit der Methode **GetRange**. **GetRange** erwartet zwei Parameter: den Index des ersten zu exportierenden Elements und die Anzahl der Elemente. Das Ergebnis weisen Sie einfach einem anderen **ArrayList**-Objekt zu, wie hier **arrayList** zu **arrayList2**:

```
arrayList = new ArrayList();
arrayList.Add("Element 1");
arrayList.Add("Element 2");
arrayList.Add("Element 3");
arrayList2 = new ArrayList();
arrayList2 = arrayList.GetRange(1, 2);
foreach (string eintrag in arrayList2) {
    Console.WriteLine(eintrag);
}
//Ausgabe:
//Element 2
//Element 3
```

ArrayList mit Elementen einer beliebigen Collection füllen

Wenn Sie alle Elemente einer Collection an einer bestimmten Stelle eines **ArrayList**-Objekts einfügen und somit die dort vorliegenden Elemente überschreiben wollen, erledigen Sie dies mit der **SetRange**-Methode. Diese erwartet den Startindex für die zu ersetzenden Elemente der Ziel-**ArrayList**

sowie einen Verweis auf die Collection, welche die zu kopierenden Elemente liefert:

```
ArrayList zielArrayList = new ArrayList();
zielArrayList.Add("Element 1");
zielArrayList.Add("Element 2");
zielArrayList.Add("Element 3");
zielArrayList.Add("Element 4");
zielArrayList.Add("Element 5");
ArrayList quellArrayList = new ArrayList();
quellArrayList.Add("Element 6");
quellArrayList.Add("Element 7");
zielArrayList.SetRange(2, quellArrayList);
foreach (string Element in zielArrayList)
{
    Console.WriteLine("Element: {0}", Element);
}
//Ausgabe:
//Element 1
//Element 2
//Element 6
//Element 7
//Element 5
```

ArrayList in Array überführen

Sie können die Elemente der ArrayList auch in ein Array des Typs **Object** überführen. Dazu verwenden Sie die Methode **ToArray()** und schreiben das Ergebnis mit einer Variable des Typs **Object**:

```
arrayList = new ArrayList();
arrayList.Add("Element 1");
arrayList.Add("Element 2");
arrayList.Add("Element 3");
ArrayList arrayList6 = new ArrayList();
object[] array;
array = arrayList.ToArray();
foreach (string Element in array)
{
    Console.WriteLine("Element: {0}", Element);
}
```

Mit CopyTo in ein Array kopieren

Die **CopyTo**-Methode ist recht flexibel, da sie es erlaubt, sowohl die Anzahl als auch die Position der zu kopierenden Elemente aus dem **ArrayList**-Objekt sowie der zu ersetzenden Elemente aus dem Array festzulegen.

Variante 1 nur mit dem Zielarray als Parameter sieht so aus und ersetzt alle Elemente des Zielarrays durch die Elemente der **ArrayList**:

```
arrayList = new ArrayList();
arrayList.Add("Element 4");
arrayList.Add("Element 5");
arrayList.Add("Element 6");
String[] Zielarray = new String[3];
Zielarray[0] = "Element 1";
Zielarray[1] = "Element 2";
Zielarray[2] = "Element 3";
arrayList.CopyTo(Zielarray);
```

Die erste Überladung dieser Methode erwartet als zweiten Parameter den Index des Zielarrays, an dem mit dem Einfügen begonnen werden soll – zum Beispiel:

```
arrayList.CopyTo(Zielarray,1);
```

Dies kopiert den kompletten Inhalt von **arrayList** in **Zielarray**, und zwar dort ab dem mit dem zweiten Parameter angegebenen Index. Für die obige Konstellation würde dies einen Fehler auslösen, weil drei Elemente ab dem zweiten Element des Arrays nicht genügend Platz haben.

Die zweite Überladung erwartet als ersten Parameter den Startindex für das Zielarray, als dritten den Index des ersten zu kopierenden Elements und als vierten die Anzahl der von dieser Position aus zu kopierenden Elemente:

```
arrayList.CopyTo(1,Zielarray,1,1);
```

Mit der obigen Konstellation würde nur das Element **Element 5** an die Position von **Element 2** kopiert.

Steuerelemente anordnen

Unter Access ist das Anordnen von Steuerelementen einfach: Sie platzieren diese einfach auf dem Formular. In neueren Access-Versionen gibt es noch die Eigenschaften zum Verankern von Steuerelementen, aber damit sind die Möglichkeiten bereits ausgeschöpft. Unter WPF gibt es zahlreiche weitere Möglichkeiten, um die Position von Steuerelementen festzulegen. Dieser Artikel zeigt, wie es funktioniert und welche Vorteile sich gegenüber Access ergeben. Dabei lernen Sie auch gleich noch das Grid-Steuerelement und das GridSplitter-Steuerelement kennen, mit dem Sie Bereiche des Fensters vergrößern oder verkleinern können.

Das Ziel beim Programmieren mit WPF und einer Programmiersprache wie C# ist ja, die Benutzeroberfläche unabhängig vom dahinter steckenden Code entwickeln zu können – und umgekehrt. In der ersten Instanz werden wir noch nicht komplett dazu übergehen, sondern beispielsweise das Auslösen von Ereignissen noch in der Definition der Benutzeroberfläche verankern. Später lernen Sie Techniken kennen, auch dies von der Benutzeroberfläche zu entkoppeln. Theoretisch können Sie also tatsächlich einen Designer mit der Bearbeitung der Benutzeroberfläche betrauen, während Sie sich um den Code kümmern, der die Benutzeroberfläche steuert und von ihren Elementen ausgelöst wird.

Hierarchische Struktur

Wenn Sie ein Steuerelement zu einem neuen Fenster hinzufügen, sieht dies zunächst nicht viel anders aus wie eines in einem Access-Formular. Die XAML-Definition zeigt allerdings schnell, dass hier eine verschachtelte Struktur entstanden ist: Das **Window**-Element (das dem Formular unter Access entspricht) enthält zunächst ein **Grid**-Element, das dann wiederum etwa ein **Button**-Element aufnimmt (siehe Bild 1). Was es mit dem **Grid**-Element auf sich hat, schauen wir uns weiter unten an.

Element markieren

Wenn Sie nun das **Button**-Element markieren wollen, klicken Sie es einfach an. Beim **Grid**-Element und beim **Window**-Element wird das schon schwieriger. Nach etwas Experimentieren findet man heraus, dass man das **Window**-Element mit einem Klick auf den Fenstertitel markieren kann. Eine sichere Methode, das richtige Element zu markieren, bietet das Kontextmenü eines der Elemente. Dazu wählen Sie das Element der untersten gewünschten Hierarchie-Ebene aus, also beispielsweise das **Button**-Element aus der Abbildung. Das Kontextmenü bietet per Rechtsklick auf das Element den Eintrag **Aktuelle Aus-**

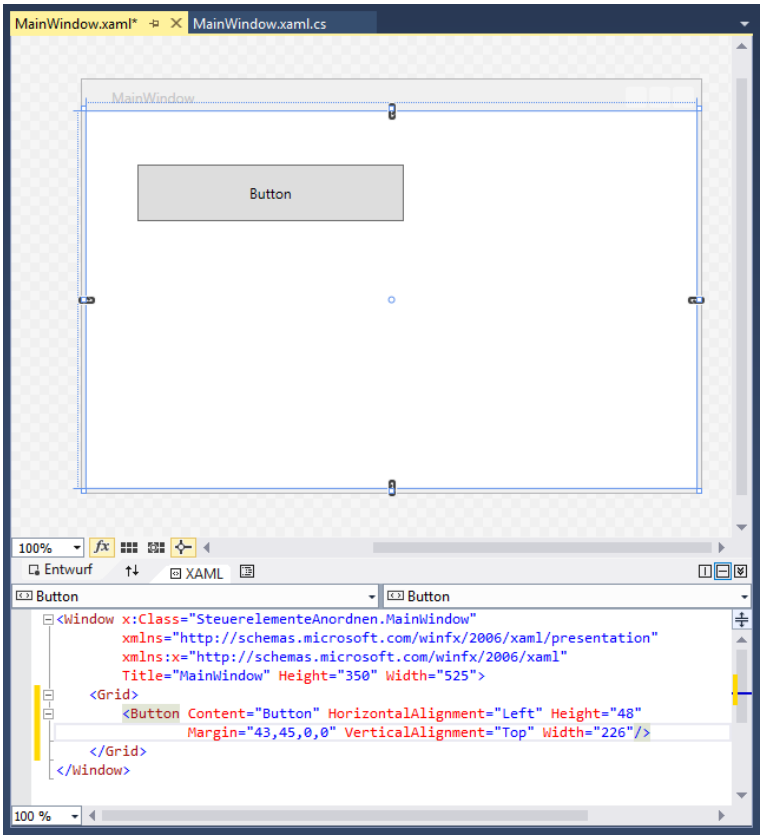


Bild 1: Ein Element mit seinen Eigenschaften

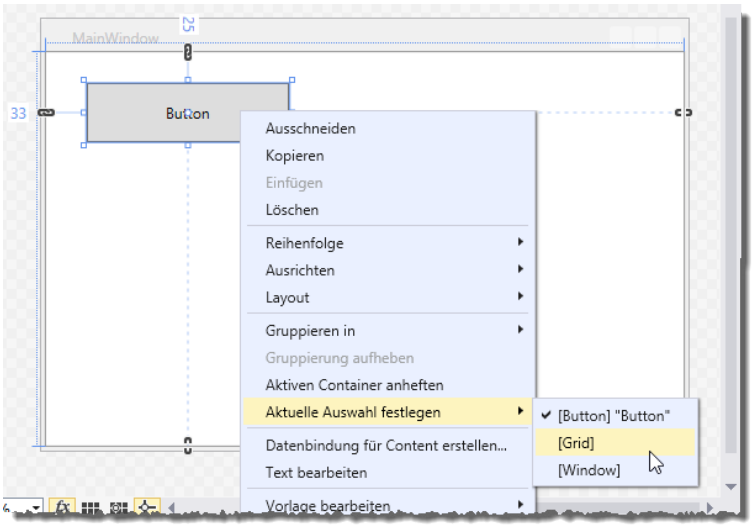


Bild 2: Markieren eines Elements der Hierarchie per Kontextmenü

wahl festlegen an, mit dem Sie eines der Elemente **[Button]**, **[Grid]** oder **[Window]** auswählen können (siehe Bild 2).

Größe und Position

Im Gegensatz zum Formularentwurf unter Access fällt gleich auf, dass die relevanten Maße des aktuell markierten Elements im Entwurf angezeigt werden. So werden die Abstände zum übergeordneten Element (im Falle des **Button**-Elements etwa zum **Grid**-Element) ständig angezeigt. Die Breite und Höhe eines Elements erscheinen, wenn Sie diese mit der Maus ändern (siehe Bild 3).

Width und Height

Die Größe finden Sie in der WPF-Definition unter den Eigenschaften **Height** beziehungsweise **Width** des jeweiligen Elements wieder. Damit legen Sie die Größe des Elements explizit fest. Mit **MinWidth, MaxWidth**, **MinHeight** und **MaxHeight** gibt es jedoch noch vier weitere Eigenschaften. Damit geben Sie die minimale und die maximale Höhe und Breite für ein Steuerelement an. Standardmäßig erscheint das Steuerelement dann mit den in **MinWidth** und **MinHeight** angegebenen Maßen. Es kann jedoch sein, dass sich der ursprüngliche Inhalt einmal ändert – beispielsweise, weil eine Schaltflächenbeschriftung in einer anderen Sprache angezeigt werden soll und mehr Platz einnimmt oder weil die Beschriftung/Schriftart/Schriftgröße je nach Kontext verändert werden soll. In einem solchen Fall würde man die **Min...**-Eigenschaften nutzen.

Sind beispielsweise **MinWidth**, **MaxWidth** und **Width** gesetzt, wird **Width** verwendet, sofern es innerhalb des Bereichs von **MinWidth** und **MaxWidth** liegt – das gilt auch für die entsprechenden **Height**-Werte.

Sind keine Breiten- oder Höhenangaben gemacht worden, richten sich die Abmessungen nach den maximal für den Inhalt nötigen Werten.

Wenn Sie die aktuelle Höhe und Breite eines Elements ermitteln wollen, verwenden Sie die Eigenschaften **ActualHeight** und **ActualWidth**.

Sie sollten keine absoluten Werte für Höhe und Breite angeben, wenn dies nicht dringend nötig ist – anderenfalls kann es sein, dass ein zur Laufzeit zugewiesener Text abgeschnitten dargestellt wird.

Margin

Die Position relativ zum übergeordneten Element ergibt sich aus der Eigenschaft **Margin**. Diese nimmt verschiedene Zeichenfolgen entgegen. Enthält diese etwa nur einen Zahlenwert, wird dieser für die Abstände zum linken, rechten,

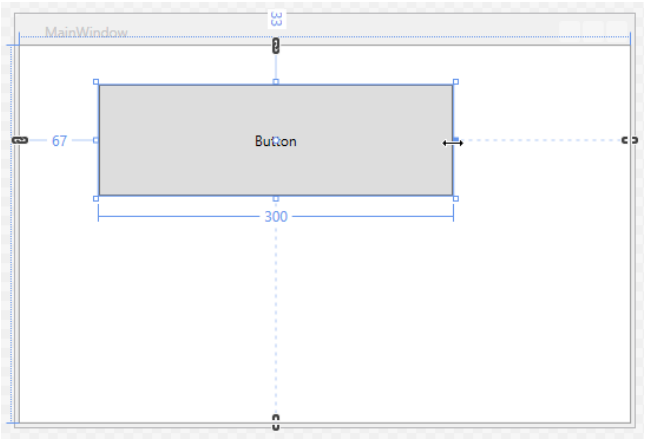


Bild 3: Anzeigen der Größe bei Anpassen von Höhe oder Breite

oberen und unteren Rand verwendet. Der folgende Button legt den Abstand von allen vier Seiten des übergeordneten Elements auf **10** fest:

```
<Button Margin="10" Width="96" Height="25" ... />
```

Wenn Sie diese vier Werte einzeln angeben möchten, können Sie einen Ausdruck für **Margin** angeben, der die vier Werte für **Left**, **Top**, **Width** und **Height** als kommaseparierte Liste entgegennimmt:

```
<Button Margin="10, 10, 10, 10" Width="96" Height="25" ... />
```

VerticalAlignment und HorizontalAlignment

Unter Access kannten Sie möglicherweise die mit den neueren Versionen eingeführten Eigenschaften **Horizontaler Anker** und **Vertikaler Anker**. Damit konnten Sie festlegen, ob ein Steuerelement beim Verändern der Formulargröße nach rechts oder unten verschoben wird oder ob sogar seine Größe entsprechend angepasst wurde.

Unter WPF übernehmen die beiden Eigenschaften **VerticalAlignment** und **HorizontalAlignment** diese Aufgabe. Die Eigenschaft **HorizontalAlignment** kann die Werte **Left**, **Center**, **Right** und **Stretch** annehmen, die Eigenschaft **VerticalAlignment** die Werte **Top**, **Center**, **Bottom** und **Stretch**. Bild 4 zeigt Beispiele für Schaltflächen, die mit den Werten **Left**, **Center** und **Right** für **HorizontalAlignment** und **Top**, **Center** und **Bottom** für **VerticalAlignment** ausgestattet wurden:

```
<Grid>
    <Button Content="Button" HorizontalAlignment="Left"
VerticalAlignment="Top" />
    <Button Content="Button" HorizontalAlignment="Left"
VerticalAlignment="Center" />
    <Button Content="Button" HorizontalAlignment="Left"
VerticalAlignment="Bottom" />
    <Button Content="Button" HorizontalAlignment="Center"
VerticalAlignment="Top" />
    <Button Content="Button" HorizontalAlignment="Center"
VerticalAlignment="Center" />
```

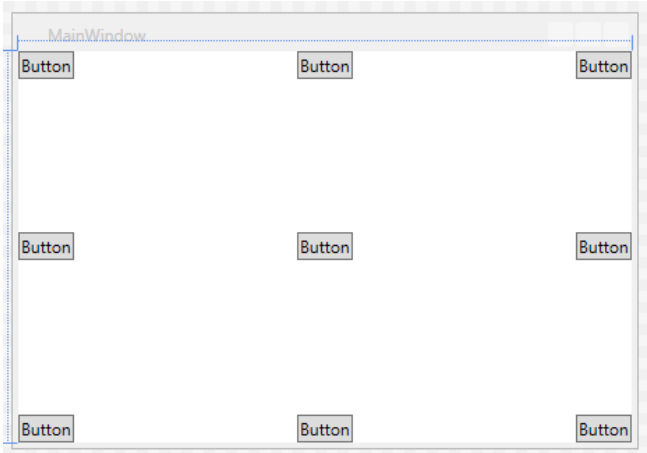


Bild 4: Schaltflächen mit verschiedenen Ausrichtungen

```
    <Button Content="Button" HorizontalAlignment="Center"
VerticalAlignment="Bottom" />
    <Button Content="Button" HorizontalAlignment="Right"
VerticalAlignment="Top" />
    <Button Content="Button" HorizontalAlignment="Right"
VerticalAlignment="Center" />
    <Button Content="Button" HorizontalAlignment="Right"
VerticalAlignment="Bottom" />
</Grid>
```

Mit diesen Einstellungen bleiben die **Button**-Elemente jeweils am Rand kleben, das mittlere in der Mitte.

Die hier nicht verwendete Einstellung ist gleichzeitig die Standardeinstellung: Wenn Sie **HorizontalAlignment** oder **VerticalAlignment** weglassen, verwendet WPF die Einstellung Stretch für das jeweilige Attribut.

Padding

Mit dem Attribut **Padding** legen Sie fest, welchen Abstand der Inhalt eines Steuerelements, also beispielsweise die Beschriftung einer Schaltfläche, von den Rändern haben soll. Mit **Padding="10,10,10,10"** legen Sie also den Abstand des Inhalts von den Rändern des Objekts fest.

Dabei hängt es von den Einstellungen der beiden Eigenschaft **HorizontalContentAlignment** und **VerticalContentAlignment** ab, ob die Einstellungen für **Padding** tatsächlich

relevant sind. Bei einer großen Schaltfläche, deren Beschriftung klein ist und wie folgt ausgerichtet wird, spielt Padding keine Rolle:

```
<Button Content="Button mit einer etwas umfangreicheren
Beschriftung" HorizontalAlignment="Left" Height="50" Mar-
gin="50,50,0,0" VerticalAlignment="Top" Width="200"
                Padding="10,10,10,10" HorizontalContentAli
gnment="Center" VerticalContentAlignment="Center"/>
```

Standardeinstellungen

Neu hinzugefügte WPF-Elemente erhalten je nach der Art des Hinzufügens unterschiedliche Standardattribute. Wenn Sie etwa einfach auf das **Button**-Element im Werkzeugkasten klicken und dann im Fenster per Klick die Position markieren, an der das Steuerelement landen soll, fügt Visual Studio mit **Content** die Beschriftung **Button**, mit **HorizontalAlignment** und **VerticalAlignment** die Werte **Left** und **Top**, mit **Margin** die aktuelle Position und mit **Width** eine Standardbreite von **75** vor.

Was aber geschieht, wenn wir alle Attribute außer **Content** entfernen? In diesem Fall nimmt das **Button**-Steuerelement die komplette Fläche des übergeordneten Elements ein. Das bedeutet also, dass **HorizontalAlignment** und **VerticalAlignment** offensichtlich den Wert **Stretch** als Standardwert einsetzen.

Experimente zum Einüben der Eigenschaften

Am besten finden Sie sich in die verschiedenen Eigenschaften für die Einstellung der Größe und Ausrichtung ein, indem Sie ein wenig experimentieren. Das gelingt in diesem Fall sehr gut, wenn Sie sich auf die Eingabe per XAML konzentrieren. Legen Sie beispielsweise in einem leeren Fenster einmal folgenden Code an:

```
<Window ... Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Button Content="Beispielschaltfläche" />
    </Grid>
</Window>
```

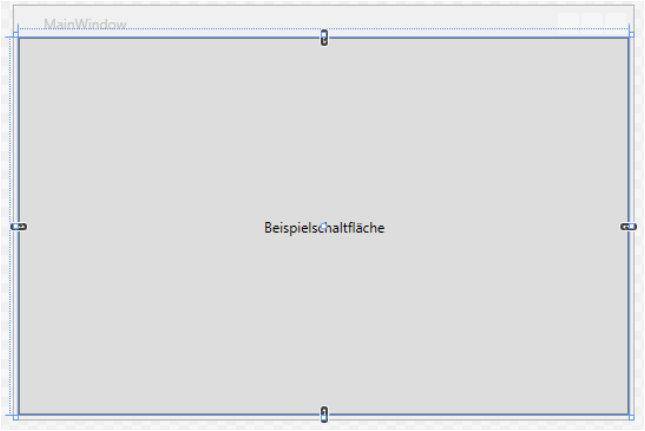


Bild 5: Schaltfläche ohne Attribute für Größe und Position

Dies zeigt eine Schaltfläche an, die sich über das komplette **Grid**-Element im Fenster erstreckt (siehe Bild 5).

Ergänzen Sie nun das Attribut **Margin** wie folgt, bringt dies einen Abstand zum oberen, linken, rechten und unteren Rand wie in Bild 6 ein:

```
<Button Content="Beispielschaltfläche"
    Margin="20,20,20,20"/>
```

Stellen Sie dann die Attribute **HorizontalAlignment** und **VerticalAlignment** auf **Left** beziehungsweise **Top** ein, werden die Eigenschaften für den rechten und unteren Rand im Attribut **Margin** wirkungslos (siehe Bild 7):

```
<Button Content="Beispielschaltfläche"
    Margin="20,20,20,20" HorizontalAlignment="Left"
    VerticalAlignment="Top"/>
```

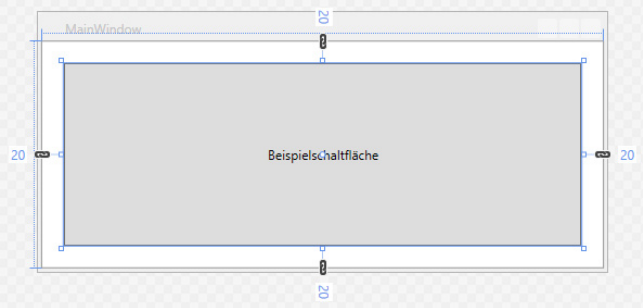


Bild 6: Schaltfläche mit **Margin**

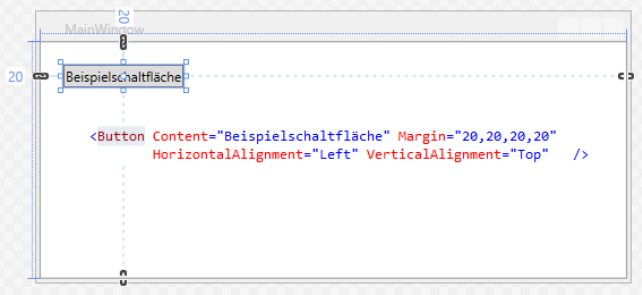


Bild 7: Schaltfläche mit horizontaler und senkrechter Ausrichtung

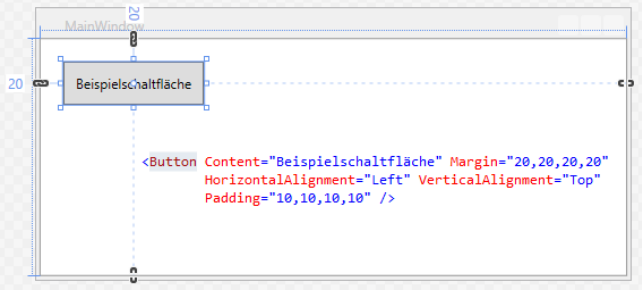


Bild 8: Schaltfläche mit Padding

Bild 9: Schaltfläche mit Padding und zu wenig Platz für die Beschriftung

Nun wollen wir, dass die Schaltfläche etwas luftiger aussieht und vergrößern den Abstand vom Inhalt, also der Beschriftung, zum Rand der Schaltfläche. Dies erledigen wir mit der Eigenschaft **Padding** (Ergebnis siehe Bild 8).

```
<Button Content="Beispielschaltfläche"
    Margin="20,20,20,20" HorizontalAlignment="Left"
    VerticalAlignment="Top" Padding="10,10,10,10" />
```

In Bild 9 sehen Sie schließlich noch, warum eine feste Höhe und Breite sich nachteilig auswirken kann: Durch die **Padding**-Eigenschaft und den dadurch entstehenden Abstand zum Steuerelementrand wird der Platz für die Beschriftung zu eng:

```
<Button Content="Beispielschaltfläche" Margin="20,20,20,20"
    HorizontalAlignment="Left" VerticalAlignment="Top"
    Padding="10,10,10,10" Width="100" Height="32" />
```

Das Grid-Element

Es gibt unter WPF verschiedene Elemente zum Positionieren von Steuerelementen – sogenannte **Panel**-Elemente. Eines davon heißt **Grid**. Dieses haben Sie weiter oben bereits kennen gelernt. Es wird standardmäßig zu neuen Fenstern hinzugefügt. So, wie wir es oben genutzt haben, macht es sich nicht bemerkbar, denn es besitzt nur eine einzige Zelle. Mit der Bezeichnung »Zelle« gelangen wir auch direkt zur prinzipiellen Bauweise eines solchen Grids: Es hat nämlich starke Ähnlichkeit mit einer Tabelle, die ihre Inhalte durch Zeilen und Spalten unterteilt.

Das **Grid**-Element bietet interessanterweise keine Attribute, mit denen Sie etwa die Anzahl der Zeilen und Spalten angeben können. Stattdessen definieren Sie die Struktur über zwei Unterelemente namens **Grid.RowDefinitions** und **Grid.ColumnDefinitions**. Darin legen Sie für jede Zeile ein weiteres Element namens **RowDefinition** beziehungsweise **ColumnDefinition** ab. Das folgende Beispiel ändert zunächst nichts, da wir hier ja nur eine Zeile und eine Spalte definieren:

```
<Grid >
    <Grid.RowDefinitions>
        <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
</Grid>
```

Um eine sichtbare Wirkung zu erzielen, müssten wir also zumindest ein weiteres **RowDefinition**- oder **ColumnDefinition**-Element hinzufügen. Dies erledigen wir gleich:

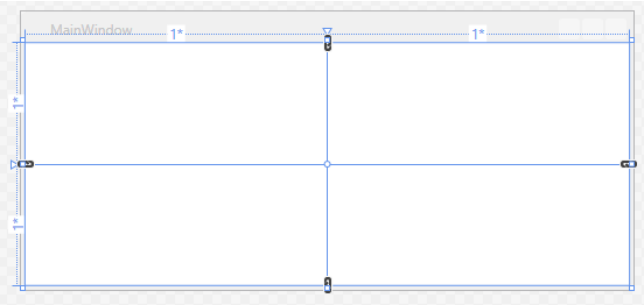


Bild 10: **Grid**-Steuerelement mit zwei mal zwei Zellen

```
<Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
</Grid.ColumnDefinitions>
```

Das Ergebnis finden Sie in Bild 10 – das **Grid**-Element enthält nun zwei mal zwei Zellen.

Steuerelemente zum Grid hinzufügen

Nun wollen wir dem **Grid**-Element einige Steuerelemente spendieren. Nehmen wir an, wir wollen ihm einen Beschreibungstext, zwei Textfelder mit Bezeichnungsfeldern sowie darunter eine Schaltfläche spendieren (siehe Fenster **GridRowsAndColumns.xaml** in der Beispieldatenbank).

Dazu erhöhen wir zunächst die Anzahl der Zeilen durch Hinzufügen zweier **RowDefinition**-Elemente auf vier.

Außerdem fügen wir die Steuerelemente durch Markieren des Steuerelements im Werkzeugkasten und anschließendes Aufziehen eines Rahmens mit der Maus an den Stellen ein, die wir für diese vorgesehen haben. Dadurch erscheinen die Steuerelemente auch innerhalb des **Grid**-Elements im XAML-Code.

Neben den üblichen Attributen erscheinen dort nun auch solche wie **Grid.Column**, **Grid.Row**, **Grid.ColumnSpan** und **Grid.RowSpan**. Diese Attribute geben an, in welcher Zeile und Spalte sich die linke obere Ecke des Steuerelements befindet und über wie viele Spalten und Zeilen sich das Steuerelement erstreckt. Wichtig dabei ist, dass die Attribute **Grid.Column** und **Grid.Row** nur gesetzt werden, wenn das Steuerelement sich nicht in der ersten Spalte oder Zeile befindet. Außerdem ist zu beachten, dass der Index für diese Eigenschaften 0-basiert ist.

Das heißt, dass ein Steuerelement mit **Grid.Column = 1** und **Grid.Row = 1** nicht in der linken, oberen Zelle erscheint, sondern in der zweiten Spalte von links und der zweiten Zeile von oben. Ein Element, das in der ersten Zeile von oben und der ersten Spalte von links beginnt und sich bis in die zweite Zeile von oben und die zweite Zeile von links erstreckt, erhält dementsprechend zwar keinen Wert für die Attribute **Grid.Column** und **Grid.Row**, aber **Grid.ColumnSpan = 2** und **Grid.RowSpan = 2**. **Grid.Column** und **Grid.Row** werden aber auch nur nicht gesetzt, weil der Standardwert **0** ist.

Auch hier kann es wieder sinnvoll sein, etwas mit den Elementen und Attributen zu spielen. Also legen wir einmal folgende Konstellation an:

```
<Label Content="Beispiel für ein Grid mit mehreren Zellen"
    Grid.Column="0" Grid.Row="0" Grid.ColumnSpan="2"
    FontSize="24" Margin="10,0,10,0"/>
<Label Content="Textfeld 1:" Grid.Row="1"
    Margin="10,10,10,10"/>
<Label Content="Textfeld 2:" Grid.Row="2"
    Margin="10,10,10,10"/>
```

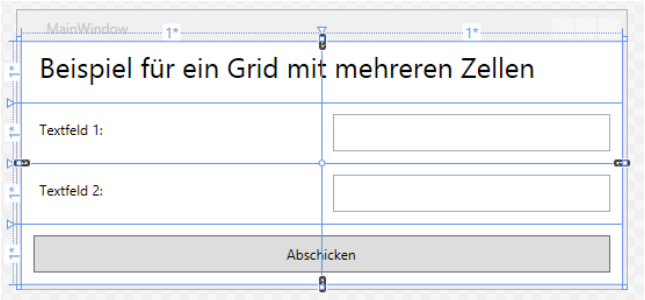


Bild 11: Grid mit einigen Steuerelementen

Daten im DataGrid-Steuerelement anzeigen I

In einem Access-Magazin würde man erstmal die Schaltfläche oder das Textfeld als Steuerelemente vorstellen. Im DATENBANKENTWICKLER wollen wir jedoch von mehreren Seiten an die Programmierung von Datenbank-Anwendungen mit Visual Studio herangehen – unter anderem, indem wir die Nutzung von ADO.NET vorstellen oder die Erstellung von Benutzeroberflächen mit WPF beschreiben. Für beides wollen wir überhaupt einmal die Daten der gewünschten Datenherkunft anzeigen, statt diese immer per Console auszugeben. Dies gelingt am einfachsten mit dem DataGrid-Steuerelement, das stark vereinfacht mit einem Unterformular unter Access in der Datenblattansicht zu vergleichen ist.

Das **DataGrid**-Steuerelement finden Sie, wenn Sie ein Projekt für C#/WPF erstellt haben, direkt in den **Häufig verwendten WPF-Steuerelementen** im **Werkzeugkasten**. Ziehen Sie dieses einfach in den Entwurf eines Fensters und benennen Sie es über die Eigenschaft beispielsweise **ctlDataGrid**. Nun benötigen Sie eine Datenquelle. Dafür verwenden wir die Beispieldatenbank **Suedsturm.mdb**, die wir einfach als Ressource zum Projekt hinzufügen. Dazu ziehen Sie die entsprechende Datei einfach auf das Projekt im Projektmappen-Explorer und stellen dann die Eigenschaft **In Ausgabeverzeichnis kopieren** auf den entsprechenden Wert ein (siehe Bild 1):

- **Nicht kopieren**: Kopiert die Datei niemals in das Ausgabeverzeichnis.
- **Immer kopieren**: Kopiert die Datei immer beim Erstellen der Anwendung in das Ausgabeverzeichnis.
- **Kopieren, wenn neuer**: Kopiert die Datei nur in das Ausgabeverzeichnis, wenn die aktuell im Projekt gespeicherte Version neuer als die Version im Ausgabeverzeichnis ist.

Wenn Sie immer eine unbefleckte Version der Datei benötigen, was beispielsweise bei Tests vorteilhaft ist, wählen Sie **Immer kopieren**. Wenn Sie hingegen in der Anwendung Daten ändern und beim nächsten Erstellen wieder auf die geänderten Daten zugreifen möchten, wählen Sie **Kopieren, wenn neuer**.

Damit das **DataGrid**-Steuerelement nun etwa die Daten einiger Felder der Tabelle **tblArtikel** anzeigt, müssen Sie an

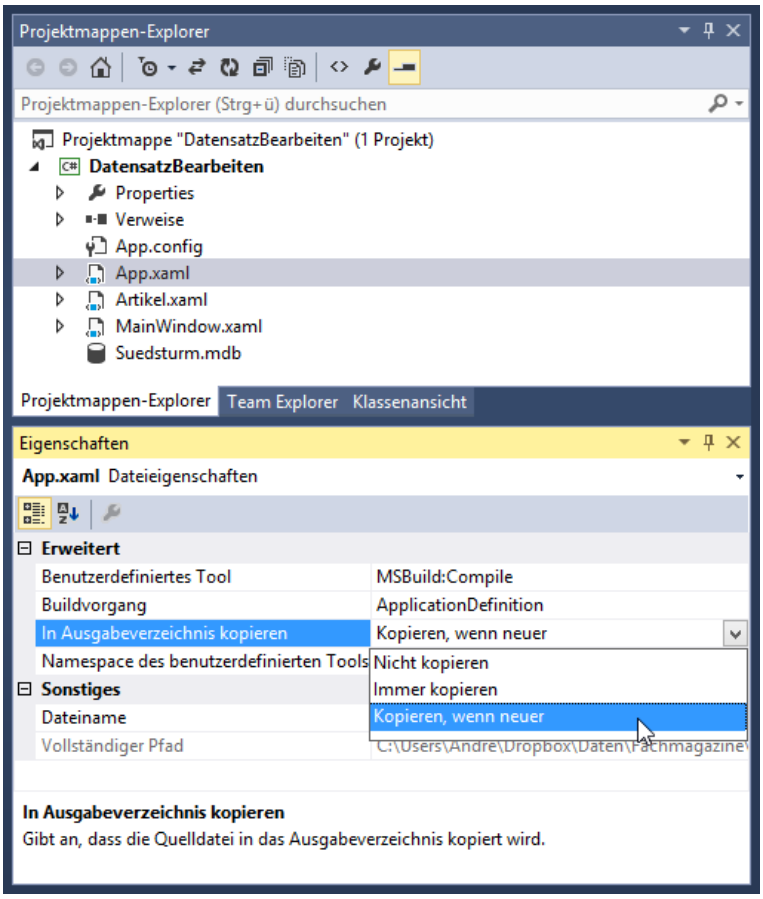


Bild 1: Hinzufügen einer Datenbankdatei zu einem Projekt

den Eigenschaften des **DataGrid**-Steuerelements erstmal nichts ändern. Wir wollen zu Beispielzwecken ein neues Fenster namens **Artikel** erstellen, was wir mit dem Menübefehl **Projekt|Fenster hinzufügen** erledigen. Diesem fügen Sie dann wie in Bild 2 ein **DataGrid**-Steuerelement hinzu.

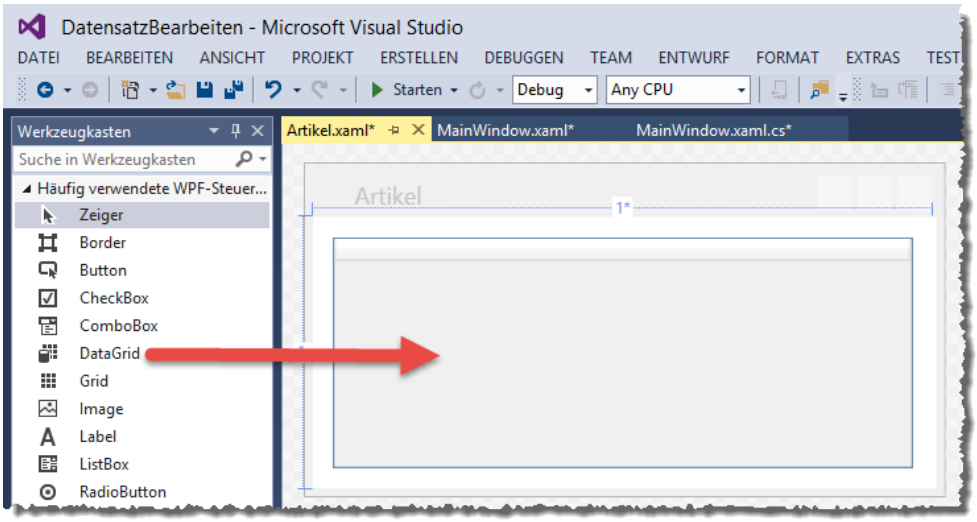


Bild 2: Hinzufügen des **DataGrid**-Steuerelements

Fenster aufrufen

Beim Starten des Projekts mit dem entsprechenden Menübefehl zeigt dieses automatisch das Fenster **MainWindow.xaml** an. Diesem fügen wir nun eine Schaltfläche hinzu, mit der Sie unser Beispielfenster **Artikel.xaml** öffnen können.

Hinterlegen Sie für diese Schaltfläche namens **btnArtikelAnzeigen** die folgende Ereignisprozedur (Schaltfläche markieren, dann im Eigenschaftsfenster auf den Blitz klicken, dann Doppelklick in das Eigenschaftsfeld **Click** – das Ergebnis sieht wie in Bild 3 aus):

```
private void btnArtikelAnzeigen_Click(object sender,
        RoutedEventArgs e) {
    Artikel wnd = new Artikel();
    wnd.Show();
}
```

Diese Prozedur erzeugt in der Variablen **wnd** eine neue Instanz des Fenster-Objekts und blendet dieses mit der **Show**-Methode ein. Ein Klick auf Starten öffnet das Fenster **MainWindow**, von dem aus Sie mit der Schaltfläche **Artikel anzeigen** das gewünschte Fenster mit dem **DataGrid**-Steuerelement öffnen können. Dieses zeigt erwartungsgemäß zunächst keine Daten an.

Einfache Daten anzeigen

Das ändert sich allerdings, wenn wir dem Fenster **Artikel.xaml** eine Schaltfläche namens **btnArtikelInDataGrid** hinzufügen, welche die Ereignisprozedur aus Listing 1 ausführt.

Dies liefert im Fenster **Artikel.xaml** die Ansicht aus Bild 4. Damit die Prozedur funktioniert, benötigen Sie zwei Klassen,

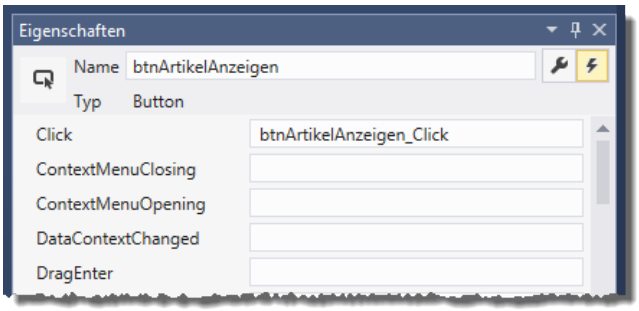


Bild 3: Ereignisprozedur anlegen

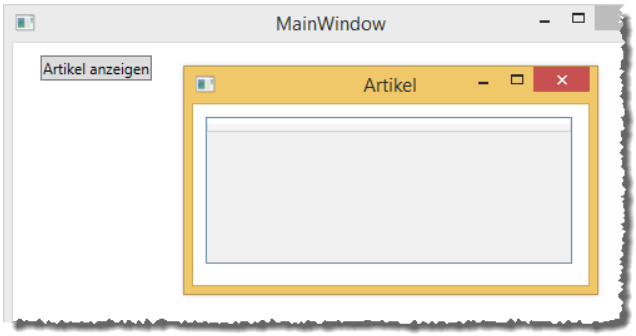


Bild 4: Erster Testlauf

```
private void btnArtikelInDataGrid_Click_1(object sender, RoutedEventArgs e)
{
    string Connectionstring = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Suedsturm.mdb";
    OleDbConnection cnn = new OleDbConnection(Connectionstring);
    OleDbCommand cmd = new OleDbCommand();
    cmd.Connection = cnn;
    cmd.CommandText = "SELECT ArtikelID, Artikelname FROM tblArtikel";
    cnn.Open();
    OleDbDataReader dr = cmd.ExecuteReader();
    DataTable dt = new DataTable();
    dt.Load(dr);
    ctlDataGrid.ItemsSource = dt.DefaultView;
    cnn.Close();
}
```

Listing 1: Füllen eines **DataGrid**-Steuerelements mit den Werten einer Tabelle

die Sie mit der **using**-Anweisung im Kopf des Klassenmoduls **Artikel.xaml.cs** einbinden:

```
using System.Data;
using System.Data.OleDb;
```

Die Erste liefert das hier benötigte **DataTable**-Objekt, die Zweite einige weitere Objekte, die speziell für den Zugriff auf eine Access-Datenbank geeignet sind. Wenn Sie hier statt auf eine Access-Datenbank etwa auf eine SQL Server-Datenbank zugreifen wollten, müssten Sie beispielsweise die Klasse **System.Data.Sqlclient** einbinden und später die entsprechenden Objekte verwenden.

Hier aber geht die Prozedur wie folgt vor: Zunächst trägt sie die Verbindungszeichenfolge, welche den Provider und die Datenquelle definiert, in die Variable **Connectionstring** ein. Da die Datei **Suedsturm.mdb** im Ausgabeverzeichnis der Anwendung landet, brauchen wir hier keinen Pfad anzugeben. Danach erstellt die Prozedur für die Variable **cnn** ein neues **OleDbConnection**-Objekt und übergibt beim Erstellen mit dem **new**-Schlüsselwort die Verbindungszeichenfolge aus **Connectionstring**. Für VBA-Entwickler ist dies Neuland: Dort wurde beim Erstellen niemals ein Parameter übergeben. Unter C# und anderen .NET-Sprachen ist dies jedoch möglich. Dann erstellt die Prozedur ein neues **OleDbCommand**-

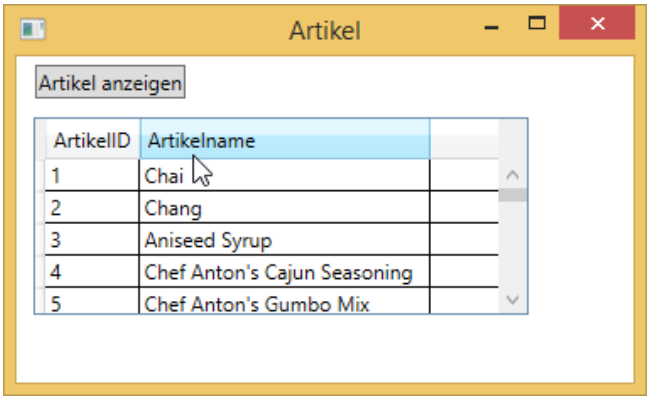


Bild 5: Anzeige der Daten der Tabelle **tblArtikel**

Objekt namens **cmd**, diesmal ohne Parameter. Das Objekt aus **cnn** weisen wir dann der Eigenschaft **Connection** des **Command**-Objekts in **cmd** zu. Die SQL-Anweisung, welche die anzuzeigenden Daten liefern soll, landet hingegen in der Eigenschaft **CommandText**: **SELECT ArtikelID, Artikelname FROM tblArtikel**.

Die **Open**-Methode des Objekts **cnn** öffnet die Verbindung, die **ExecuteReader**-Methode des **Command**-Objekts erstellt ein neues **OleDbDataReader**-Objekt und speichert es in der Variablen **dr**.

Den Inhalt des **OleDbDataReader**-Objekts können wir leider nicht direkt dem **DataGrid**-Steuerelement zuweisen. Dazu benötigen wir noch ein weiteres Objekt des Typs **DataTab-**

Objektorientierte Programmierung: Grundlagen II

Im ersten Teil dieser Artikelreihe haben wir uns die Grundlagen zur objektorientierten Programmierung mit C# angesehen – zum Beispiel die Klasse Program, das Erstellen neuer Klassen, die Verwendung von Eigenschaften und von Methoden. Dies greifen wir im zweiten Teil auf und gehen auf verschiedene Arten der Eigenschaftsdefinition, Konstruktoren, aufgeteilte Klassen, Objektreferenzen, Überladung von Methoden und verschiedene Parameter-Techniken ein.

Öffentliche Variablen oder »Felder«

Wenn eine Eigenschaft einer Klasse nur gelesen und geschrieben werden soll, ohne dass beim Lesen oder Schreiben weitere Aktionen erfolgen sollen, können Sie diese theoretisch einfach als öffentliche Variable deklarieren:

```
class Person_VornameOeffentlich
{
    public string Vorname;
}
```

Sie greifen dann lesend und schreibend auf die Variable zu:

```
public void OeffentlicheVariableTesten()
{
    Person_VornameOeffentlich person =
        new Person_VornameOeffentlich();
    person.Vorname = "André";
    Console.WriteLine("Vorname: {0}", person.Vorname);
}
```

Eine solche Variable/Eigenschaft nennt man auch Feld.

Eigenschaft kapseln

Meist ist aber kein derart unkontrollierter Zugriff auf eine solche Variable erwünscht. Dann deklariert man die Variable als private Variable und erstellt eine öffentliche Eigenschaft, die einen **get**- und einen **set**-Block enthält. In der Kurzform sieht dies so aus:

```
private string vorname;
```

Dies ist die öffentliche Variable mit dem **get**- und dem **set**-Block – **get** liefert den aktuellen Wert der privaten Variablen **vorname** aus und **set** stellt diesen beim Zuweisen eines Wertes für die Eigenschaft auf den Inhalt der Variablen **value** ein:

```
public string Vorname
{
    get { return vorname; }
    set { vorname = value; }
}
```

Damit halten Sie sich die Möglichkeit offen, den Wert der Eigenschaft auch innerhalb der Klasse für andere Zwecke einzusetzen – etwa für Berechnungen weiterer Eigenschaftswerte oder zum Zusammensetzen von Ausdrücken. Gegebenenfalls benötigen Sie dies gar nicht und wollen nur von außen den Wert der Eigenschaft eintragen und diesen später wieder auslesen.

Ein weiterer Vorteil der Kapselung ist: Sie können damit auch festlegen, dass eine Eigenschaft nur schreibend oder nur lesend genutzt werden kann. Wenn die Eigenschaft Vorname beispielsweise nur lesend genutzt werden soll, lassen Sie einfach den **set**-Teil weg:

```
public string Vorname
{
    get { return vorname; }
}
```

Für einen ausschließlich schreibenden Zugriff bleibt hingegen der **get**-Teil außen vor:

```
public string Vorname
{
    set { vorname = value; }
}
```

Getter und Setter schnell erstellen

Benötigen Sie **get**- und **set**-Block, wollen aber keine Einschränkungen vornehmen wie etwa eine Prüfung der Eingabe und benötigen auch keine Variable, auf die Sie von der Klasse aus zugreifen wollen, können Sie diese verkürzte Variante nutzen:

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

Der Zugriff auf die beiden Eigenschaften der Klasse erfolgt dann genau wie bei der vorherigen Version.

Unterstützung durch Visual Studio

Beim schnellern Erstellen von Eigenschaften unterstützt Visual Studio Sie mit Shortcuts. Dazu geben Sie in einer leeren Zeile in der gewünschten Klasse zunächst die vier Buchstaben **prop** ein. Nun erscheinen bereits weiterführende Möglichkeiten in der IntelliSense-Liste (siehe Bild 1). Für uns sind aktuell zwei Varianten interessant. Die erste erfordert die Eingabe von **prop + Tab + Tab** und liefert die folgende Vorlage:

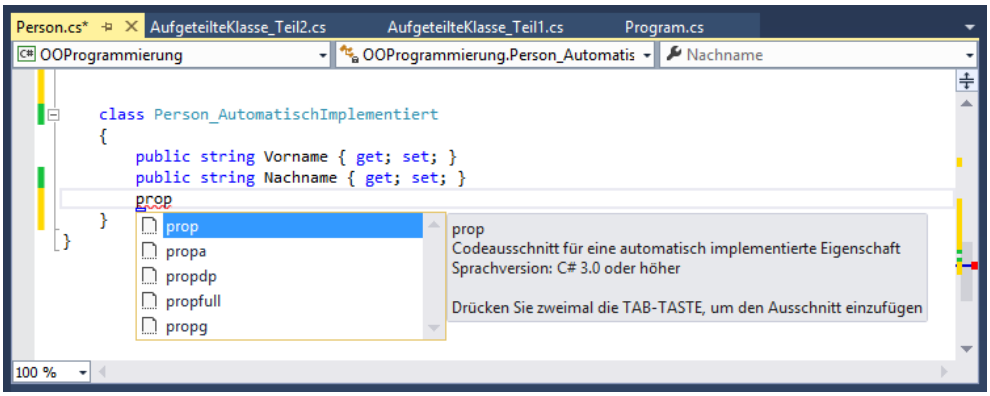


Bild 1: Schnelles Erstellen einer Property

```
public int MyProperty { get; set; }
```

Dabei werden int und **MyProperty** gelb hinterlegt und können mit der Tabulator-Taste durchlaufen werden. Auf diese Weise können Sie die Platzhalter leicht durch den Datentyp und den Namen der Eigenschaft ersetzen.

Die zweite Variante ist die Eingabe von **propfull + Tab + Tab** (es reicht auch **propf + Tab + Tab** aus). Dies liefert die folgende Vorlage:

```
private int myVar;

public int MyProperty
{
    get { return myVar; }
    set { myVar = value; }
}
```

Hier können Sie den Datentyp (**int**), den Variablennamen (**myVar**) und den Eigenschaftsnamen (**MyProperty**) per Tabulatortaste erreichen und ändern.

Klassen mit Konstruktor

Unter VBA gibt es keine Möglichkeit, gleich beim Erstellen einer Klasse einen oder mehrere Werte zu übergeben. Dazu muss man nach dem Erstellen dafür vorgesehene Eigenschaften nutzen. Das ist schon ein Nachteil, wenn für das Verwenden der Klasse bestimmte Pflichtinformationen übergeben werden müssen, bevor man etwa bestimmte Methoden dieser Klasse nutzt.

Das heißt, dass man vor dem Ausführen der Methode immer noch prüfen muss, ob die Pflichtinformationen überhaupt übergeben wurden. Davon abgesehen ist es natürlich auch viel einfacher,

die wichtigsten Eigenschaften gleich beim Initialisieren des Objekts zu übergeben statt mühsam Eigenschaft für Eigenschaft zu übergeben.

Unter C# und anderen objektorientierten Programmiersprachen gibt es deshalb die sogenannten Konstruktoren. Das sind Parameter, die gleich beim Erstellen einer Klasse übergeben werden müssen beziehungsweise können.

Bevor Sie eine Klasse mit einem Konstruktor erstellen, müssen Sie sich überlegen, welche Varianten Sie für die Initialisierung mit oder ohne Parameter zur Verfügung stellen wollen. Eine Klasse **Person** mit den beiden Eigenschaften **Vorname** und **Nachname** kann beispielsweise diese beiden Eigenschaften beim Initialisieren per Parameter entgegennehmen.

Die Definition der Klasse sieht dann etwa wie folgt aus:

```
class Person {
    private string vorname;
    private string nachname;
    public Person() {}
    public Person(string Vorname,
            string Nachname) {
        vorname = Vorname;
        nachname = Nachname;
    }
    public string Vorname {
        get { return vorname; }
        set { vorname = value; }
    }
    public string Nachname {
        get { return nachname; }
        set { nachname = value; }
    }
```

Hier sehen Sie die beiden privat deklarierten Variablen **vorname** und **nachname**. Außerdem sind natürlich die beiden Eigenschaftsmethoden **Vorname** und **Nachname** mit den **get**- und **set**-Blöcken vorhanden.

Neu ist zunächst die folgende Zeile:

```
public Person() {}
```

Dies ist der erste Konstruktor. Er ermöglicht die Initialisierung eines Objekts auf der Basis der Klasse **Person**, ohne direkt eine der beiden Eigenschaften **Vorname** oder **Nachname** übergeben zu müssen. Der Konstruktor hat genau den gleichen Namen wie die Klasse selbst. Der Unterschied zu einer normalen öffentlichen Variablen ist, dass Sie hinter dem Bezeichner erstens ein rundes Klammernpaar (**()**) und dann noch ein geschweiftes Klammernpaar finden (**{}**).

Der zweite Konstruktor erwartet die Übergabe von Werten für die beiden Parameter **Vorname** und **Nachname**. Er heißt ebenfalls genauso wie die Klasse, enthält aber in Klammern eine Liste der Parameter, die beim Initialisieren übergeben werden müssen. Dahinter folgt ein Paar geschweifter Klammern, innerhalb derer die übergebenen Parameter **Vorname** und **Nachname** den Variablen **vorname** und **nachname** übergeben werden:

```
public Person(string Vorname, string Nachname) {
    vorname = Vorname;
    nachname = Nachname;
}
```

Die Eigenschaften der Klasse können Sie nun gleich beim Initialisieren übergeben:

```
Person person;
person = new Person("André", "Minhorst");
Console.WriteLine("{0} {1}", person.Vorname,
    person.Nachname);
//Ausgabe:
//André Minhorst
```

Wir können das Objekt aber, da wir ja auch einen parameterlosen Konstruktor angelegt haben, auch ohne Übergabe von Parametern initialisieren und **Vorname** und **Nachname** erst später übergeben.

Wenn es Pflichteigenschaften gibt, ohne welche die Methoden einer Klasse nicht sinnvoll genutzt werden können, können Sie natürlich auch entscheiden, dass die Eigenschaften auf jeden Fall direkt beim Initialisieren des Objekts gefüllt werden müssen. Dann legen Sie einfach nur einen Konstruktor an, der alle notwendigen Eigenschaften als Parameter erwartet und lassen den parameterlosen Konstruktor weg.

Von Class_Initialize zum Konstruktor

Für alle, die von VBA kommen: Unter VBA gibt es in Klassen die beiden Ereignisprozeduren **Class_Initialize** und **Class_Terminate**. Diese werden, soweit implementiert, beim Initialisieren und beim Zerstören der Klasse ausgelöst – also beim Setzen der letzten Objektvariablen mit einer Referenz auf die Klasse auf den Wert **Nothing**.

Sie raten richtig: Der beziehungsweise die Konstruktoren ersetzen das Ereignis **Initialize** (zum **Terminate**-Ereignis kommen wir später). Das heißt also, dass Sie im Konstruktor auch solche Anweisungen unterbringen sollten, die immer beim Initialisieren der Klasse ausgeführt werden sollen – beispielsweise das Einlesen von Informationen aus Datenbanken, das Erstellen weiterer Objekte et cetera. Wenn eine Klasse keinen Konstruktor benötigt, der beim Initialisieren Parameter zur Verfügung stellt, aber zu diesem Zeitpunkt dennoch Anweisungen ausführen wollen, verwenden Sie den oben bereits vorgestellten Konstruktor ohne Parameter:

```
public Person() {}
```

Diesem können Sie natürlich beliebige Anweisungen hinzufügen:

```
public Person() {
    //... wichtige Anweisungen beim Initialisieren
}
```

Umleitung eines Konstruktoraufrufs

Stellen wir uns nun vor, wir möchten dem Benutzer auch die Möglichkeit geben, nur den Nachnamen der Person beim Initialisieren an die Klasse **Person** zu übergeben. Dazu fügen wir einen weiteren Konstruktor zur Klasse hinzu, der nur diesen einen Parameter verwendet:

```
public Person_KonstruktorUmleiten(string Nachname)
{
    nachname = Nachname;
}
```

Damit haben wir nun eine Möglichkeit mehr, die Klasse zu initialisieren:

```
Person person = new Person("Minhorst");
```

Was hat das nun mit der Umleitung eines Konstruktoraufrufs zu tun? Nun: wir kommen hier wieder auf die Funktion des Konstruktors als die beim Starten ausgelöste Methode zurück, in der Sie Anweisungen zur Vorbereitung des Objekts unterbringen. Diese möchten Sie sicherlich nur in einem Konstruktor unterbringen und nicht in jedem einzelnen. Dazu wählen Sie dann logischerweise den Konstruktor aus, der die wenigsten Parameter entgegennimmt (im Optimalfall gar keinen).

Verwendet die initialisierende Klasse diesen Konstruktor, läuft alles nach Plan. Aber was geschieht, wenn einer der Konstruktoren genutzt wird, der einen oder mehrere Parameter verwendet? Es würde naheliegen, von diesem aus dann den Konstruktor mit den Anweisungen zu starten, die bei jeder Initialisierung aufgerufen werden sollen.

Die Vorgehensweise sieht jedoch etwas anders aus. Der Konstruktor enthält nur die Anweisungen, die bei jedem Initialisierungsvorgang ausgeführt werden sollen:

```
public Person() {
    //... Anweisungen beim Initialisieren
}
```

Der Konstruktor, der nur einen Parameter erwartet, trägt diesen wie gewohnt in die entsprechende Variable ein. Zusätzlich enthält der Kopf der Methode hinter einem Doppelpunkt

das Schlüsselwort **this()**, was bedeutet, dass es auf den Konstruktor ohne Parameter umgeleitet wird:

```
public Person(string Nachname) : this() {
    nachname = Nachname;
}
```

Ein Konstruktor mit zwei Parametern würde dann im Kopf auf den Konstruktor mit dem einen Parameter umleiten – was man an der Angabe des entsprechenden Parameters erkennen kann:

```
public Person(string Vorname,
        string Nachname) : this(Nachname) {
    vorname = Vorname;
}
```

Wenn Sie die Klasse nun mit dem Konstruktor mit einem Parameter initialisieren, löst dies nicht etwa zuerst die Anweisungen dieses Konstruktors aus, sondern die des hinter dem Doppelpunkt angegebenen Konstruktors. Beim Aufruf des Konstruktors mit den zwei Parametern ruft dieser erst den Konstruktor mit dem einen Parameter auf, der dann den Konstruktor ohne Parameter ansteuert. Sprich: Die Anweisungen innerhalb der Konstruktoren werden in umgekehrter Reihenfolge ausgeführt (also: Konstruktor ohne Parameter – Konstruktor mit einem Parameter – Konstruktor mit zwei Parametern).

Unterschied zwischen Konstruktor vs. Eigenschaften und Methoden

Wie kann man nun die Definition eines Konstruktors von der einer Methode oder einer Eigenschaft unterscheiden? Ein Konstruktor enthält niemals einen Datentyp für einen Rückgabewert (**int**, **string** et cetera) oder das Schlüsselwort **void** wie eine Methode ohne Rückgabewert.

Eigenschaften beim Initialisieren übergeben – auch ohne Konstruktor!

Wenn Sie die Eigenschaften einer Klasse bislang immer erst nach dem Initialisieren übergeben haben, möchten Sie nun vielleicht der einen oder anderen Klasse einen Konstruktor hinzufügen, um die Eigenschaften komfortabler zu übergeben. Dies ist jedoch gar nicht unbedingt nötig, denn es gibt noch eine andere Methode, die Eigenschaften gleich in der Zeile mit der **new**-Anweisung unterzubringen.

Dabei tragen Sie hinter dem Paar runder Klammern des Klassennamens eine öffnende geschweifte Klammer ein und fügen dort einfach entsprechende Name-Wert-Paare für die Eigenschaftswerte hinzu – also etwa wie folgt:

```
Person person;
person = new Person() {Vorname="André",
    Nachname="Minhorst"};
```

Mit der folgenden Anweisung greifen Sie dann wie gewohnt auf die Eigenschaften zu:

```
Console.WriteLine("{0} {1}", person.Vorname,
    person.Nachname);
```

Für mehr Komfort betätigen Sie nach der Eingabe der öffnenden geschweiften Klammern einmal die Tastenkombination **Strg + Leertaste**. Das Resultat finden Sie in Bild 2 – alle verfügbaren Eigenschaften werden per IntelliSense angezeigt.

Der Destruktor

Natürlich gibt es auch eine Möglichkeit, Anweisungen beim Zerstören des Objekts auszulösen – und zwar im sogenannten Destruktor. Die Anwendung eines Destruktors und das kontrollierte Zerstören eines Objekts inklusive Freigabe der darin benutzten Ressourcen würde jedoch an dieser Stelle den Rahmen sprengen. Daher gehen wir zu einem späteren Zeitpunkt auf dieses Thema ein.

Klassendefinitionen splitten

Normalerweise sollten Sie der Übersichtlichkeit halber für jede Klasse eine eigene Datei mit der Endung **.cs** erstellen. Sie können natürlich auch mehrere Klassen in einer Klassendatei unterbringen, wie wir es auch sicher in dem einen oder

Objektorientierte Programmierung: Vererbung I

Eines der wichtigsten Merkmale der objektorientierten Programmierung ist die Vererbung. Dank der Vererbung kann man für ein Objekt eine Basisklasse erstellen, die grundlegende Eigenschaften und Methoden für dieses Objekt bereitstellt. Benötigen Sie nun ein weiteres Objekt, das auf dem ersten Objekt aufbaut, aber weitere oder geänderte Eigenschaften und Methoden enthält, müssten Sie ohne Vererbung die gleiche Klasse nochmal erstellen und nach ihren Wünschen anpassen. Dank Vererbung brauchen Sie in der Klasse für das neue Objekt jedoch nur die Erweiterungen und Änderungen zu definieren. Den Rest verwenden Sie von der ersten Klasse – und brauchen eine Menge Code nur an einer Stelle zu warten.

Sie können natürlich, genau wie bei Tabellen einer Datenbank, ein Objekt erstellen, das alle denkbaren Eigenschaften und Methoden liefert (beziehungsweise im Fall einer Tabelle entsprechende Felder enthält). Ein Beispiel wäre eine Tabelle, die zunächst nur Personendaten enthält, die dann aber um Daten für verschiedene Arten von Personen erweitert wird – für Mitarbeiter, Kunden et cetera.

In der Tabelle blieben dann, je nachdem, ob man einen Mitarbeiter oder einen Kunden pflegen möchte, die jeweils für den anderen Personentyp vorgesehenen Felder leer. Damit diese nicht versehentlich gefüllt werden, würde man dafür verschiedene Formulare vorsehen, von denen eines nur die Basisdaten plus die Mitarbeiterfelder und ein anderes nur die Basisdaten plus die Kundenfelder anzeigt.

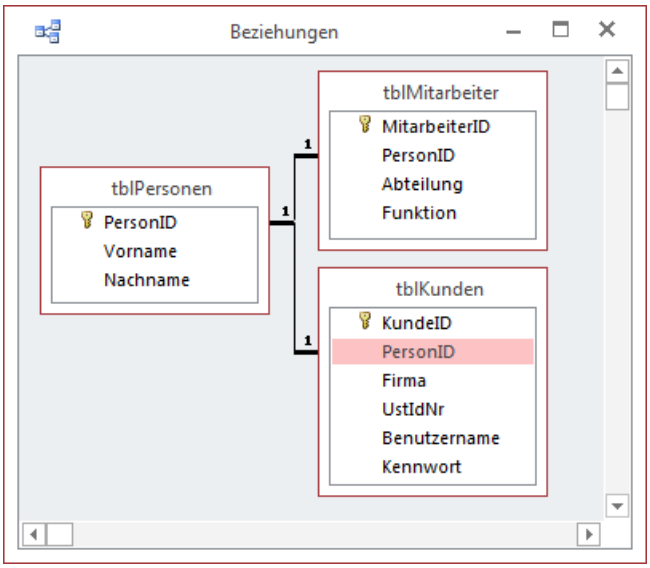


Bild 1: Spezialisierung in einem Datenmodell

Im Datenbankentwurf würde man hier nun eine Tabelle namens **tblPersonen** erstellen und für die jeweiligen Spezialisierungen Tabellen wie **tblKunden** oder **tblMitarbeiter**. Diese würden dann per 1:1-Beziehung miteinander verknüpft werden, damit nicht etwa die Daten eines Datensatzes der Tabelle **tblKunden** mehreren Datensätzen der Tabelle **tblPersonen** zugeordnet werden können und umgekehrt. Das Ergebnis sähe dann etwa in einer Access-Datenbank wie in Bild 1 aus.

In der objektorierten Programmierung sieht es ähnlich aus: Auch hier könnten Sie natürlich eine Klasse entwerfen, die sowohl die Basisdaten einer Person aufnehmen als auch die speziellen Daten für Mitarbeiter oder Kunden. Allerdings wird es dann schwer, bei der Programmierung immer nur genau die für den jeweiligen Typ gültigen Eigenschaften und Methoden zu nutzen, wenn immer alle Member etwa per IntelliSense angeboten werden.

Eine Alternative wäre es, jeweils eine Klasse für die Kunden und eine für die Mitarbeiter zu erstellen. Dabei würden beide jedoch eine Reihe identischer Eigenschaften enthalten wie zum Beispiel **Vorname** und **Nachname**. Und vielleicht gibt es auch noch Methoden, die Sie in beiden Klassen implementieren würden – wie etwa **GeburtstagskarteSchicken**. Diese

müssten Sie dann erstens doppelt anlegen und zweitens Änderungen immer direkt in zwei Klassen durchführen.

Und hier kommt die Vererbung ins Spiel – auch die Ableitung von Klassen genannt: Sie erstellen dabei genau wie bei der Basistabelle für die Personendaten erst einmal eine Basisklasse etwa namens **Person**. Diese stellt Eigenschaften bereit, mit denen Sie Daten wie **Vorname** oder **Nachname** einstellen können.

Wenn Sie nun einen Kunden anlegen möchten, benötigen Sie eine weitere Klasse: nämlich eine solche, die von der Klasse Person erbt beziehungsweise davon abgeleitet ist und die zusätzlich die für einen Kunden spezifischen Eigenschaften und Methoden hinzufügt. Diese wollen wir schlicht **Kunde** nennen. Das Gleiche erledigen wir für die spezielle Ausprägung einer Person namens Mitarbeiter - wir legen eine neue Klasse namens **Mitarbeiter** an, die von der Klasse **Person** erbt und ihre eigenen Eigenschaften und Methoden beisteuert.

Vererbung programmieren

Wir wollen uns an den Beispielfeldern der Tabellen aus dem obigen Datenbankbeispiel orientieren und entsprechende Klassen erstellen.

Wir benötigen also zunächst die Klasse **Person**, die wir wie folgt in der Datei **Program.cs** anlegen:

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

Das ist die Kurzfassung, wenn Sie die Eigenschaften **Vorname** und **Nachname** nur lesen und schreiben möchten, aber kein weiterer Zugriff auf die Werte erforderlich ist. Wenn Sie beispielsweise in einer Eigenschaft namens Bezeichnung den **Vornamen** und den **Nachnamen** zusammengesetzt zurückgeben wollen, müssen Sie folgende Klassendefinition verwenden:

```
class Person {
    private string vorname;
    private string nachname;
    public string Vorname {
        get {
            return vorname;
        }
        set {
            vorname = value;
        }
    }
    public string Nachname {
        get {
            return nachname;
        }
        set {
            nachname = value;
        }
    }
    public string Bezeichnung {
        get {
            return vorname + " " + nachname;
        }
    }
}
```

Hier gibt es also interne Variablen namens **vorname** und **nachname**, welche die über die Eigenschaften **Vorname** und **Nachname** übergebenen Daten speichern. Für beide Eigenschaften legen wir jeweils einen **get**- und einen **set**-Block an, der übergebene Werte in der entsprechenden Variablen speichert oder diesen wieder herausgibt. Die dritte Eigenschaft heißt **Bezeichnung** und soll eine aus den Inhalten von **vorname** und **nachname** zusammengesetzte Zeichenkette zurückliefern.

Die Klasse nutzen Sie beispielsweise wie folgt:

```
public void KlasseOhneVererbung() {
    Person person = new Person();
    person.Vorname = "André";
```

```
    person.Nachname = "Minhorst";
    Console.WriteLine("Bezeichnung: {0}", person.Bezeichnung);
    Console.ReadLine();
}
```

Die Klasse Kunde

Nun erstellen wir die Klasse **Kunde**, welche von der Klasse **Person** erbt. Mit dieser Klasse hat es kaum etwas Besonderes auf sich – sie enthält vier private Variablen, die über die entsprechenden **get**-/**set**-Eigenschaften von außen lesend und schreibend zugreifbar sind. Der feine Unterschied zu einer einfachen Klasse ist jedoch, dass hinter dem Klassennamen ein Doppelpunkt gefolgt vom Namen der Klasse **Person** steht:

```
class Kunde : Person {
    private string firma;
    private string ustIdNr;
    private string benutzername;
    private string kennwort;
    public string Firma {
        get { return firma; }
        set { firma = value; }
    }
    public string UstIdNr {
        get { return ustIdNr; }
        set { ustIdNr = value; }
    }
    public string Benutzername {
        get { return benutzername; }
        set { benutzername = value; }
    }
    public string Kennwort {
        get { return kennwort; }
        set { kennwort = value; }
    }
}
```

Dadurch, dass die Klasse **Kunde** von der Klasse **Person** erbt, finden Sie in der Liste der Eigenschaften und Methoden der Klasse **Kunden** beim Zugriff über IntelliSense nun nicht mehr

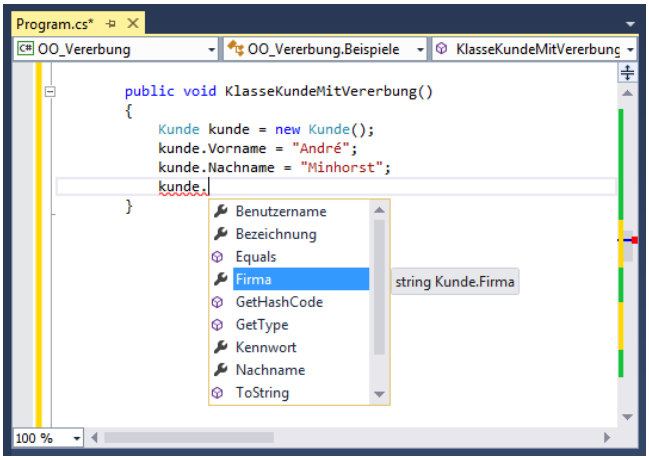


Bild 2: Auswahl der Eigenschaften einer erbenden Klasse

nur die Member der Klasse **Kunden**, sondern auch noch die der Klasse **Person** – also beispielsweise auch **Vorname** oder **Nachname** (siehe Bild 2).

Der Zugriff erfolgt dann durch Deklarieren und Initialisieren des Objekts **kunde** auf Basis der Klasse **Kunde**. Hier ist es dann unerheblich, dass sich hinter der Klasse eigentlich zwei Klassen verbergen:

```
public void KlasseKundeMitVererbung() {
    Kunde kunde = new Kunde();
    kunde.Vorname = "André";
    kunde.Nachname = "Minhorst";
    kunde.Benutzername = "aminhorst";
    kunde.Firma = "André Minhorst Verlag";
    kunde.Kennwort = "geheim";
    kunde.UstIdNr = "DE123456789";
}
```

Die Klasse Mitarbeiter

Auf die gleiche Weise erstellen wir die Klasse **Mitarbeiter**, die ebenfalls von der Klasse **Person** erbt:

```
class Mitarbeiter : Person {
    private string abteilung;
    private string funktion;
    public string Abteilung {
        get { return abteilung; }
```

Experimentieren mit der Konsole

Wenn Sie in die Programmierung mit C# einsteigen, werden Sie – genau wie wir in unseren Beiträgen – eine Menge ausprobieren wollen. Als Umsteiger von VBA haben Sie es da schwer: Unter VBA haben Sie zum Ausprobieren von Code einfach eine neue Prozedur in einem Standardmodul angelegt und diese dann mit F5 gestartet. In Visual Studio verwenden Sie zum Testen von Code zum Beispiel eine Konsolenanwendung. Verschiedene Prozeduren schreiben und mal eben aufrufen wie im VBA-Editor gelingt dort allerdings nicht. Wir zeigen Ihnen, wie Sie dennoch einigermaßen komfortabel experimentieren können.

Eine Konsolenanwendung für C# unter Visual Studio reicht leicht aus, um sich in die Sprache einzuarbeiten. Das Experimentieren damit ist allerdings nicht ganz so einfach wie etwa im VBA-Editor, wo man fernab von Klassenmodulen auch noch Standardmodule zum Programmieren einfacher und sofort aufrufbarer Prozeduren zur Verfügung hatte. Einfach die Prozedur eintippen, die Einfügemarke auf der gewünschten Prozedur platzieren, auf **F5** drücken – los geht es.

Unter C# wird aus den vorhandenen Klassen zunächst eine ausführbare **.exe**-Datei erstellt, die dann aufgerufen wird. Diese **.exe**-Datei hat genau einen Einstiegspunkt, nämlich die Methode **Main** in der Klasse **Program.cs**. Wenn Sie also irgendwelche Codezeilen testen wollen, führt kein Weg daran vorbei, die Konsolenanwendung zu kompilieren, die **.exe**-Datei zu erzeugen und diese zu starten. Gnädigerweise bietet Visual Studio diese Schritte alle über **F5** beziehungsweise den Menüeintrag **Debuggen|Debugging starten** an.

Nun wollen wir aber vielleicht nicht nur eine Prozedur oder, um im C#-Sprachgebrauch zu bleiben, eine Methode verwenden, um unsere vielen Ideen auszuprobieren (und auch in unserer Beispielklasse zu verewigen).

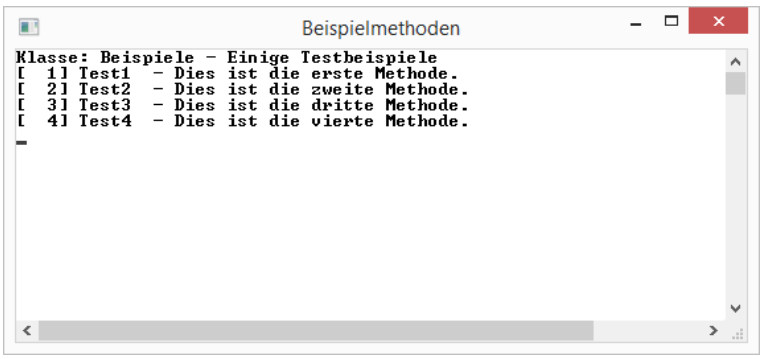


Bild 1: Anzeige der Beispielmethoden einer Klasse

Wie also können wir unsere Beispiele in verschiedene Methoden auslagern und diese dennoch komfortabel testen? Wir verwenden die Einstiegsmethode **Main** genau für diesen Zweck. Dazu fügen wir ihr ein paar Zeilen Code hinzu, welche die verschiedenen Beispiele nach dem Start der Beispielanwendung in der Konsole auflisten – also beispielsweise wie in Bild 1.

Wenn der Benutzer nun eine der in den eckigen Klammern angegebene Zahlen eingibt und die Eingabetaste betätigt, soll die entsprechende Funktion ausgeführt werden. Dies programmieren wir im einfachsten Fall so wie in Listing 1.

Diese Methode stellt zunächst einige Eigenschaften des Konsolen-Fensters ein – zugegebenermaßen hauptsächlich, damit die Screenshots der Artikel in **DATENBANKENTWICKLER** nicht so dunkel daherkommen. Danach durchläuft die Methode die folgenden Anweisungen in einer **do**-Schleife – und zwar so lange, bis die **String**-Variable **methodeNummer** keine leere Zeichenfolge enthält.

Diese Variable nimmt die Benutzereingabe nach der Anzeige der verfügbaren Beispiel-Methoden entgegen, also die Nummer der auszuführenden Methode. Diese Nummern werden samt Methodenname und einer kurzen Beschreibung durch die ersten Anweisungen innerhalb der **do**-Schleife

```
static void Main(string[] args) {
    Console.Title = "Beispielmethoden";
    Console.BackgroundColor = ConsoleColor.White;
    Console.ForegroundColor = ConsoleColor.Black;
    Console.Clear();
    String methodeNummer = "";
    do {
        Console.WriteLine("Klasse: Beispiel");
        Console.WriteLine("[ 1] Test1 - Dies ist die erste Methode.");
        Console.WriteLine("[ 2] Test2 - Dies ist die zweite Methode.");
        Console.WriteLine("[ 3] Test3 - Dies ist die dritte Methode.");
        Console.WriteLine("[ 4] Test4 - Dies ist die vierte Methode.");
        methodeNummer = Console.ReadLine();
        switch (methodeNummer) {
            case "1":
                Beispiele.Test1();
                break;
            case "2":
                Beispiele.Test2();
                break;
            case "3":
                Beispiele.Test3();
                break;
            case "4":
                Beispiele.Test4();
                break;
            default:
                break;
        }
    } while (methodeNummer != "");
}
```

Listing 1: Anzeige der Testfunktionen und deren Aufruf per Eingabe der entsprechenden Zahl

ausgegeben. Nach der Eingabe etwa der Zahl **1** für die erste Methode prüft die Prozedur in einer **switch**-Bedingung den Wert von **methodeNummer**. Im Falle der Zahl **1** landet diese im ersten **case**-Zweig und ruft die Methode **Test1()** der Klasse **Beispiele** auf.

Für die Beispiele legen Sie also dementsprechend eine eigene Klasse namens **Beispiele** an und fügen dort die gewünschten Methoden wie in Listing 2 ein. Die Methoden müssen Sie mit den Schlüsselwörtern **public static** versehen, damit Sie diese ohne Instanzieren eines Objekts auf Basis der Klasse aufrufen können. Die Methoden enthalten lediglich eine einfache **Console.WriteLine**-Anweisung, die Letzte gibt einen **integer**-Wert zurück.

```
public class Beispiele {
        public static void Test1() {
            Console.WriteLine("Test 1");
        }
        public static void Test2() {
            Console.WriteLine("Test 2");
        }
        public static void Test3() {
            Console.WriteLine("Test 3");
        }
        public static int Test4() {
            return 3;
        }
    }
```

Listing 2: Beispiel für eine Klasse mit Testprozeduren

Was ist zu tun? Um neue Beispielmethoden schnell auszuprobieren und die vorhandenen Beispiele ebenfalls im schnellen Zugriff zu haben, müssen Sie also neben einer neuen Methode zwei Dinge erledigen: das Hinzufügen einer **Console.WriteLine**-Anweisung zu Beginn der **do**-Schleife der Methode **Main** sowie eines neuen Zweiges in der **switch**-Bedingung.

Geht es auch weniger aufwendig?

Nachdem ich auf diese Weise einige Beispielmethoden erstellt und über die **Main**-Methode startbereit gemacht habe, stellte ich fest, dass dies doch immer einige Schritte sind. Am schönsten wäre es doch, wenn man einfach nur die neue Beispielmethode anlegen und diese dann einfach in der Liste der Beispielmethoden in der Konsole

vorfinden würde. Das heißt also, dass wir irgendwie dynamisch auf die Methoden zugreifen müssen. Unter VBA hätte man mit den Methoden der Objektbibliothek **Microsoft Visual Basic Extensibility 5.3 Object Library** auf den Quellcode der Beispielmethoden zuzugreifen und so per VBA den Code zu generieren, der für den Aufruf der Methoden nötig ist. Nun wissen wir ja bereits, dass man einfache Routinen in Standardmodulen unter VBA einfach so aufrufen kann, weshalb dieser Aufwand gar nicht nötig ist.

Aber wie können wir uns unter C# behelfen? Die Lösung ist eine Technik namens **Reflection**.

Zugriff auf den Code mit Reflection

Die **Reflection**-Klasse von C# machen Sie mit der entsprechenden **using**-Anweisung verfügbar:

```
using System.Reflection;
```

Danach ändern wir die Methode **Main**, die zuvor noch die manuell erstellte Liste der verfügbaren Beispielmethoden enthielt und diese aufgerufen hat, durch Code, der automatisch zur Laufzeit die Beispielklasse nach Methoden durchsucht und diese in der Konsole ausgibt.

Mehr noch: Auch für den Aufruf der dort angezeigten Methoden ist nach wie vor nur die Eingabe der entsprechenden Nummer notwendig. Die Methode wird dann automatisch aufgerufen – ohne dass wir den Aufruf wie zuvor für jede Methode einzeln in eine **switch**-Bedingung eintragen müssten. Wir investieren nun also einmalig etwas mehr Zeit in eine Methode, lernen ein paar Techniken aus einem Bereich, den man als C#-Anfänger vielleicht sonst erst später berücksichtigen würde und erhalten eine enorme Zeit- und Arbeitsersparnis.

Bevor wir hier einsteigen, schauen wir uns noch eine Erweiterung an, die den Spaß am Testen der Beispiele noch steigern wird: Natürlich kann man ausreichend sprechende Methodennamen für die Beispiele definieren. Aber manchmal ist es doch hilfreich, wenn man noch eine kurze Bemerkung hinzufügen kann – eben so, wie in der Abbildung weiter oben.

Wie aber wollen wir diese in den Methoden unterbringen, sodass diese auch eingelesen und in der Konsole ausgegeben werden kann? Kein Problem: Es gibt sogenannte benutzerdefinierte Attribute für die Elemente einer Klasse. Diese können Sie etwa wie in Listing 3 im Code der Klasse unterbringen, und zwar in dieser Form:

```
[Beschreibung("Einige Testbeispiele")]
public class Beispiele
{
    [Beschreibung("Dies ist die erste Methode.")]
    public static void Test1()
    {
        Console.WriteLine("Test 1");
    }
    [Beschreibung("Dies ist die zweite Methode.")]
    public static void Test2()
    {
        Console.WriteLine("Test 2");
    }
    [Beschreibung("Dies ist die dritte Methode.")]
    public static void Test3()
    {
            Console.WriteLine("Test 3");
    }
    [Beschreibung("Dies ist die vierte Methode.")]
    public static int Test4()
    {
        return 3;
    }
}
```

Listing 3: Ausstatten der Beispiele mit Beschreibungstexten