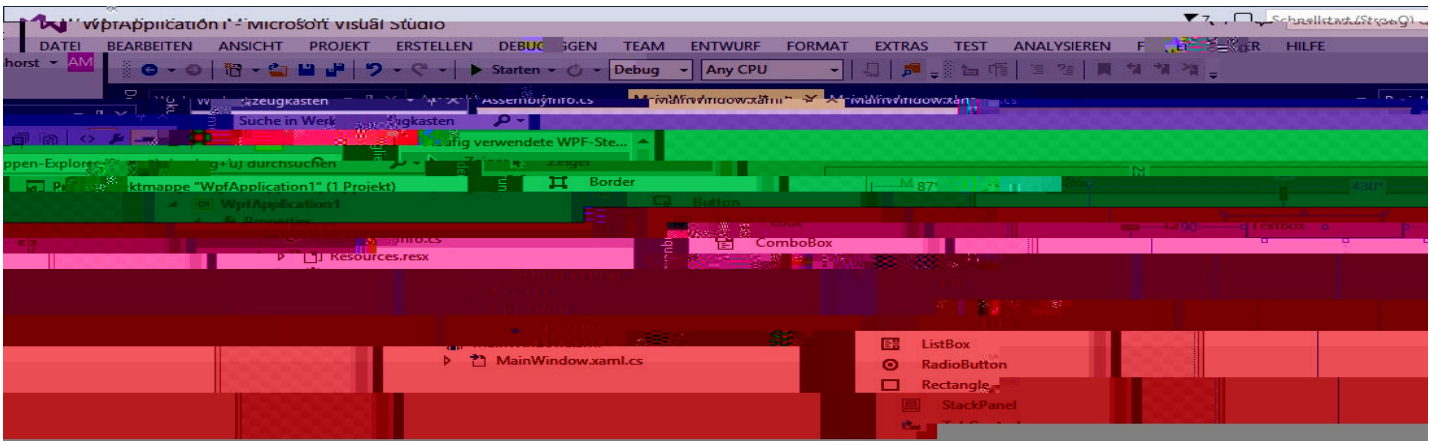
AUSGABE 01/2015

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
Druck: www.booksfactory.de

Einstieg in Visual Studio Community 2013

Mit Visual Studio Community 2013 bietet Microsoft für Selbstständige und kleine Unternehmen eine kostenlose Version seiner Entwicklungsumgebung an, die im Gegensatz zu den Express Editionen vorheriger Versionen keine Einschränkungen aufweist. In der Artikelkategorie »Visual Studio nutzen« stellen wir die Entwicklungsumgebung vor und zeigen nicht nur, wie Sie den Übergang vom VBA-Editor schaffen. Im ersten Teil beginnen wir mit der Installation und ersten Schritten.

Visual Studio Community 2013 installieren

Die Installation birgt keine größeren Herausforderungen in sich. Sie benötigen ein Microsoft-Konto und können die Installation dann direkt von der entsprechenden Microsoft-Webseite ausführen oder die Installationsdateien als DVD-Image herunterladen. Zur Installationsseite finden Sie, da sich die URLs von Zeit zu Zeit ändern, am einfachsten über den Suchbegriff **Microsoft Visual Studio Community 2013 installieren**.

Sprache einstellen

Wenn Sie nicht die englische, sondern die deutsche Version verwenden möchten, sollten Sie gleich noch das entsprechende Language Pack herunterladen und installieren. Wenn Sie danach Visual Studio starten, wählen Sie den Menüeintrag **Extras|Optionen** aus und wechseln im nun erscheinenden Dialog **Optionen** zum Bereich **Internationale Einstellungen**. Hier finden Sie alle aktuell installierten Sprachen vor (siehe Bild 1).

Alternativen zur Entwicklungsumgebung

Theoretisch könnten Sie auch in einem Texteditor entwickeln. Die notwendigen Werkzeuge zum Kompilieren des Codes liefert das Framework mit, Sie müssten diese dann per Eingabeaufforderung steuern. Das macht natürlich keinen Spaß, also beschreiben wir an dieser Stelle auch gar nicht, wie das funktioniert. Zu den Alternativen für andere Betriebssysteme wie Linux, Unix oder OS X kommen wir in späteren Ausgaben.

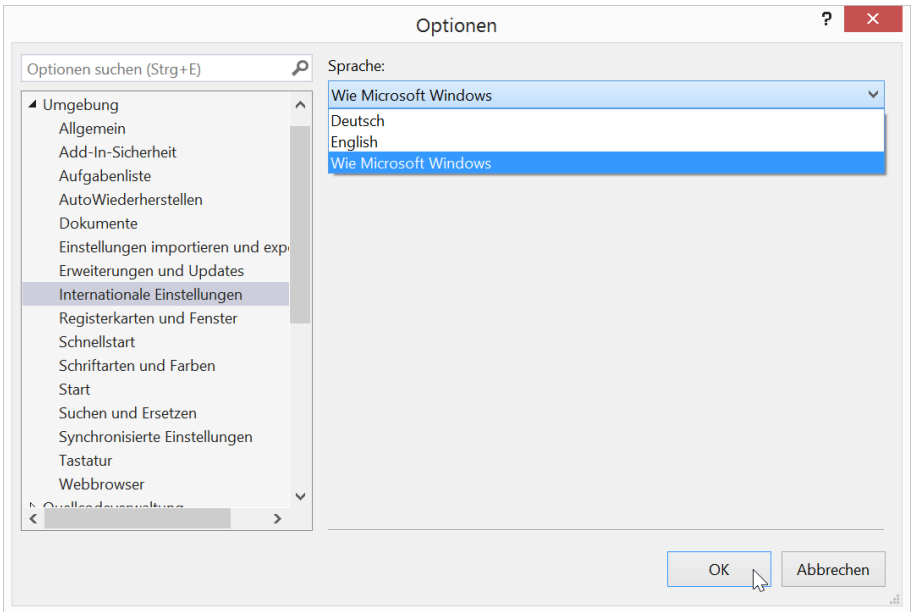


Bild 1: Einstellen der Sprache der Entwicklungsumgebung

Startbildschirm

Nach dem Öffnen von Visual Studio erscheint der Startbildschirm, der etwa die Möglichkeit zum Erstellen oder Öffnen von Projekten anbietet. Außerdem finden Sie hier Informationen über die Entwicklungsumgebung oder Neuigkeiten rund um Visual Studio. Wir wollen uns nicht damit aufhalten, sondern für unsere Aktionen die entsprechenden Menübefehle nutzen.

Elemente der Entwicklungsumgebung

Je nachdem, was für ein Projekt Sie erstellen, zeigt die

Entwicklungsumgebung verschiedene Fensterbereiche an. Diese unterscheiden sich zumeist nur unwesentlich – so erscheint etwa nach dem Erstellen einer Konsolen-Anwendung das Code-Fenster der Hauptklasse, nach dem Erstellen einer WPF-Anwendung finden Sie statt des Code-Editors den XAML-Designer zum Definieren des ersten Formulars vor. Uns interessieren die folgenden Elemente:

- Code-Editor
- Editor für die Benutzeroberfläche von Formularen
- Projektmappen-Explorer
- Eigenschaftsfenster
- Werkzeugkasten
- Fehlerliste

Einige dieser Elemente werden gleich zu Beginn komplett eingeblendet, andere sind über die Seitenleisten erreichbar (siehe Bild 2).

Wenn Sie etwa den Werkzeugkasten einblenden möchten, um die Steuerelemente für die WPF-Formulare ständig greifbar zu haben, klicken Sie links auf den vertikal angeordneten Registerreiter **Werkzeugkasten** und dann nach dem Einblenden des Werkzeugkastens auf das Pin-Icon oben rechts. Das Ergebnis sieht dann etwa wie in Bild 3 aus – der Werkzeugkasten ist nun ständig verfügbar.

Der Projektmappen-Explorer

Der Projektmappen-Explorer sieht etwa wie der Projekt-Explorer im VBA-Editor aus. Allerdings sind in Visual Studio

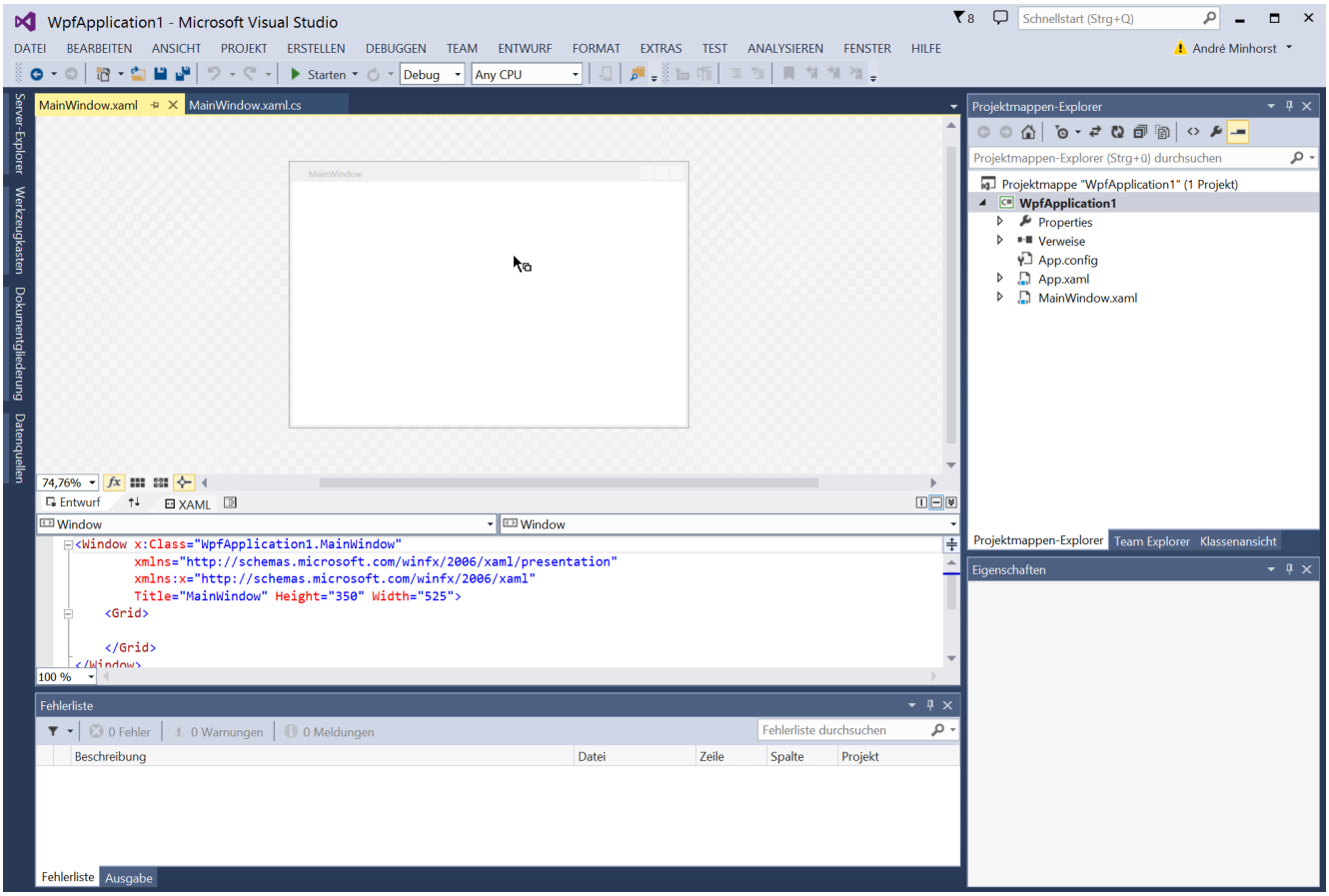


Bild 2: Standardansicht der Entwicklungsumgebung beim Bearbeiten eines WPF-Formulars

ja nicht nur die Werkzeuge zum Bearbeiten des Codes untergebracht, sondern auch die Designer zum Definieren der Benutzeroberfläche der Anwendung. Daher finden sie dort noch eine Reihe weiterer Elemente.

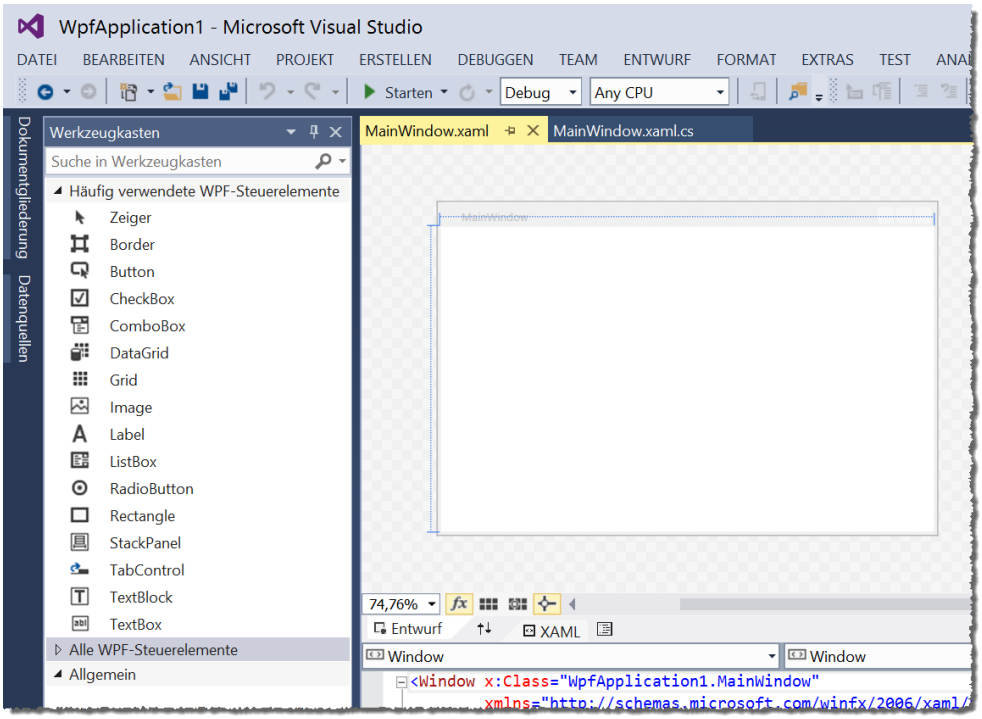


Bild 3: Einblenden und Fixieren des Werkzeugkastens

Unter dem Eintrag **Properties** erhalten Sie beispielsweise drei weitere Elemente:

- **AssemblyInfo.cs**: Nimmt beispielsweise Informationen wie Titel, Beschreibung oder Produktbezeichnung auf. Die in dieser Textdatei gespeicherten Informationen können Sie auch über einen Eigenschaftsdialog bearbeiten, der wie in Bild 4 aussieht. Den Dialog öffnen Sie, indem Sie den Menüeintrag **Projektl<Projektname>-Eigenschaften...** auswählen, im nun erscheinenden Fenster zum Bereich **Anwendung** wechseln und dort auf die Schaltfläche **Assemblyinformationen...** klicken.

- **Resources.resx**: Verwaltung der Ressourcen, also zusätzlicher Dateien. Den Inhalt dieses Bereichs bearbeiten Sie ebenfalls über den Eigenschaften-Dialog der Projektmappe, und zwar im Bereich Ressourcen (siehe Bild 6). Dort fügen Sie Ressourcen wie Bilddateien et cetera hinzu oder bearbeiten diese.

- **Settings.settings**: Bereich zum Festlegen von Anwendungs- oder Benutzereinstellungen

Unter dem Eintrag **Verweise** finden Sie außerdem die aktuell in das Projekt eingebundenen Verweise.

Der Code-Editor

Wenn Sie C#-Code eingeben, werden Sie bemerken, dass Visual Studio nach der Eingabe eines Methodennamens und der öffnenden geschweiften Klammer automatisch die schließende Klammer anfügt. Wenn Sie nun die Eingabe-

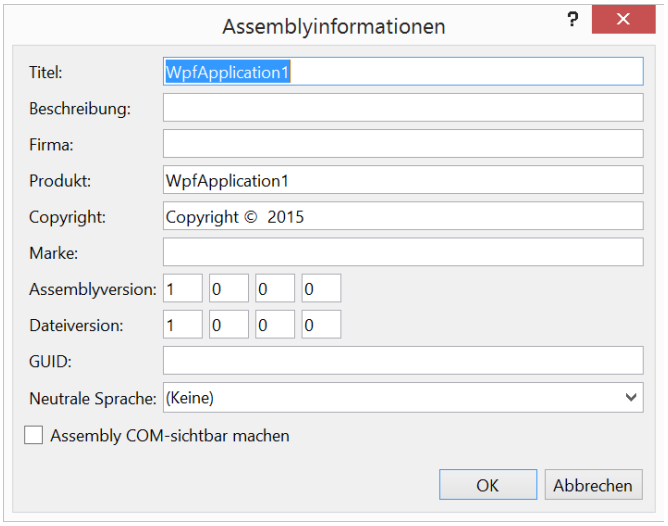


Bild 4: Assemblyinformationen

Von VBA zu C#: Erste Anwendung und Variablen

Wer von VBA zu C# wechselt, muss an einigen Stellen umdenken. Der Sprung von VBA zu Visual Basic unter .NET wäre möglicherweise etwas einfacher, allerdings ist C# wohl die weiter verbreitete Sprache. Das bedeutet nicht nur, dass es mehr Entwickler gibt, die damit entwickelte Anwendungen leichter weiterentwickeln oder Sie unterstützen können, sondern es existieren auch mehr Beispiele zu C# im Internet als zu Visual Basic. Im ersten Artikel der Kategorie »Von VBA zu C#« schauen wir uns an, wie Sie eine erste kleine Konsolenanwendung bauen und welche einfachen Datentypen es dort gibt.

Für die Beispiele dieser Artikelreihe wollen wir jeweils eine Konsolenanwendung verwenden. Damit können wir uns auf die Sprache selbst konzentrieren und müssen uns nicht mit zusätzlichen Elementen wie etwa der Benutzeroberfläche beschäftigen. Eine Konsolenanwendung erstellen Sie über den Menübefehl **Datei|Neu|Projekt**. Dies zeigt den Dialog **Neues Projekt** an, wo Sie das Zielverzeichnis und den Namen des Projekts angeben sowie die Projektvorlage auswählen – in diesem Fall **Vorlagen|Visual C#|Windows-Desktop|Konsolenanwendung**.

Ein Klick auf die Schaltfläche **OK** legt dann ein neues Projekt an, das im Wesentlichen aus einem Klassenmodul namens **Program.cs** besteht. Klassenmodule sind Ihnen, wenn Sie bereits mit VBA programmiert haben, ein Begriff – dort gab es die zu Formularen oder Berichten gehörenden Klassenmodule oder alleinstehende Klassenmodule, die Sie ähnlich wie Standardmodule angelegt haben.

Unter VBA war der wesentliche Unterschied zwischen Klassenmodulen und Standardmodulen, dass Sie zunächst ein Objekt auf Basis des Klassenmoduls erstellen mussten, um auf die enthaltenen Elemente wie etwa Methoden oder Eigenschaften zuzugreifen, während Sie die in Standardmodulen enthaltenen Prozeduren oder Funktionen direkt etwa über den Direktbereich des VBA-Editors aufrufen konnten.

Von der Idee der Standardmodule können Sie sich nun verabschieden, denn unter C# gibt es nur noch Klassenmodule. In unserem neuen Projekt heißt das aktuell einzige Exemplar **Program.cs** und sieht wie in Bild 1 aus. Was es mit den **using**-Anweisungen und den Elementen **namespace**, **class** und **static void** auf sich hat, schauen wir uns weiter unten an.

Erster Test

Die Konsolenanwendung heißt so, weil die Ausgabe in der Konsole erfolgt, also in einem Fenster wie der Eingabeaufforderung. Um dort einen Text auszugeben und das Fenster durch Betätigen der Eingabetaste wieder zu schließen,

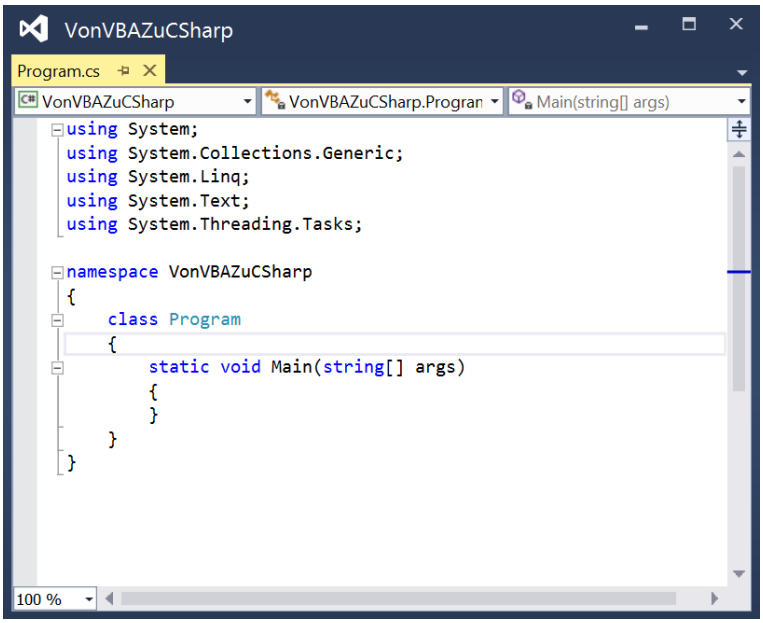


Bild 1: Standardmäßig vorhandenes Klassenmodul in einem neuen C#-Projekt

ergänzen Sie die Prozedur, die mit der Zeile **static void Main...** beginnt, durch zwei Zeilen, die Sie wie folgt zwischen den geschweiften Klammern einfügen:

```
static void Main(string[] args)
{
    Console.WriteLine("Eingabetaste drücken!");
    Console.ReadLine();
}
```

Ein Klick auf die Taste **F5** oder die Auswahl des Menüeintrags **Debuggen|Debugging starten** sorgt für die Anzeige des Fensters aus Bild 2, das Sie durch Betätigen der Eingabetaste wieder schließen.

Der erste Befehl, die Methode **WriteLine** des Objekts **Console**, hat also zunächst für die Anzeige des Textes in der Konsole gesorgt. Damit die Konsole danach nicht direkt wieder verschwindet, haben wir die **ReadLine**-Methode des **Console**-Objekts hinterhergeschickt. Diese unterbricht den Ablauf des Programms, bis der Benutzer die Eingabetaste betätigt.

Alternativ zu **ReadLine** hätten wir auch die Methode **ReadKey** nutzen können, die auf die Betätigung einer jeden Taste reagiert:

```
Console.ReadKey();
```

Semikolon als Zeilenabschluss

Sie haben es schon erkannt: Unter C# müssen Sie jede Anweisung mit einem Semikolon beenden. Es reicht nicht aus, wie unter VBA einfach die neue Anweisung in die nächste Zeile zu schreiben. Hier ist das Semikolon das Maß aller Dinge bei der Kennzeichnung des Endes einer Anweisung. Dafür können Sie unter C# auch mehrere Anweisungen in eine Zeile schreiben – Sie müssen diese nur durch das Semikolon voneinander trennen.

Dies sähe dann für unser Beispiel so aus:

Bild 2: Anzeige der Konsole mit dem gewünschten Text

```
Console.WriteLine("Eingabe...!");Console.ReadLine();
```

Zeilen umbrechen

Durch den zwingenden Zeilenabschluss per Semikolon können Sie Zeile im Gegensatz zu VBA etwas komfortabler aufteilen – zum Beispiel so:

```
Console.WriteLine(
    "Eingabetaste drücken!");
```

Sie müssen also nicht etwa das Unterstrich-Zeichen angeben, um den Zeilenumbruch zu markieren. Leider gelingt das nicht mitten in einem Literal, also innerhalb von Anführungszeichen. Folgendes führt zu einem Kompilierfehler, der übrigens rot unterstrichen markiert wird:

```
Console.WriteLine("Eingabetaste
    drücken!");
```

Außerdem können Sie sich bei dieser Gelegenheit gleich mit der Fehlerliste von Visual Studio bekanntmachen, die Sie sicher noch häufiger konsultieren werden – und zwar mit dem Menüeintrag **Ansicht|Fehlerliste** oder der Tastenkombination **Strg + ^, E** (also erst **Strg + ^**, dann **E**).

Ungewohnte Klammern

Nicht nur, dass Sie sich mit geschweiften Klammern herumschlagen müssen, die einen Codeblock beispielsweise innerhalb eines Namespaces, einer Methode oder auch eines Konstrukts wie einer Schleife oder einer Bedingung markieren – C# verlangt auch nach allen möglichen Methoden nach der Angabe eines Klammernpaares. Dies geschieht unabhängig davon, ob Parameter zu übergeben sind oder nicht. Sollten Ihnen also einmal ein Syntaxfehler

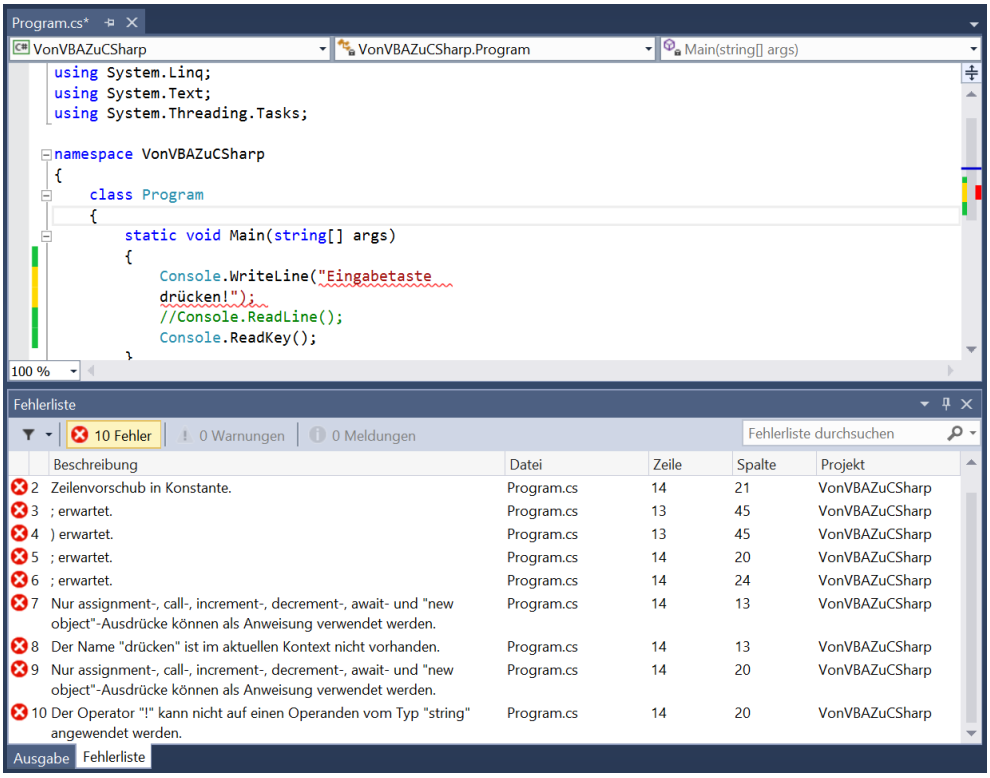


Bild 3: Fehlermeldungen

über den Weg laufen, den Sie sich auf den ersten Blick nicht erklären können, prüfen Sie, ob die Anweisung eventuell nach einem Paar abschließender Klammern verlangt.

Fehlerliste

Diese sieht in diesem Fall wie in Bild 3 aus. Mehr Syntaxfehler kann man wohl kaum per Einzeiler provozieren. Sie können diese Anzeige jedoch zum Anlass für die erste Erkundung der Fehlerliste nutzen, die es im VBA-Editor ja gar nicht gibt: Diese zeigt genau, wo der Fehler vorliegt und um was für einen Fehler es sich handelt. Im Gegensatz zum VBA-Editor, wo etwa beim Debuggen nur ein Syntaxfehler nach dem anderen moniert wird, finden Sie hier gleich eine Liste aller aktuell vorliegenden Syntaxfehler.

Noch schöner ist, dass Sie per Doppelklick auf einen der Einträge direkt zu der fraglichen Stelle im Code springen können. Darüber hinaus zeigt die Fehlerliste auch die Datei (also in diesem Fall die Klasse), die Zeile und die Spalte des Fehlers an. Sie könnten sich also auch über die Zeilennummer den Fehler anschauen – vorausgesetzt, dass das Codefenster gerade

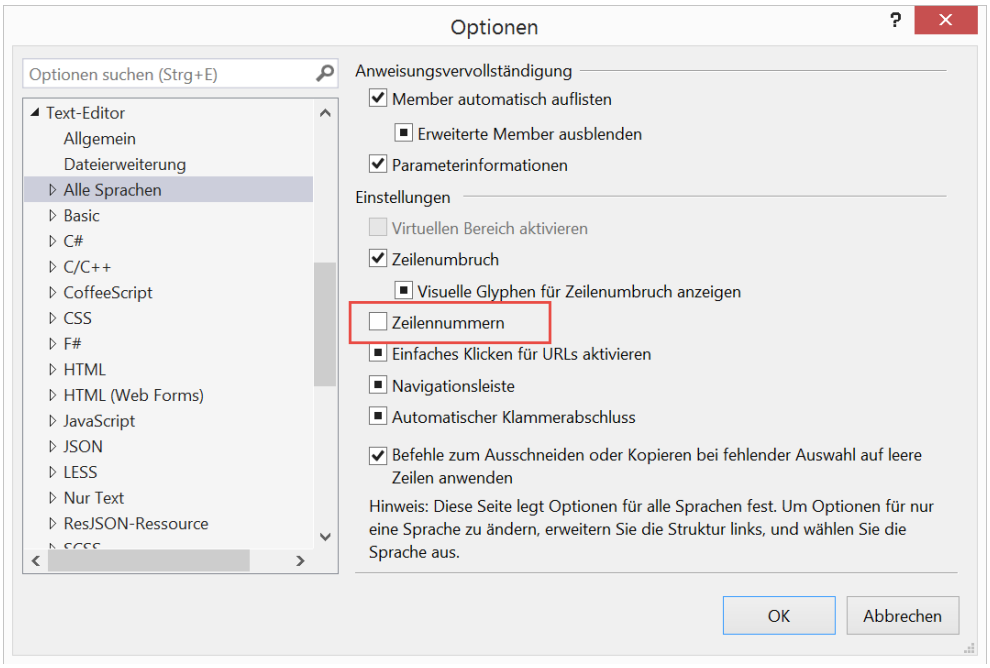


Bild 4: Einblenden der Zeilennummern

die Zeilennummern anzeigt. Im Screenshot ist dies nicht der Fall, aber Sie blenden die Zeilennummern ganz einfach mit der Option **Zeilennummern** in den Visual Studio-Optionen (**Extras|Optionen**) im Bereich **Text-Editor|Alle Sprachen** ein (siehe Bild 4).

Im vorliegenden Fall beheben Sie die knapp zehn Fehler übrigens, indem Sie den Zeilenumbruch innerhalb der Zeichenkette wieder entfernen.

Groß und klein

Wenn Sie schnell einen Syntaxfehler provozieren möchten, brauchen Sie nur einmal **WriteLine** mit ausschließlich kleinen Buchstaben zu schreiben. Visual Studio korrigiert dies nicht – zumindest dann nicht, wenn Sie die IntelliSense-Option nicht nutzen.

Stattdessen zeigt die Fehlerliste den Fehler **"System.Console" enthält keine Definition für "writeline".** an.

Die Groß- und Kleinschreibung ist also wichtig – Visual Studio erkennt Elemente der Objektbibliothek nicht, wenn diese nicht hundertprozentig richtig geschrieben sind.

Dies gilt übrigens auch für die Benennung von Objekten, Variablen et cetera: Sie können durchaus zwei Variablen namens **kundeid** und **KundeID** verwenden. Durch die unterschiedliche Groß- und Kleinschreibung werden diese als verschiedene Variablen erkannt.

Blockweise

C#-Code ist, bis auf die oberste Ebene, immer in Blöcke eingefasst. Ein Block wird durch eine öffnende geschweifte Klammer vor dem Block und eine schließende geschweifte Klammer hinter dem Block gekennzeichnet. Blöcke stehen meist in einem bestimmten Kontext, beispielsweise als Inhalt einer Methode, eines Namespaces, einer Schleife oder einer Bedingung.

Sie können aber auch einfach so einen Block mit geschweiften Klammern zusammenfassen. Hier ein sinnloses Beispiel, dass aber zeigt, wie es aussehen kann:

```
static void Main(string[] args)
{
    Console.WriteLine("Eingabetaste drücken!");
    {
        Console.ReadLine();
    }
}
```

Kommentare

Einzeilige Kommentare leiten Sie mit zwei Schrägstrichen ein:

```
//Kommentarzeile
```

Im Gegensatz zu VBA können Sie auch mehrere Zeilen zu einem Kommentarbereich zusammenfassen:

```
/*Mehrzeiliger
Kommentar*/
```

Wie unter VBA können Sie auch einen Kommentar hinter einer Zeile anbringen:

```
Console.WriteLine("Blabla"); //
Kommentar
```

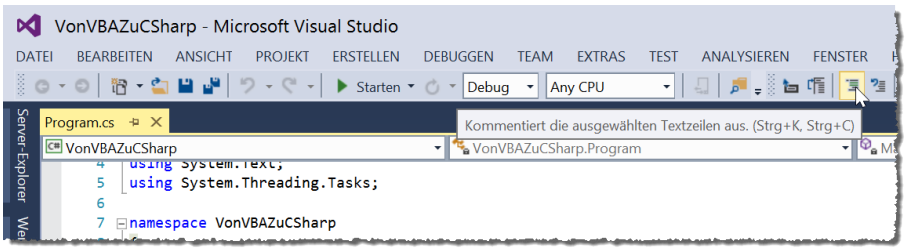


Bild 5: Symbolleistenbefehl zum Auskommentieren von Codezeilen

Schnell auskommentieren

Wenn Sie unter VBA schnell einmal ein paar Zeilen gleichzeitig auskommentieren wollten, haben Sie die Symbolleiste **Bearbeiten** ein-

Von VBA zu C#: Operatoren

Die Anzahl der Operatoren unter VBA ist überschaubar. Die .NET-Programmiersprachen bieten hier schon eine ganze Menge neuer Möglichkeiten. Zum Beispiel brauchen Sie, wenn Sie einen Zähler um eins erhöhen wollen, nicht mehr x = x + 1 zu schreiben, sondern einfach nur x += 1. Diese und viele andere Operatoren stellt der vorliegende Artikel vor.

Operatoren

Es gibt verschiedene Arten von Operatoren, die wir in die folgenden Kategorien einteilen:

- Zuweisungsoperatoren
- Arithmetische Operatoren
- Vergleichsoperatoren
- Logische Operatoren
- Andere Operatoren

Beispiele

Wenn Sie die nachfolgend vorgestellten Beispiele ausprobieren möchten, erstellen Sie einfach eine C#-Konsolenanwendung. Fügen Sie die Anweisungen in die Methode **static void Main()** ein und führen Sie diese durch Betätigen der Taste **F5** aus.

Zuweisungsoperatoren

Als Erstes schauen wir uns die Operatoren an, mit denen Sie einer Variablen einen neuen Wert zuweisen können:

- **x = y**: **x** erhält den Wert von **y**.
- **x += y**: **y** wird zu **x** addiert und in **x** gespeichert (entspricht unter VBA dem Ausdruck **x = x + y**)
- **x -= y**: **y** wird von **x** subtrahiert (**x = x - y**)
- **x *= y**: **x** wird mit **y** multipliziert (**x = x * y**)
- **x /= y**: **x** wird durch **y** dividiert (**x = x / y**)
- **x %= y**: Liefert den Rest von **x/y** (**x = x mod y**)
- **x &= y**: Bitweise AND-Operation (mehr weiter unten – unter VBA **AND**)
- **x |= y**: Bitweise OR-Operator (mehr weiter unten – unter VBA **OR**)
- **x ^= y**: XOR-Operator (mehr weiter unten)
- **x <<= y**: Verschiebt **x** um **y** bitweise nach links.
- **x >>= y**: Verschiebt **x** um **y** bitweise nach rechts.
- **=>**: Lambda-Operator. Darauf gehen wir später ein.

Bitweise Vergleichsoperationen

Die Auflistung oben enthielt einige bitbezogene Operatoren. Diese vergleichen die einzelnen Bits von Zahlenwerten und liefern das entsprechende Ergebnis. Wenn Sie also beispielsweise die Zahl **6** (binär **110**) und **2** (**10**) vergleichen, erhalten Sie bei der **AND**-Operation den Wert **2** (also **10**), weil nur das zweite Bit bei beiden Zahlenwerten gesetzt ist.

Bei einer **OR**-Operation mit den Zahlen **4** (**100**) und **1** (**1**) käme der Wert **5** heraus (**101**).

Die **XOR**-Operation liefert nur solche Bits, die sich bei beiden Operatoren unterscheiden. Der Vergleich von **7** (**111**) und **3** (**11**) würde also **4** ergeben (**100**).

Die bitweise Verschiebung mit den Operatoren **<<=** und **>>=** sorgt dafür, dass die Bits um die angegebene Anzahl nach links oder rechts verschoben werden:

```
Console.WriteLine(5 << 1);
//Liefert: 10
```

5 entspricht binär dem Wert **101**. Wenn Sie **101** um eine Position nach links verschieben, wird der freie Platz rechts mit dem Wert **0** aufgefüllt, Sie erhalten also **1010**, was dem dezimalen Wert **10** entspricht.

Verschieben Sie **101** hingegen um eins nach rechts, erhalten Sie **10** – aus **5** wird so also **2**:

```
Console.WriteLine(5 >> 1);
//Liefert: 2
```

Arithmetische Operatoren

Unter den arithmetischen Operatoren verstehen wir die Grundrechenarten sowie einige weitere. Die Operatoren für Addition (**+**), Subtraktion (**-**), Multiplikation (*****) und Division (**/**) brauchen wir nicht weiter zu erläutern.

Das Plus- und das Minuszeichen dienen außerdem als Vorzeichen.

Daneben gibt es noch das Prozentzeichen (**%**), mit dem Sie den Rest einer Division ermitteln:

```
x = 5;
y = 2;
Console.WriteLine("Der Rest von 5/2 ist: {0}", y % x);
//Liefert 1
```

Beim Dividieren zweier Zahlen vom Datentyp **int** müssen Sie beachten, dass das Ergebnis wieder den gleichen Datentyp hat. Im folgenden Beispiel wird aus **5/2** dann **2** statt des erwarteten Wertes **2.5**:

```
int x = 5;
int y = 2;
Console.WriteLine("5/2 ist: {0}", y / x);
//Liefert 2
```

Der Grund ist, dass für das Ergebnis einer Rechenoperation der größte Datentyp verwendet wird – und der ist in diesem Fall **int**. Dementsprechend gibt die Konsole den Wert **2** statt **2.5** aus.

Wenn Sie mindestens einen der beiden Werte als Datentyp mit Nachkommastellen deklarieren, erhalten Sie das gewünschte Ergebnis:

```
int e = 5;
double f = 2;
Console.WriteLine("5/2 ist: {0}", e/f);
//Liefert 2.5
```

Interessanterweise tritt das gleiche Problem auf, wenn Sie ohne Variablen arbeiten und die Gleichung mit konkreten Zahlenwerten angeben:

```
Console.WriteLine("5/2 ist: {0}", 5 / (double)2);
//Liefert 2
```

Dies können Sie ändern, indem Sie einen der Werte explizit als **double** deklarieren. Dies erledigen Sie durch Voranstellen des Schlüsselworts **(double)** in Klammern:

```
Console.WriteLine("5/(double)2 ist: {0}", 5 / (double)2);
//Liefert 2.5
```

Alternativ geben Sie eine echte Dezimalzahl an, also mit Nachkommastelle:

```
Console.WriteLine("5/2.0 ist {0}", 5 / 2.0);
//Liefert 2.5
```

Inkrement und Dekrement

Der Operator **++** vor einer Zahl oder einer Variablen addiert **1** zum Wert hinzu (Inkrementoperator):

Von VBA zu C#: Bedingungen

Unter Access-VBA haben Sie als Bedingungen die Befehlsstrukturen If...Then und deren Varianten sowie Select Case kennen gelernt. Außerdem gibt es noch Funktionen, mit denen sich bestimmte Werte in Abhängigkeit des Wertes eines Parameters zurückgeben lassen – zum Beispiel IIf oder Choose. Unter C# sieht dies etwas anders aus, vor allem wegen der Strukturierung mit geschweiften Klammern und des fehlenden End-Schlüsselwortes. Dieser Beitrag stellt die Pendants zu den VBA-Bedingungen vor.

Einfaches If...Then

Die übliche **If...Then**-Bedingung bauen Sie in VBA wie folgt auf:

```
If intZahl = 0 Then
    MsgBox "Die Zahl lautet Null."
End If
```

Unter C# können Sie diese Bedingung sogar noch einfacher schreiben:

```
if (Zahl == 0)
    Console.WriteLine("Die Zahl lautet Null.");
```

Wenn Sie diese Anweisungen ausprobieren möchten, erstellen Sie eine neue C#-Konsolenanwendung mit Visual Studio und ergänzen die Main-Prozedur wie folgt:

```
static void Main(string[] args) {
    int Zahl;
    Console.WriteLine("Geben Sie eine ganze Zahl ein.");
    Zahl = Convert.ToInt32( Console.ReadLine());
    if (Zahl == 0)
        Console.WriteLine("Die Zahl lautet Null.");
}
```

Das Ergebnis sieht dann etwa wie in Bild 1 aus. Wieso aber gelingt dies mit nur zwei Zeilen – und auch noch ohne geschweifte Klammern? Nun: Dies ist eine vereinfachte Form, bei der Sie nur eine einzige Zeile eingeben dürfen. Sobald Sie mehr als eine Zeile verwenden wollen, müssen Sie die Anweisungen in geschweifte Klammern einfassen. Dies können Sie der Konsistenz halber allerdings auch bereits bei Verwendung einer einzigen Anweisung tun:

```
if (Zahl == 0) {
    Console.WriteLine("Die Zahl lautet Null.");
}
```

Die **If...Then**-Bedingung gerät unter C# auf jeden Fall zur **If**-Bedingung, denn das Schlüsselwort **Then** wird ja hier gar nicht benötigt.

If...Then...Else-Bedingung

Nehmen wir noch einen **Else**-Zweig hinzu. Unter VBA verwenden Sie folgende Struktur:

```
If intZahl = 0 Then
    MsgBox "Die Zahl lautet Null."
Else
    MsgBox "Die Zahl lautet nicht Null."
End If
```

Unter C# könnten Sie hier wieder die Syntax für die Einzeiler verwenden:

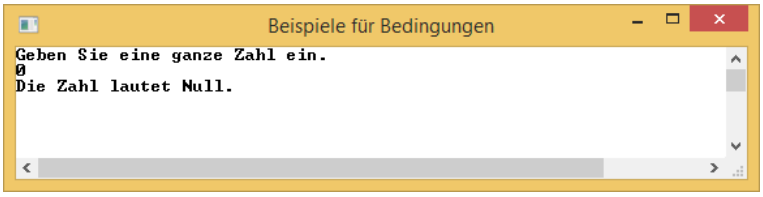


Bild 1: Ausgabe eines Ergebnisses per **If**-Bedingung

```
if (Zahl == 0)
    Console.WriteLine("Die Zahl lautet Null.");
else
    Console.WriteLine("Die Zahl lautet nicht Null.");
```

Mit geschweiften Klammern erhalten Sie folgenden Code. Hier können Sie nun auch noch weitere Anweisungen je Zeile unterbringen:

```
if (Zahl == 0) {
    Console.WriteLine("Die Zahl lautet Null.");
}
else {
    Console.WriteLine("Die Zahl lautet nicht Null.");
}
```

Verschachteltes If...Then...Else

Wenn Sie zwei **If...Then**-Bedingungen unter VBA verschachteln wollten, haben Sie dies einfach nach folgendem Schema erledigt:

```
If <Äußere Bedingung> Then
    If <Innere Bedingung> Then
        MsgBox "Äußere wahr, innere wahr"
    Else
        MsgBox "Äußere wahr, innere falsch"
    End If
End If
```

Wenn Sie unter C# ohne geschweifte Klammern arbeiten, lässt sich die Zuordnung des **else**-Teils einer verschachtelten **if...else**-Konstruktion möglicherweise nicht sehr gut erkennen. Einfacher ist es immer, mit geschweiften Klammern zu arbeiten – zumindest aus Gründen der Übersicht. Dies sieht dann etwa wie folgt aus:

```
if (Zahl != 0) {
    if (Zahl > 5) {
        Console.WriteLine("Die Zahl ist größer als 5.");
    }
    else {
        Console.WriteLine("Die Zahl ist <= 5.");
    }
}
```

In diesem Fall prüft die äußere **if**-Bedingung, ob der Benutzer eine Zahl größer **0** eingegeben hat. Die innere prüft, ob die Zahl größer als **5** ist und gibt eine entsprechende Meldung aus. Beim Wert **0** oder negativen Zahlen geschieht nichts.

If...ElseIf

Es fehlt noch die von VBA bekannte Möglichkeit, mehr als nur den **If**- und den **Else**-Teil auszuwerten – nämlich der **ElseIf**-Zweig (oder auch mehrere):

```
If <Erste Bedingung> Then
    MsgBox "Erste Bedingung"
ElseIf <Zweite Bedingung> Then
    MsgBox "Zweite Bedingung"
Else
    MsgBox "Andere Bedingung"
End If
```

Ein **ElseIf** gibt es unter C# nicht. Hier müssen Sie sich mit immer weiter verschachtelten **else**/**if**-Bedingungen behelfen.

Sprich: Wenn die erste **if**-Bedingung nicht erfüllt wird (**Zahl == 0**), wird der **else**-Teil angelaufen, der eine weitere **if**-Bedingung enthält (**Zahl == 1**) – und einen weiteren **else**-Teil:

```
if (Zahl == 0) {
    Console.WriteLine("Die Zahl lautet 0.");
}
else {
    if (Zahl == 1) {
        Console.WriteLine("Die Zahl lautet 1.");
    }
    else {
        if (Zahl == 2) {
```

Von VBA zu C#: Schleifen

Schleifen spielen bei der Programmierung eine wichtige Rolle. Unter VBA gibt es die For...Next- und die For...Each-Schleife sowie die verschiedenen Varianten der Do...-Schleife. In diesem Artikel schauen wir uns an, wie Sie diese Schleifentypen unter C# abbilden und welche Besonderheiten dabei auftreten.

Beispiele

Wenn Sie die Beispiele dieses Artikels ausprobieren möchten, erstellen Sie eine C#-Konsolenanwendung und fügen die Anweisungen in die Prozedur **static void main()** ein. Betätigen Sie dann die Taste **F5**, um die Methode zu starten.

For...Next-Schleife

Die **For...Next**-Schleife aus VBA sieht so aus:

```
For i = 1 To 10
    Debug.Print "Aktuelle Zahl: " & i
Next i
```

Unter C# sieht dies etwas anders aus, lässt sich aber identisch abbilden:

```
for (int i = 1; i <= 10; i++)
{
    Console.WriteLine("Aktuelle Zahl: {0}", i);
}
```

Der wichtigste Teil ist hier die erste Zeile – die innerhalb der Schleife ausgeführten Anweisungen packen Sie, wie von C# gewohnt, einfach in geschweifte Klammern. Die erste Zeile enthält das Schlüsselwort **for** sowie in Klammern drei Parameter:

- Der erste deklariert die Laufvariable, hier **i**, und den Startwert der Variablen.
- Der zweite legt die Abbruchbedingung fest. In diesem Fall soll die Schleife solange laufen, wie **i** einen Wert kleiner oder gleich **10** aufweist. Die Abbruchbedingung muss immer einen Boolean-Wert als Ergebnis liefern.
- Schließlich folgt der dritte Parameter, der die Änderung der Laufvariablen festlegt. In diesem Fall sorgt **i++** dafür, dass **i** mit jedem Durchlauf um den Wert **1** erhöht wird.

Das Ergebnis dieser Schleife sieht so aus:

```
Aktuelle Zahl: 1
Aktuelle Zahl: 2
Aktuelle Zahl: 3
...
Aktuelle Zahl: 10
```

Deklaration der Laufvariablen

Nun haben wir die Laufvariable direkt im ersten Parameter der Schleife deklariert und instanziert:

```
int i = 1
```

Dadurch legen wir gleichzeitig den Gültigkeitsbereich der Variablen auf die Schleife selbst fest. Nach dem Beenden der Schleife können Sie nicht mehr auf die Variable zugreifen.

Das ist aber kein Problem, denn für Fälle, in denen Sie den Inhalt der Variablen später noch benötigen, deklarieren Sie die Laufvariable einfach vor Beginn der Schleife. Dies sieht dann wie folgt aus:

```
int i;
for (i = 1; i <= 10; i++)
{
    Console.WriteLine("Aktuelle Zahl: {0}", i);
}
Console.WriteLine("Wert von i: {0}", i);
```

Die abschließende Anweisung gibt nun den Wert der Zählervariablen nach dem letzten Durchlaufen der Schleife aus:

```
Aktuelle Zahl: 1
Aktuelle Zahl: 2
Aktuelle Zahl: 3
...
Aktuelle Zahl: 10
Wert von i: 11
```

Hätten Sie im ersten Beispiel auf die gleiche Art auf die Laufvariable zugegriffen, hätte dies einen Syntaxfehler hervorgerufen.

Ob Sie die Zählervariablen nun jeweils vor dem Start der Schleife oder erst in der **for...**-Anweisung deklarieren, hängt von der geplanten Nutzung dieser Variable ab – soll diese vor dem Start der Schleife oder im Anschluss gefüllt oder ausgelesen werden, müssen Sie die Deklaration natürlich vor dem ersten Zugriff programmieren. Anderenfalls haben Sie die freie Wahl.

Herunterzählen

Wenn Sie der Wert der Zählervariablen beim Initialisieren der Schleife größer sein soll als beim Beenden, legen Sie die Werte im ersten und zweiten Parameter der **for...**-Anweisung entsprechend fest. Für den dritten Parameter definieren Sie dann die Schrittweiter für das Herunterzählen. Angenommen, Sie wollen jeweils den Wert **1** abziehen, verwenden Sie an dieser Stelle den Ausdruck **i--**:

```
int i;
for (i = 10; i >= 1; i--)
{
    Console.WriteLine("Aktuelle Zahl: {0}", i);
}
```

Schrittweite festlegen

Was geschieht nun, wenn Sie der Zählervariablen nicht mit jedem Durchlauf genau den Wert **1** hinzufügen möchten, sondern beispielsweise den Wert **2**? Die Antwort ist einfach: Sie können jeden beliebigen Ausdruck für den dritten Parameter verwenden, welcher der Laufvariablen einen Wert zuweist. Im folgenden Beispiel sollen alle Werte von **0** bis **10** mit der Schrittweite **2** durchlaufen werden. Also verwenden wir für den dritten Parameter den Ausdruck **i +=2**:

```
int i;
for (i = 0; i <= 10; i += 2)
{
    Console.WriteLine("Aktuelle Zahl: {0}", i);
}
```

Sie könnten dort auch nach guter, alter VBA-Manier **i = i + 2** einsetzen:

```
for (i = 0; i <= 10; i = i + 2)
```

Wichtig ist allein, dass Sie **i** dort einen Wert zuweisen.

Verschachtelte for-Schleifen

Genau wie die **For...Next**-Schleife unter VBA können Sie auch die **for**-Schleife unter C# verschachteln.

Das folgende Beispiel soll beispielsweise eine äußere Schleife mit der Laufvariablen **x** und eine innere Schleife mit der Laufvariablen **y** durchlaufen. Dabei soll die Anweisungen der inneren Schleife das Produkt der aktuellen Werte von **x** und **y** hintereinander ausgeben, wobei jeweils fünf Zeichen Platz für die Zahlenwerte vorgesehen werden sollen.

Beim ersten Durchlauf der äußeren Schleife geschieht zunächst nichts, dann beginnt der erste Durchlauf der inneren Schleife mit der Ausgabe des Produkts von **1** und **1**. In der Folge wird die innere Schleife solange durchlaufen, bis die Abbruchbedingung erfüllt ist. Danach folgt die letzte Anweisung der äußeren Schleife, die dafür sorgt, dass die zehn in der aktuellen Zeile der Konsole eingetragenen Werte mit einem Zeilenumbruch abgeschlossen werden:

Von VBA Zu C#: Arrays

Arrays gibt es wohl in jeder Programmiersprache, jeweils mit eigenen Besonderheiten. Wie füllt man ein Arrays? Wie greift man auf die Werte eines Arrays zu? Wie organisiert man mehrdimensionale Arrays? Wie durchläuft man die Werte eines Arrays oder ermittelt die Anzahl der enthaltenen Elemente? All diese Fragen beantwortet der vorliegende Artikel.

Unter VBA konnten Arrays nur einfache Werttypen aufnehmen, also beispielsweise Zahlen oder Zeichenketten. Wenn Sie etwa Objekte in einer Liste speichern wollten, mussten Sie auf eine Alternative wie das **Collection**- oder das **Dictionary**-Objekt zugreifen.

Unter C# können Sie dies gleichermaßen mit einem Array erledigen. Dies ist jedoch nicht der einzige Unterschied. In den folgenden Abschnitten erhalten Sie die Grundlagen für die Programmierung von Arrays mit C#.

Deklaration von Arrays

Unter VBA haben Sie ein Array durch das Hinzufügen eines Klammernpaares bei der Deklaration kenntlich gemacht:

```
Dim Zahlwoerter() As String
```

Wenn Sie schon wussten, wie viele Elemente das Array enthalten sollte, haben Sie diese Anzahl in Klammern angegeben:

```
Dim Zahlwoerter(9) As String
```

Dies bedeutete ohne weitere Angaben, dass das Array zehn Elemente mit den Indizes **0** bis **9** aufnehmen konnte.

Unter C# verwenden Sie keine runden, sondern eckige Klammern, und diese geben Sie auch nicht hinter dem Variablennamen, sondern hinter dem Datentyp an:

```
string[] Zahlwoerter;
```

Nun ist natürlich noch nicht bekannt, wie viele Elemente das Array aufnehmen soll. Dies geben Sie auch nicht bekannt, indem Sie die Anzahl einfach in die eckigen Klammern schreiben. Vielmehr ist eine Initialisierung unter Angabe der Anzahl der Elemente nötig. Diese können Sie in einer neuen Zeile erledigen:

```
Zahlwoerter = new string[10];
```

Oder Sie verwenden einfach einen Einzeiler zur gleichzeitigen Deklaration und Initialisierung:

```
string[] Zahlwoerter = new string[10];
```

Die Verwendung des Schlüsselworts **new** stellt hier im Gegensatz zum Vorgehen unter VBA heraus, dass es sich bei dem Array um ein Objekt handelt.

Stellt sich gleich die erste Frage: Besitzt das Array nun zehn Elemente? Und die zweite gleich hinterher: Wie werden diese indiziert – beginnend mit **0** oder **1**?

Das Array enthält nun tatsächlich genau die angegebene Elementanzahl. Im Gegensatz zu der etwas unübersichtlichen Vorgehensweise von VBA, bei der die Anzahl von der Basis des Index abhängt, initialisiert C# genau mit der angegebenen Menge von Elementen.

Der Index basiert unter C# auf dem Wert **0**. Auf ein Element greifen Sie über die Angabe des Indexwertes in eckigen Klammern zu, also etwa über **Zahlwoerter[0]** – mehr dazu weiter unten.

Array füllen

Schauen wir uns erstmal eine einfache Methode an, das Array zu füllen – und zwar direkt bei der Initialisierung.

Ausgehend vom obigen Einzeiler gelingt dies so (in einer Zeile):

```
string[] Zahlwoerter = new String[10] { "Eins", "Zwei",
"Drei", "Vier", "Fünf", "Sechs", "Sieben", "Acht", "Neun",
"Zehn" };
```

Sie geben also einfach die gewünschten Werte in einer geschweiften Klammer und durch Kommata getrennt an.

In diesem Fall soll das Array zehn Elemente aufnehmen und jedes der Elemente wird auch gleich gefüllt.

Es gibt noch eine Alternative, bei der Sie die Anzahl der Elemente nicht angeben und diese direkt aus der Anzahl der übergebenen Elemente ermittelt wird. In diesem Fall lassen Sie einfach die Angabe der Anzahl innerhalb der eckigen Klammern weg (in einer Zeile):

```
string[] Zahlwoerter = new String[] { "Eins", "Zwei",
"Drei", "Vier", "Fünf", "Sechs", "Sieben", "Acht", "Neun",
"Zehn" };
```

Anschließend geben wir die im Array gespeicherten Werte in der Konsole aus. Für den Zugriff auf den Inhalt eines der Elemente geben Sie den Variablennamen gefolgt vom gewünschten Index in eckigen Klammern an.

Am einfachsten bekommen wir den Inhalt eines Elements so auf die Konsole:

```
Console.WriteLine(Zahlen[0]);
```

Wir wollen aber gleich alle zehn, und dann auch noch mit einem kleinen Hinweistext:

```
Console.WriteLine("Wert von Zahlwoerter[0]: {0}", Zahlen[0]);
Console.WriteLine("Wert von Zahlwoerter[1]: {0}", Zahlen[1]);
Console.WriteLine("Wert von Zahlwoerter[2]: {0}", Zahlen[2]);
...
Console.WriteLine("Wert von Zahlwoerter[9]: {0}", Zahlen[9]);
```

Das Ergebnis sieht nun wie folgt aus:

```
Wert von Zahlwoerter[0]: Eins
Wert von Zahlwoerter[1]: Zwei
Wert von Zahlwoerter[2]: Drei
...
Wert von Zahlwoerter[9]: Zehn
```

Wichtig: Wenn Sie die Anzahl der zu erstellenden Elemente angeben und gleich dahinter die Elemente hinzufügen, muss deren Anzahl übereinstimmen.

Die schnellste Alternative, um ein Array direkt mit Werten zu füllen, lässt noch ein paar Schlüsselwörter weg und übergibt die Werte direkt als Array an die Variable:

```
string[] Zahlwoerter = { "Eins", "Zwei", "Drei", "Vier",
"Fünf", "Sechs", "Sieben", "Acht", "Neun", "Zehn" };
```

Elemente nach Initialisierung füllen

Wenn Sie das Array zunächst deklarieren und initialisieren, aber noch nicht füllen, holen Sie dies in späteren Anweisungen nach.

Dazu weisen Sie einfach dem entsprechenden Element, das Sie über die Angabe des Indexwertes in eckigen Klammern referenzieren, den gewünschten Wert zu:

```
string[] Zahlwoerter = new string[10];
Zahlwoerter[0] = "Eins";
Zahlwoerter[1] = "Zwei";
Zahlwoerter[2] = "Drei";
```

Wenn Sie nun alle Elemente ausgeben, bleiben die hinteren sieben leer. Die Elemente werden bei der Initialisierung gleich mit leeren Zeichenketten gefüllt.

0 als initialer Wert von Zahlen-Arrays

Probieren wir dies gleich noch mit Zahlen aus, um zu prüfen, mit welchem Wert das Element eines Zahlen-Arrays initial gefüllt wird:

```
int[] Zahlen = new int[5];
Zahlen[0] = 1;
Zahlen[1] = 2;
Zahlen[2] = 3;
Console.WriteLine("Wert von Zahlen[0]: {0}", Zahlen[0]);
Console.WriteLine("Wert von Zahlen[1]: {0}", Zahlen[1]);
Console.WriteLine("Wert von Zahlen[2]: {0}", Zahlen[2]);
Console.WriteLine("Wert von Zahlen[3]: {0}", Zahlen[3]);
Console.WriteLine("Wert von Zahlen[4]: {0}", Zahlen[4]);
```

Wir definieren also ein Array mit fünf **int**-Elementen und füllen dann nur drei davon. Das Ergebnis bei der Ausgabe aller Elemente inklusive der übrigen zwei sieht so aus:

```
Wert von Zahlen[0]: 1
Wert von Zahlen[1]: 2
Wert von Zahlen[2]: 3
Wert von Zahlen[3]: 0
Wert von Zahlen[4]: 0
```

Leere Zahlenelemente werden also direkt mit dem Wert **0** gefüllt.

Größe einer Dimension eines Array ermitteln

Unter VBA ermitteln Sie die Größe eines Arrays über die Differenz der Funktionen **UBound** und **LBound** mit dem Array als Parameter:

```
Debug.Print UBound(intZahlen()) - LBound(intZahlen())
```

Für mehrdimensionale Arrays gibt es noch weitere Besonderheiten. Unter C# gibt es verschiedene Eigenschaften, mit denen Sie die Anzahl der Elemente einer Dimension sowie die Gesamtzahl der Elemente ermitteln können. Zunächst reicht uns die Anzahl der Elemente einer einzigen Dimension. Die erhalten Sie mit der Funktion **GetLength**:

```
Console.WriteLine(Zahlen.GetLength(0));
```

GetLength erwartet als Parameter den Index der zu untersuchenden Dimension, in diesem Fall **0**.

Im Falle eines eindimensionalen Arrays liefert die Funktion **Length** den identischen Wert. Für mehrdimensionale Arrays liefert sie hingegen die Anzahl aller Elemente. Dementsprechend kommt sie ohne Parameter aus:

```
Console.WriteLine(Zahlen.Length);
```

Größe des Arrays zur Laufzeit ändern

Weiter oben haben Sie bereits gesehen, dass Sie die Größe eines Arrays zur Laufzeit festlegen können. Dies ist gleichbedeutend mit der Initialisierung. Den dort angegebenen Zahlenwert, der die Anzahl der Elemente angibt, kann auch als Variable übergeben werden – nicht immer ist ja gleich beim Programmieren bekannt, wie groß ein Array gegebenenfalls werden soll.

Im folgenden Beispiel legen wir die Anzahl der Elemente per Variable fest (die natürlich auch etwa über die Konsole abgefragt oder auf andere Weise ermittelt werden kann) und initialisieren das Array damit:

```
string[] Zahlwoerter;
int AnzahlZahlwoerter;
AnzahlZahlwoerter = 10;
Zahlwoerter = new string[AnzahlZahlwoerter];
```

Aber was geschieht, wenn wir das Array bereits mit Werten füllen oder auslesen wollen, aber später die Größe des Arrays ändern müssen? Mit der Initialisierung per **new**-Schlüsselwort ist uns nicht geholfen, da das Array dann ja wieder leer ist.

Genau genommen gibt es bei den Arrays keine Möglichkeit, es ohne Umweg zu vergrößern oder zu verkleinern. Immerhin gibt es einen Workaround. Dabei verwenden wir neben dem eigentlichen Array **Zahlen** ein Hilfsarray namens **temp**. **Zahlen** deklarieren wir als Array mit der Elementanzahl **0**, **temp** zunächst ohne Initialisierung.

Dann durchlaufen wir eine Schleife von **1** bis **10**, da wir dem Array mit jedem Durchlauf ein weiteres Element

Die Console-Klasse

Im Gegensatz zu VBA können Sie mit den .NET-Programmiersprachen wesentlich komfortabler auf die Eingabeaufforderung zugreifen – und zwar über die Console-Klasse des System-Namespaces. Dieses bietet nicht nur die Möglichkeit, Texte auszugeben, sondern auch solche zum Einlesen von Benutzereingaben. Außerdem können Sie die Eingabeaufforderung damit nach Ihren eigenen Wünschen formatieren. Dieser Artikel beschreibt die Verwendung der Console-Klasse.

Lesen und Schreiben

In einigen Beispielen anderer Artikel verwenden wir die Konsole, um ermittelte Daten auszugeben oder um mit dem Benutzer zu interagieren. Dafür benötigen wir vor allem die **Read...**- und **Write...**-Methoden der **Console**-Klasse. Wenn Sie die folgenden Beispiele ausprobieren möchten, legen Sie am einfachsten eine neue Konsolenanwendung für C# an. Dort fügen Sie die folgenden Beispielanweisungen einfach in die Prozedur **static void Main(string{] args)** ein und drücken auf **F5**, um diese auszuprobieren. Die **Write**-Methode gibt einfach den als Parameter angegebenen Text in der Konsole aus:

```
Console.Write("1");
```

Dummerweise wird die Konsole unmittelbar darauf wieder geschlossen, sodass diese bestenfalls kurz aufflackert.

Damit dies nicht geschieht, müssen Sie zumindest die Betätigung einer einzigen Taste einfordern, was der nachfolgende Aufruf der **ReadKey**-Methode bewirkt:

```
Console.Write("1");
Console.ReadKey();
```

Wenn Sie mehrere Ausgaben in einer Zeile tätigen wollen, beispielsweise in einer Schleife, rufen Sie einfach mehrmals die **Write**-Methode auf:

```
Console.Write("1");
Console.Write("2");
Console.ReadKey();
```

Das Ergebnis sieht dann wie in Bild 1 aus.

Auf Ihrem Rechner dürfte diese Meldung noch mit schwarzem Hintergrund und weißer Schrift erscheinen. Wir haben dies geändert, damit die Seiten dieses Artikels nicht so schwarz erscheinen – natürlich mit den Methoden und Eigenschaften des **Console**-Objekts. Dazu nutzen wir die Eigenschaften **BackgroundColor** und **ForegroundColor**. Beide sorgen dafür, dass die Hintergrund- und die Schriftfarbe für die geschriebenen Zeichen angepasst werden. Das bedeutet, dass der Rest des Fensters leider schwarz bleibt. Allerdings können wir dies ändern, indem wir nach dem Einstellen der beiden Eigenschaften noch die **Clear**-Methode des **Console**-Objekts auslösen:

```
Console.Title = "Konsole";
Console.BackgroundColor = ConsoleColor.White;
Console.ForegroundColor = ConsoleColor.Black;
```

Komplette Zeilen schreiben

Mit der **Write**-Methode schreiben Sie den angegebenen Text einfach in die Konsole und können später weitere Zei-

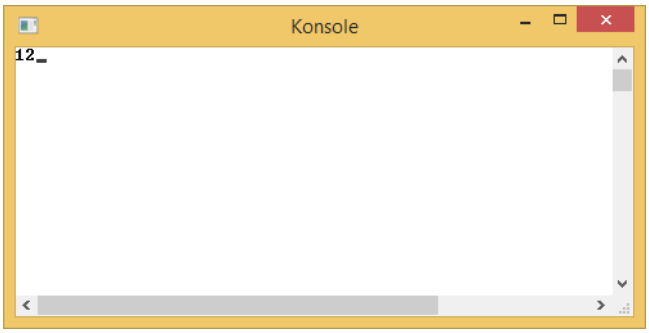


Bild 1: Ausgabe zweier **Write**-Anweisungen

chen an diesen Text anhängen. Wenn Sie nun direkt eine komplette Zeile inklusive Zeilenumbruch schreiben wollen, wie Sie es etwa von der **Debug.Print**-Methode unter VBA zum Schreiben von Zeilen in das Direktfenster kennen, verwenden Sie die Methode **WriteLine**:

```
Console.WriteLine("Eine Zeile.");
Console.ReadKey();
```

Als Ergebnis rutscht die Einfügemarke gleich in die folgende Zeile, weitere per **Write...** geschriebene Texte landen ebenfalls dort (siehe Bild 2).

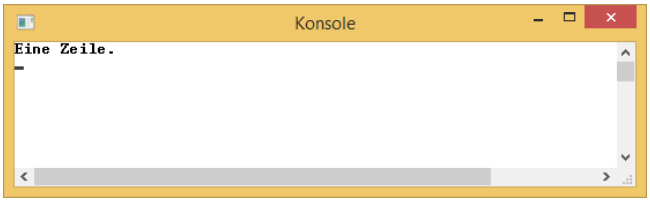


Bild 2: Ausgabe einer kompletten Zeile mit **WriteLine**

Die **WriteLine**-Methode kommt in mehreren Varianten. Bei einer weiteren geben Sie mehrere Parameter an. Der erste enthält die eigentlich auszugebende Zeichenkette, die allerdings Platzhalter im Format **{0}**, **{1}** und so weiter enthalten kann. Die dafür einzusetzenden Werte geben Sie dann mit den nachfolgenden Parametern an, also beispielsweise so:

```
string Platzhalter = "Platzhalter";
Console.WriteLine("Text mit einem {0}.", Platzhalter);
Console.ReadKey();
```

Dies gibt die folgende Zeile aus:

```
Text mit einem Platzhalter.
```

Die Platzhalter können Sie auch noch durch einen zweiten Parameter erweitern, also etwa **{0, 5}**. Dies würde den für den ersten Platzhalter angegebenen Ausdruck mit einer Breite von mindestens fünf Zeichen rechtsbündig ausgeben:

```
string Platzhalter = "x.";
Console.WriteLine("Text mit einem {0, 5}", Platzhalter);
```

würde also folgendes ergeben:

```
Text mit einem     x.
```

Ist der Text für den Platzhalter länger als die reservierte Anzahl Zeichen, wird der vollständige Platzhaltertext ausgegeben. Für die linksbündige Ausgabe geben Sie einen negativen Wert als zweiten Parameter im Platzhalter an:

```
Console.WriteLine("Text mit einem {0,-5}.", Platzhalter);
```

liefert dann:

```
Text mit einem x    .
```

Es gibt noch eine Reihe weiterer Möglichkeiten, die Ausgabe zu formatieren. Dies würde jedoch den Rahmen dieses Artikels sprengen, daher gehen wir später in einem weiteren Artikel darauf ein.

Eingaben lesen

Es gibt drei Methoden, mit denen Benutzereingaben gelesen werden:

- **ReadKey**: Liefert ein Objekt des Typs **ConsoleKeyInfo** zurück, dass beispielsweise das gedrückte Zeichen und eventuell dabei betätigte Tasten wie **Strg**, **Alt** oder **Umschalt** liefert.
- **ReadLine**: Liest den Inhalt einer Zeile bis zum Betätigen eines Zeilenumbruchs ein und liefert diesen zurück.
- **Read**: Liefert den ASCII-Code des gedrückten Zeichens zurück.

Die ReadKey-Methode

Diese Methode wartet auf die Eingabe eines Zeichens und hält solange die aufrufende Routine an, bis der Benutzer

Von Access zu WPF: Fenster

Unter Access war alles so einfach und gewohnt. Das Access-Fenster bildete den Rahmen, zeigt das Ribbon an oder stellte die Formulare und Berichte in Anwendungen dar. Für das Öffnen der Objekte brauchte man nur die DoCmd.Open...-Anweisung zu kennen. Unter .NET sieht das ganz anders aus: Das Anwendungsfenster ist bereits ein Formular, es gibt andere Steuerelemente – und das Öffnen von Formularen erfolgt auch ganz anders.

Wenn Sie eine WPF-Anwendung erstellen (in diesem Beispiel für Programmiersprache C#), wählen Sie nach dem Starten von Visual Studio (hier in der Community-Edition der Version 2013) den Menüeintrag **Datei|Neu|Projekt...** aus. Danach erscheint der Dialog **Neues Projekt**, wo Sie unter **Vorlagen|Visual C#|Windows-Desktop** die Vorlage **WPF-Anwendung** selektieren (siehe Bild 1). Es gibt noch einige weitere Anwendungstypen auf Basis von WPF, aber wir konzentrieren uns hier zunächst auf die Desktop-Anwendung.

Visual Studio bietet hier direkt ein Verzeichnis an, in dem das Projekt standardmäßig angelegt wird. Diesen können Sie natürlich ändern. Wichtig ist zu wissen, dass im angegebenen Verzeichnis ein Unterverzeichnis mit dem Namen des Projekts angelegt wird, das schließlich die eigentlichen Projektdateien enthält. In diesem Fall soll das Projekt **VonAccessZuWPF** heißen.

Das Hauptfenster

Im Vergleich zu Access, wo das Access-Fenster das Hauptfenster der Anwendung ist (wenn Sie es nicht mit Tricks ausgeblendet haben), definieren Sie das Hauptfenster Ihrer WPF-Anwendung komplett neu. Dazu bietet Visual Studio im neuen Projekt standardmäßig ein Fenster namens **MainWindow** hinzu. Wir wollen uns an dieser Stelle daraufhin einigen, statt wie in Access von Formularen künftig von Fenstern zu sprechen.

Das Formular wird mithilfe von zwei Dateien beschrieben:

- **MainWindow.xaml**: Enthält die Beschreibung des Aussehens des Fensters.
- **MainWindow.xaml.cs**: Enthält den Code des Fensters. Unter Visual Basic würde die Endung **vb** lauten.

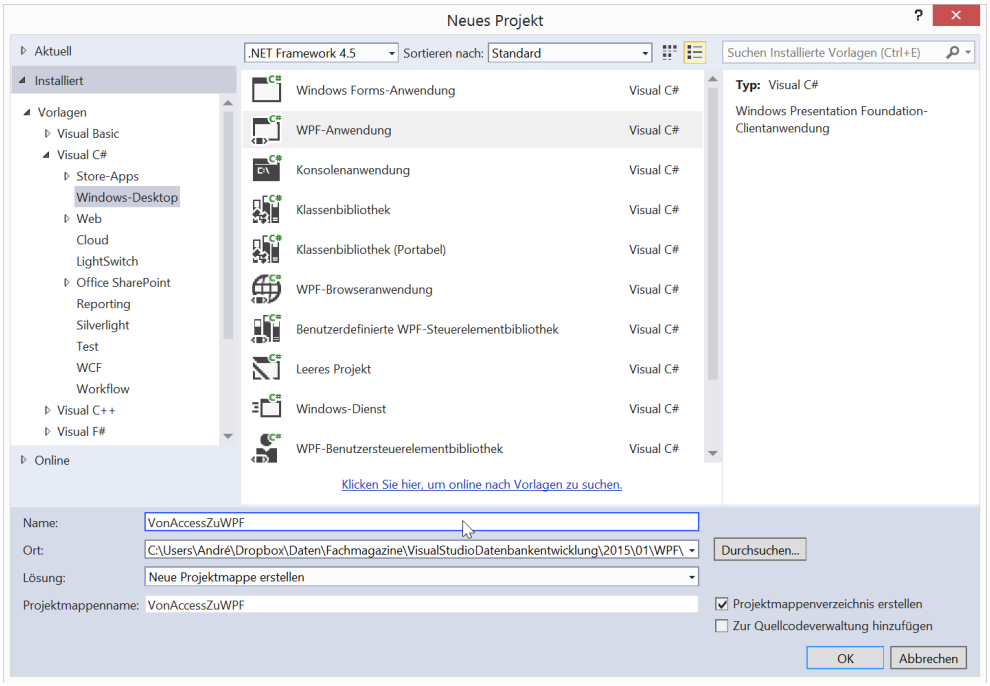


Bild 1: Neues WPF-Projekt für Visual C# erstellen

Die **.xaml**-Datei stellt Visual Studio in einem Bereich dar, der oben den Entwurf des Fensters anzeigt und unten den XML-Code, der das Aussehen definiert. Sie können das Aussehen auf folgenden Wegen beeinflussen:

- durch Ändern der Größe, Position oder anderer Eigenschaften des Fensters und der enthaltenen Steuerelemente direkt im Entwurf,
- durch Ändern der Eigenschaften im Bereich **Eigenschaften** oder
- durch Anpassen des XML-Codes zur Definition des Aussehens des Fensters.

Jegliche Änderungen an diesen drei Stellen wirken sich direkt auf die jeweils anderen Wege aus. Wenn Sie also im XML-Code die Breite des **Window**-Elements mit dem Attribut **Width** auf **600** einstellen, wird das Fenster direkt in dieser Breite dargestellt und auch der entsprechende Eintrag im Eigenschaftsfenster wird geändert.

Beispielprojekt: Den Code für die Beispiele der folgenden Abschnitte finden Sie in einem neuen, jungfräulichen WPF-Projekt.

Code des Hauptfensters

Die Datei **MainWindow.xaml** enthält den folgenden Code:

```
<Window x:Class="VonAccessZuWPF.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/
            xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="600">
    <Grid>
    </Grid>
</Window>
```

Hier finden Sie die beim Start vorhandenen Einstellungen: Der Titel lautet **MainWindow**, außerdem sind Höhe und Breite auf **350** und **600** festgelegt. Eine weitere wichtige Information finden Sie gleich in der ersten Zeile: Unter **x:Class** gibt das Dokument die C#-Datei an, die den Code für dieses Fenster enthält – in diesem Fall **VonAccessZuWPF.MainWindow**. Alle Elemente, die Sie nun zum Fenster hinzufügen, landen als Beschreibung innerhalb des **Grid**-Elements (**<Grid>...</Grid>**).

XAML und C#

Damit wird nun offensichtlich: Es gibt zwei verschiedene Dateien zur Programmierung eines Fensters unter WPF – die XML-Datei mit der XAML-Definition der Benutzeroberfläche und eine C#-Datei mit den Funktionen des Fensters.

Im Vergleich mit einem Access-Formular entspricht die Entwurfsansicht des WPF-Fensters dem Formularentwurf und die C#-Datei mit dem Code dem Klassenmodul des Formulars.

Unter Access werden intern allerdings sowohl die Beschreibung des Formulars sowie der Steuerelemente als auch der VBA-Code in einem einzigen Dokument untergebracht, was auch deutlich wird, wenn Sie dieses vom VBA-Fenster etwa mit der **SaveAsText**-Anweisung in eine Textdatei exportieren und in einem Texteditor ansehen.

Das »Klassenmodul« mit dem C#-Code hinter dem Fenster **MainWindow** hat den folgenden Inhalt:

```
namespace VonAccessZuWPF
{
    /// <summary>
    /// Interaktionslogik für MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

Hier geschieht noch nichts Spektakuläres, da wir noch keine Ereignisprozeduren et cetera angelegt haben. Die einzige Methode namens **InitializeComponent** erstellt kurz gefasst das Fenster auf Basis der XAML-Datei.

Wenn Sie genau wissen wollen, welcher Code sich hinter einer Anweisung befindet, klicken Sie ähnlich wie im VBA-Editor mit der rechten Maustaste und wählen den Eintrag **Definition einsehen** aus. Es erscheint dann ein gelb hinterlegter Bereich mit der Routine, die durch die Anweisung ausgelöst wird (siehe Bild 2).

Trennung von Oberfläche und Code

Die Definition der Benutzeroberfläche und der Code sind nicht nur dateiweise getrennt, sondern sie können auch komplett unabhängig voneinander bearbeitet werden. Während es unter Access kaum möglich ist, dass ein Entwickler Änderungen an der Benutzeroberfläche vornimmt, während ein anderer den Programmcode anpasst, gibt es für WPF mit **Blend für Visual Studio** sogar ein eigenes Werkzeug für die Verfeinerung der Benutzeroberfläche.

Vorteil Visual Studio: Zoom

Wenn Sie Ihr erstes Fenster erstellen, können Sie direkt ein Feature ausprobieren, das dem Access-Entwickler wohl für immer vorbehalten bleibt: Sie können die Ansicht des Entwurfs vergrößern oder verkleinern.

Dazu sollten Sie sich gleich die folgenden Tastenkombinationen merken:

- **Strg + Alt + +**: Vergrößern
- **Strg + Alt + -**: Verkleinern

Steuerelemente hinzufügen

Standardmäßig im linken Bereich von Visual Studio finden Sie den Werkzeugkasten. Dieser bietet verschiedene Sammlungen von Steuerelementen. Die obere liefert mit **Häufig verwendete WPF-Steuerelemente** die gängigen Steuerelemente, die Sie auch von Access her kennen – aber auch einige neue Kandidaten. Unter **Alle WPF-Steuerelemente** finden Sie die komplette Sammlung. Fügen Sie hier wie in Bild 3 eine Schaltfläche zum WPF-Fenster hinzu. Sie erkennen direkt, dass nicht nur die Schaltfläche im Fenster erscheint, sondern auch, dass der Designer ein Element zur XAML-Datei des Fensters hinzugefügt hat, und zwar in das **Grid**-Element:

```
<Button Content="Button" HorizontalAlignment="Left"
Margin="36,34,0,0" VerticalAlignment="Top" Width="75"/>
```

Dazu erhält das Steuerelement direkt einige Werte für Standardeigenschaften wie die horizontale und vertikale Ausrichtung (**HorizontalAlignment** und **VerticalAlignment**), die Ränder (**Margin**) und die Breite (**Width**). Die Beschriftung landet im Attribut **Content**. Klicken Sie nun auf **F5**, starten Sie die WPF-Anwendung und das Fenster

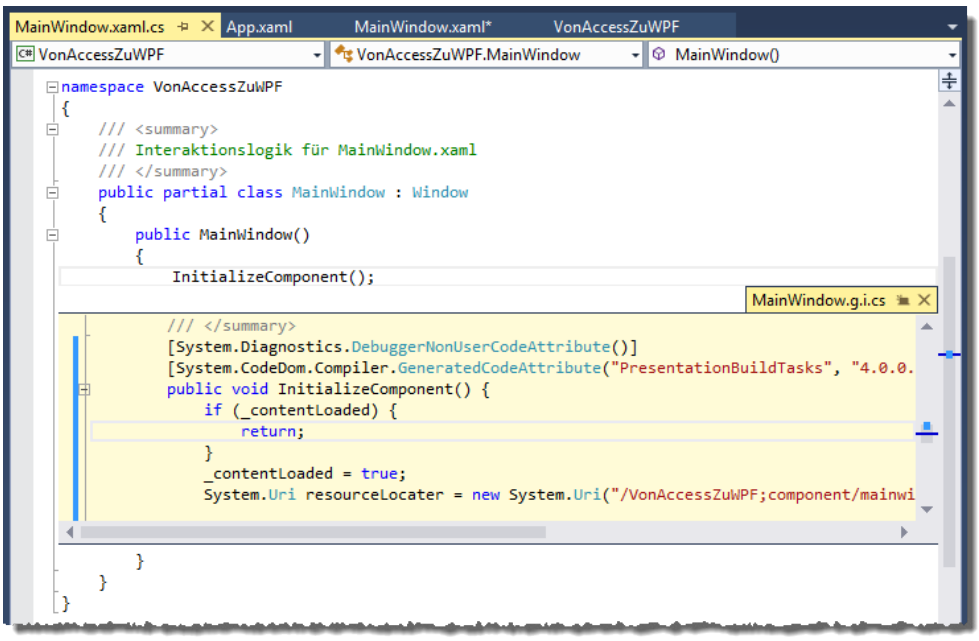


Bild 2: Der Code hinter einer Anweisung wird farbig hinterlegt eingeblendet.

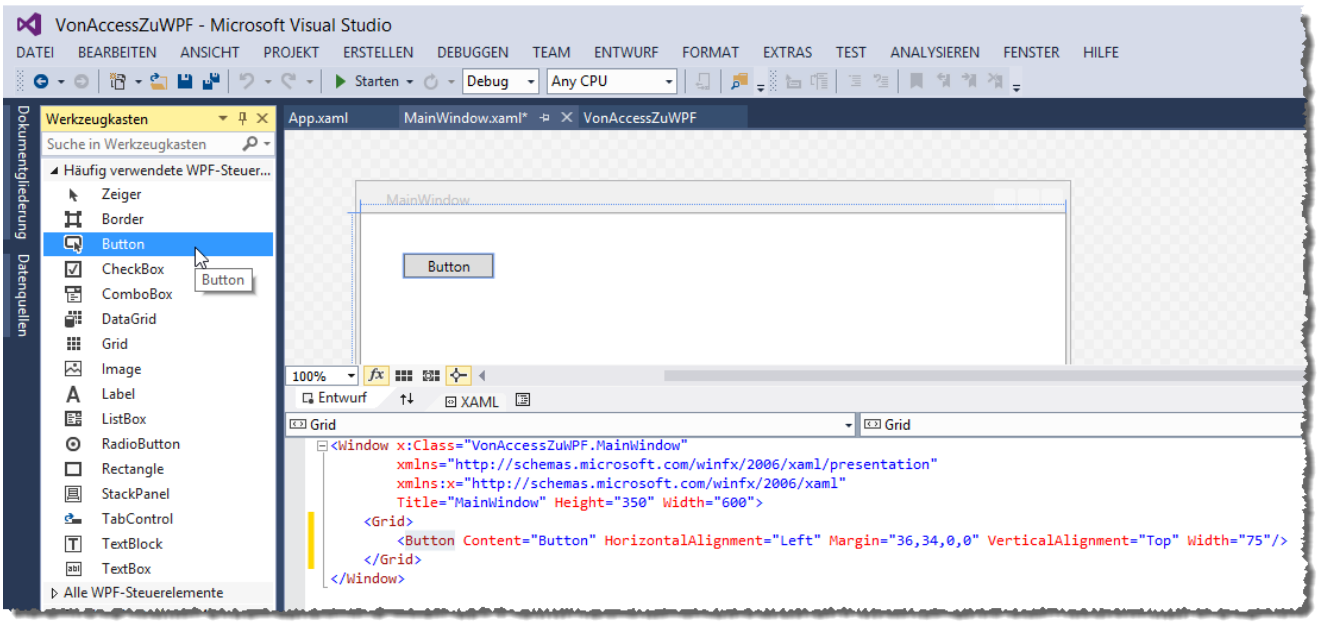


Bild 3: Neues Steuerelement im Entwurf und im XAML-Code

mit der Schaltfläche erscheint – diese ist allerdings noch ohne Funktion (siehe Bild 4).

Startfenster ändern

In einer Access-Anwendung ist standardmäßig kein Formular als Startformular der Anwendung voreingestellt. Das ist kein Problem: Sie können ja dort beispielsweise ein benutzerdefiniertes Ribbon anzeigen, mit dem der Benutzer die gewünschten Elemente der Benutzeroberfläche öffnen kann.

Alternativ wählen Sie für die Eigenschaft **Startformular** der Access-Optionen das beim Starten der Anwendung anzuzeigende Formular an. Bei einer WPF-Anwendung müssen Sie zwingend ein Startfenster angeben, denn sonst würde beim Start ja überhaupt keine Benutzeroberfläche erscheinen. Aus diesem Grund enthält ein neues WPF-Projekt ja auch gleich nach dem Start ein solches Fenster.

Wie aber gehen Sie vor, wenn Sie beispielsweise ein anderes Fenster beim Start der Anwendung anzeigen möchten? Dazu müssen Sie einfach nur den Inhalt der Datei **App.xaml** ändern. Dort gibt es im **Application**-Element ein Attribut namens **StartupUri**, das standardmäßig den Wert **MainWindow.xaml** enthält. Geben Sie hier einfach den Namen des beim Start anzuzeigenden Fensters ein (siehe Bild 5). Auch zu **App.xaml** gibt es eine entsprechende C#-Datei namens **App.xaml.cs**. Die **App.xaml**-Datei ist wichtig, weil sie der Startpunkt der Anwendung ist und, wie oben bereits beschrieben, den Namen des anzuzeigenden Fensters enthält.

Startformular vs. StartupUri

Im Prinzip entspricht das für das Attribut **StartupUri** angegebene Fenster dem Startformular, das Sie in den Optionen einer Access-Datenbank angeben können. In beiden Fällen wird das entsprechende Fenster beziehungsweise Formular aufgerufen.

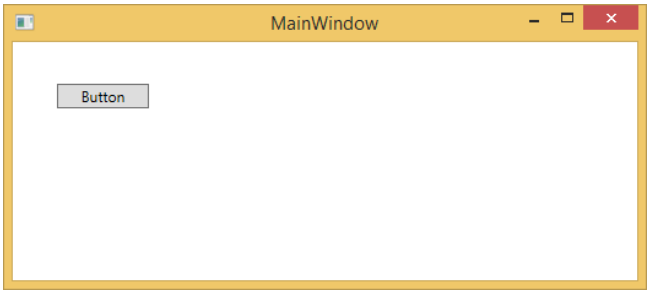


Bild 4: Erster Test des neuen Fensters

Beispielprojekt: Den Code für die Beispiele der folgenden Abschnitte finden Sie im Verzeichnis **Access-ZuWPF_Startmethode**.

Und AutoExec?

Unter Access haben Sie auch die Möglichkeit gehabt, ein Makro namens **AutoExec** anzugeben, um direkt beim Start Aktionen auszuführen. Dieses muss nicht zwangsläufig ein Formular öffnen, sondern kann auch Makroaktionen ausführen oder eine VBA-Funktion aufrufen.

In WPF-Projekten haben Sie eine ähnliche Möglichkeit. Dazu bearbeiten Sie die Datei **App.xaml**, indem Sie im Element **Application** das Attribut **StartupUri** entfernen und das Element **Startup** hinzufügen. Das gelingt übrigens prima per Intelli-Sense (siehe Bild 6).

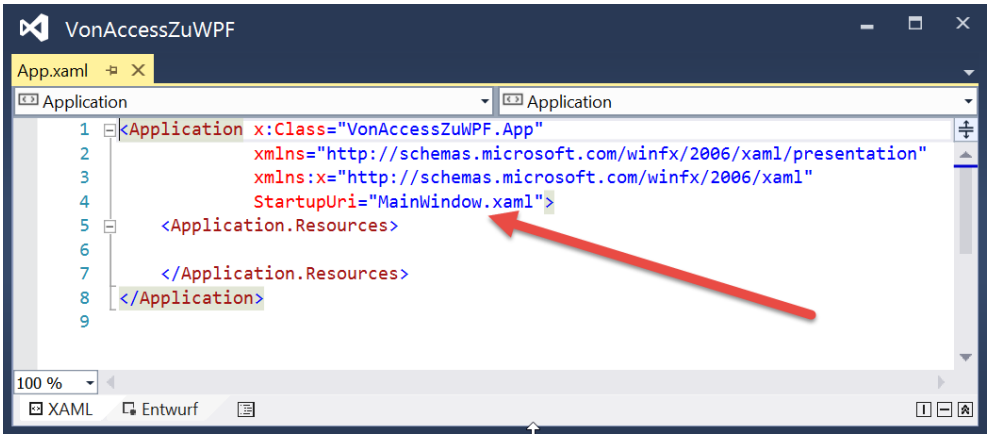


Bild 5: Angabe des Startfensters

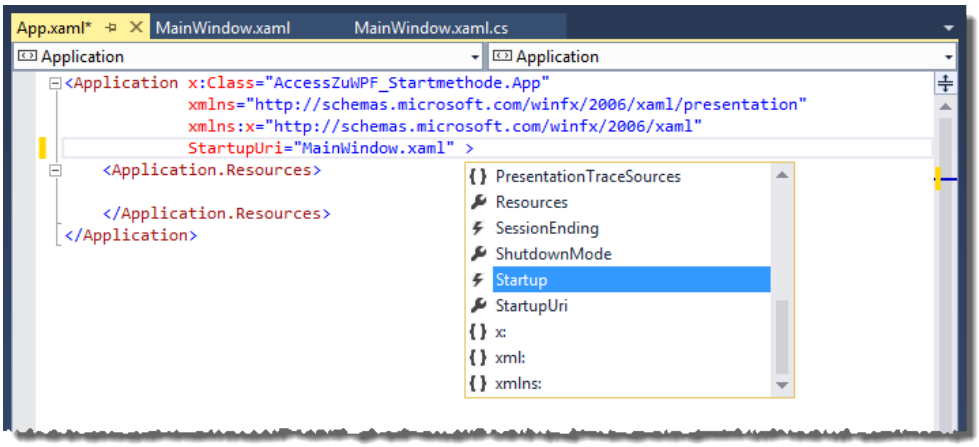


Bild 6: Auswahl des Attributs zur Angabe einer **Startup**-Methode

Nun müssen Sie nur noch die entsprechende Methode anlegen, und zwar in der Datei **App.xaml.cs**. Der Code dieser Klasse sieht dann wie folgt aus.

```
namespace AccessZuWPF_Startmethode {
    public partial class App : Application {
        private void Application_Startup(object sender,
                                         StartupEventArgs e)
        {
            MainWindow wnd = new MainWindow();
            wnd.Title = "Hauptfenster";
            wnd.Show();
        }
    }
}
```

Die hier enthaltene Methode **Application_Startup** erzeugt zunächst ein neues Objekt auf Basis unserer Fenster-Klasse **MainWindow** und weist dieses der Variablen **wnd** zu. Dann stellt sie den Titel auf Hauptfenster ein und blendet das Fenster schließlich mit der **Show**-Methode ein. Damit erhalten Sie auch gleich einen der Vorteile dieser Variante: Sie können die Eigenschaften des Fensters bereits vor dem Anzeigen ändern. Sie sehen also, dass Sie die Eigenschaften von Fenstern und Steuerelementen nicht nur über die Entwurfsansicht und den XAML-Code sowie über das Eigenschaften-Fenster anpassen können,

Datenzugriff mit ADO.NET, Teil 1

Wer lange mit Access gearbeitet hat, dem ist der Datenzugriff über die DAO-Bibliothek mit dem Recordset-Objekt, das Bearbeiten mit AddNew, Update und Edit sowie das Formulieren von SQL-Aktionsabfragen für die Execute-Anweisung in Fleisch und Blut übergegangen. Nun heißt es umdenken: Unter der Datenzugriffsbibliothek ADO.NET sieht vieles anders aus. Diese Artikelreihe zeigt, wie Sie die unter Access gelernten Abläufe auch mit Visual Studio unter C#/ADO.NET programmieren können.

ADO.NET ist nicht etwa der Nachfolger der zwischenzeitlich aufgekommenen ADODB-Bibliothek, die Sie auch in Access nutzen konnten (und können). Es ist eine komplett neue Bibliothek, die Microsoft für das .NET-Framework geschaffen hat.

ADO.NET bietet teilweise Datenzugriff mit offener Verbindung, teilweise verbindungslosen Zugriff an. Unter Access können Sie beispielsweise ein ADODB-Recordset mit den Daten einer Tabelle füllen und dann die Verbindung lösen, um die Daten zu verändern, ohne dass sich dies auf die zugrunde liegende Tabelle auswirkt. Dies geschieht mit vielen Objekten unter ADO.NET genauso.

Provider

Unter Access konnten Sie entweder direkt über die **CurrentDb**- oder die **OpenDatabase**-Funktion ein **Database**-Objekt referenzieren und dann mit Methoden wie **Execute**, **OpenRecordset** et cetera schreibend oder lesend/schreibend auf die Daten zugreifen. Oder Sie haben per ODBC mit einer entsprechenden Verbindungszeichenfolge auf weitere Datenbanken wie MySQL oder Microsoft SQL Server zugegriffen. Unter C# sieht dies etwas anders aus.

Hier verwenden Sie zwar auch noch eine Verbindungszeichenfolge, jedoch legen Sie sich bereits zuvor auf einen Datenprovider fest.

Es gibt beispielsweise die folgenden Provider:

- **SqlClient**: Microsoft SQL Server
- **OleDb**: Zugriff über einen OleDb-Provider, beispielsweise für den Zugriff auf Access-Datenbanken.
- **Odbc**: Zugriff auf Datenbanken über den jeweiligen ODBC-Treiber, zum Beispiel auf MySQL-Datenbanken oder ältere Access-Datenbanken wie Access 97.

Je nachdem, welchen Sie benötigen, fügen Sie Ihrer Klasse einen Verweis auf den jeweiligen Namespace hinzu:

```
//Für OleDb:
using System.Data.OleDb;
//Für Odbc:
using System.Data.Odbc;
//Für SQL Server:
using System.Data.SqlClient;
```

Beispieldatenbank

Für die Beispiele dieses Artikels verwenden wir eine Access-Datenbank, da wir diese einfach als Datei mitliefern können. Das heißt, dass wir den **OleDb**-Provider nutzen wollen:

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

Die Beispieldatenbank ist die Südsturm-Datenbank, die viele Leser vermutlich schon vom Magazin **Access im Unternehmen** kennen.

Die Datenbank enthält einige Tabellen, die wir für die folgenden Beispiele nutzen werden. Bild 1 zeigt das Datenmodell dieser Datenbank.

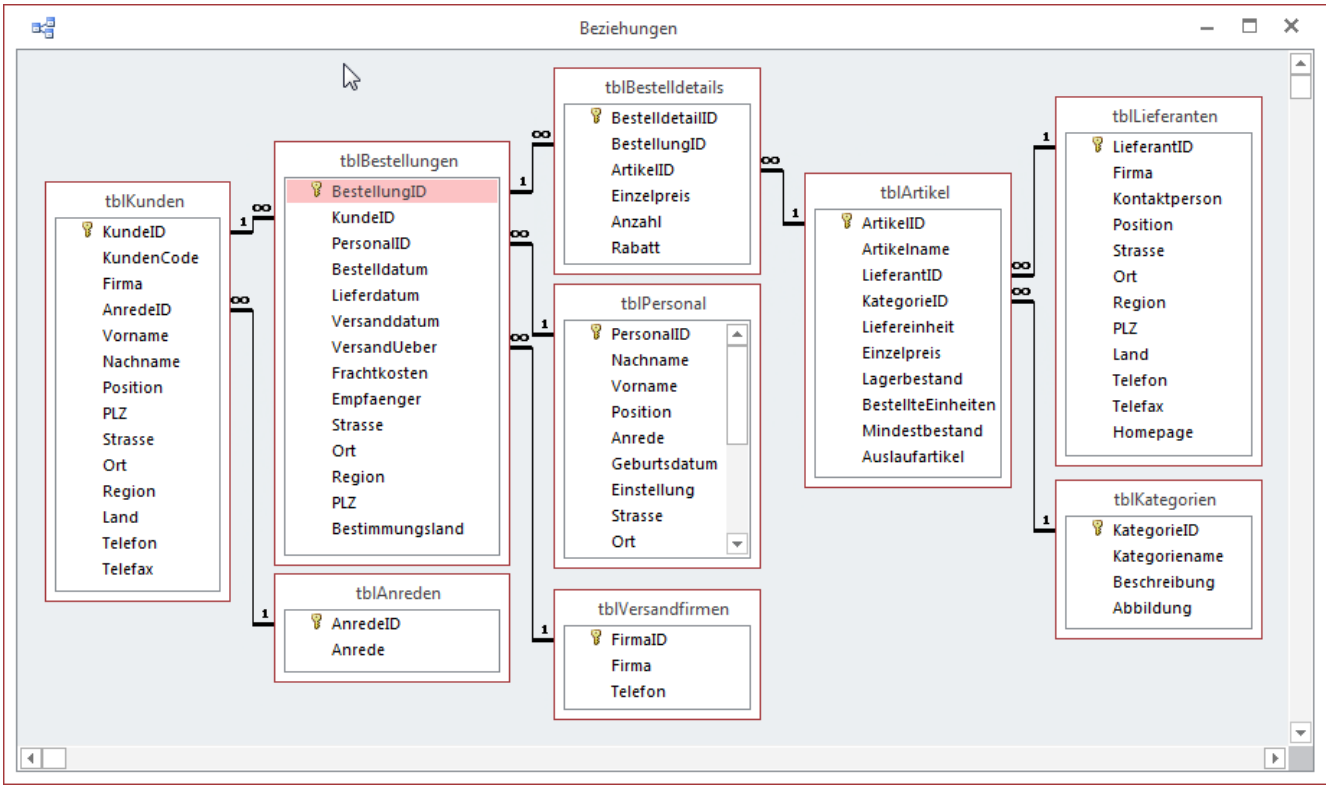


Bild 1: Datenmodell der Beispieldatenbank

Speicherort der Datenbank

Während der Tests werden wir die Anwendung von Visual Studio aus mit **F5** starten. Die **ADONET.exe**-Datei wird dann im Unterverzeichnis **\bin\debug** des Projektordners erstellt werden.

Damit wir ohne weitere Pfadangaben auf die Datei **Suedsturm.mdb** zugreifen können, fügen wir auch die Datenbankdatei zu diesem Verzeichnis hinzu.

Beispielprojekt

Als Beispielprojekt verwenden wir zunächst eine Konsolenanwendung (siehe Ordner **ADONET** im Download). Die einzelnen Beispiele können Sie jeweils in die Methode **Main** einfügen und ausführen.

Verbindung aufbauen

Grundlage für den Zugriff auf eine Datenbank ist der Aufbau einer Verbindung. Dies erledigen Sie unter C# wie in

```
static void Main(string[] args)
{
    string Connectionstring = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Suedsturm.mdb";
    OleDbConnection cnn = new OleDbConnection(Connectionstring);
    cnn.Open();
    Console.WriteLine("Provider: {0}", cnn.Provider);
    Console.WriteLine("DataSource: {0}", cnn.DataSource);
    Console.ReadLine();
    cnn.Close();
}
```

Listing 1: Herstellen einer Verbindung und Ausgeben einiger Verbindungseigenschaften

der Methode aus Listing 1. Die Methode definiert zunächst eine Verbindungszeichenfolge, wie Sie sie möglicherweise auch von **ADODB** kennen.

Dann erstellt die Methode ein neues Objekt des Typs **OleDbConnection** mit der Verbindungszeichenfolge als Parameter und speichert den Verweis darauf in der Variablen **cnn**. Dessen **Open**-Methode öffnet schließlich die Verbindung und folgenden Anweisungen geben den **Provider** (**Microsoft.Jet.OLEDB.4.0**) und die **DataSource** (**Suedsturm.mdb**) in der Konsole aus.

Objekte nach Datenquelle

Es gibt unter C# kein **Connection**-Objekt wie unter Access, das Sie mit einer datenquellenspezifischen Verbindungszeichenfolge versehen. Stattdessen finden Sie hier verschiedene Objekte vor, zum Beispiel die folgenden:

- **OleDbConnection**
- **SqlConnection**
- **OdbcConnection**

Gleiches gilt auch für die übrigen Objekte weiter unten vorgestellten Objekte: Wenn Sie die Beispiele also nicht mit Access, sondern mit einer anderen Datenquelle ausprobieren möchten, ersetzen Sie das Präfix des jeweiligen Objekts (also etwa **OleDb...** durch **Sql...**, wenn Sie auf den SQL Server zugreifen möchten).

Aktionsabfragen ausführen

Unter VBA gibt es mehrere Möglichkeiten, Aktionsabfragen auszuführen. Dabei führen Sie die angegebene SQL-Anweisung (**INSERT INTO**, **UPDATE**, **DELETE** oder **SELECT INTO**) zum Beispiel mit der **Execute**-Methode des **Database**-Objekts oder mit der **RunSQL**-Methode des **DoCmd**-Objekts aus:

```
'VBA-Code:
Dim db As DAO.Database
Set db = CurrentDb
db.Execute "DELETE FROM tblArtikel", dbFailOnError
```

oder

```
'VBA-Code:
DoCmd.RunSQL "DELETE FROM tblArtikel"
```

Es gibt auch noch eine ADODB-Methode, die wir hier nicht mehr aufführen wollen. Unter C# verwenden wir das Objekt **OleDbCommand** (beziehungsweise **SqlCommand** oder **OdbcCommand** für andere Datenquellen als Access), um eine Aktionsabfrage abzusetzen. Ein Beispiel sieht wie

```
...
string Connectionstring = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Suedsturm.mdb";
int AnzahlGeaenderteDatensaetze;
OleDbConnection cnn = new OleDbConnection(Connectionstring);
OleDbCommand cmd = new OleDbCommand();
cmd.CommandText = "UPDATE tblArtikel SET Artikelname = 'Chai (Tee)' WHERE Artikelname = 'Chai'";
cmd.Connection = cnn;
cnn.Open();
AnzahlGeaenderteDatensaetze = cmd.ExecuteNonQuery();
Console.WriteLine("Geänderte Datensätze: {0}", AnzahlGeaenderteDatensaetze);
Console.ReadLine();
cnn.Close();
...
```

Listing 2: Ausführen einer Aktualisierungsabfrage

in Listing 2 aus. Die dortigen Anweisungen (auszuprobieren innerhalb der **Main**-Methode des Konsolenprojekts) stellen wieder die Verbindungszeichenfolge zusammen. Die folgende Anweisung deklariert eine **int**-Variable namens **AnzahlGeaenderteDatensaetze** zum Erfassen der Anzahl der von der folgenden Änderungsabfrage betroffenen Datensätze.

Wie in der vorherigen Methode erstellen wir auch hier wieder ein **OleDbConnection**-Objekt. Außerdem benötigen wir ein Objekt des Typs **OleDbCommand** (referenziert mit der Variablen **cmd**).

Die Aktionsabfrage weisen wir der Eigenschaft **CommandText** des **OleDbCommand**-Objekts zu und legen über die Eigenschaft **Connection** das Verbindungsobjekt aus **cnn** fest. Die **Open**-Anweisung öffnet die Verbindung und die **ExecuteNonQuery**-Methode des **OleDbCommand**-Objekts führt die Aktionsabfrage aus. Sie liefert als Ergebnis die Anzahl der betroffenen Datensätze, die in der Variablen **AnzahlGeaenderteDatensaetze** landet. Die **Console.Write**-Methode gibt das Ergebnis in der Konsole aus.

DLookup und Co. unter ADO.NET

Wer sich in Access nicht mit SQL-Anweisungen auseinandersetzen wollte, hatte gute Karten: Abfragen ließen sich mit der Entwurfsansicht zusammenstellen und Detailinformationen zu einzelnen Datensätzen oder aggregierte Daten ließen sich mit den Domänenfunktionen wie **DLookup**, **DMax**, **DMin**, **DSum** et cetera zusammentragen.

Letztlich benötigten diese Funktionen aber fast genauso viele Informationen wie Sie zum Zusammenstellen einer entsprechenden SQL-Anweisung brauchten. Unter ADO.NET gibt es die hier genannten Vereinfachungen nicht mehr, dafür aber eine spezielle Methode des **OleDbCommand**-Objekts (beziehungsweise **OdbcCommand** oder **SqlCommand**), welche genau einen Wert aus einer Datensatzgruppe zurückliefert.

Diese Methode heißt **ExecuteScalar()**. Das Beispiel aus Listing 3 zeigt, wie es funktioniert. Die Anweisungen stellen wieder die Verbindungszeichenfolge zusammen und erstellen das **OleDbConnection**- und das **OleDbCommand**-Objekt. Die dem **OleDbCommand**-Objekt zugewiesene SQL-Abfrage braucht keine besonderen Voraussetzungen zu erfüllen, denn **ExecuteScalar** liefert den Wert des ersten Feldes des ersten gefundenen Datensatzes.

Wenn Sie also die Abfrage **SELECT ArtikelID, Artikelname FROM tblArtikel** aufrufen, erhalten Sie den Wert des Feldes **ArtikelID** für den ersten Datensatz – in diesem Fall **1**. Natürlich macht es keinen Sinn, der Abfrage mehr als ein Ausgabefeld mitzugeben, da ohnehin nur das erste Feld abgefragt wird.

```
...
string Connectionstring = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Suedsturm.mdb";
OleDbConnection cnn = new OleDbConnection(Connectionstring);
OleDbCommand cmd = new OleDbCommand();
cmd.CommandText = "SELECT Count(*) FROM tblArtikel";
cmd.Connection = cnn;
cnn.Open();
int AnzahlArtikel;
AnzahlArtikel = (int)cmd.ExecuteScalar();
Console.WriteLine("Anzahl der Artikel: {0}", AnzahlArtikel);
Console.ReadLine();
cnn.Close();
```

Listing 3: Abrufen eines einzelnen Wertes über eine Abfrage

Objektorientierte Programmierung, Teil 1

Wer mit VBA programmiert, nutzt eigentlich schon eine Reihe der Elemente der objektorientierten Programmierung – zum Beispiel, wenn er Objekte auf Basis der Klasse Database oder Recordset erstellt und dann mit deren Methoden und Eigenschaften durch die Daten einer Tabelle navigiert. Auch benutzerdefinierte Klassen lassen sich mit VBA erstellen. Unter C# erhalten Sie jedoch Zugriff auf den kompletten Umfang der objektorientierten Programmierung. Dieser Artikel liefert einen Einstieg in die objektorientierte Programmierung mit C#.

Prozeduren starten ade

Eines müssen wir direkt klarstellen: Einfach mal eben wie im VBA-Editor eine Prozedur in einem Standardmodul programmieren und diese mit **F5** ausführen – das gelingt in Visual Studio unter C# nicht. Dort müssen Sie etwa in einer Konsolenanwendung zumindest die standardmäßig vorhandene Klasse verwenden, die dort enthaltene Prozedur **Main** mit den gewünschten Codezeilen füllen und auf **F5** drücken.

Die Klasse Program

Wenn Sie ein neues Projekt auf Basis der Vorlage **Konsolenanwendung** erstellen, erhalten Sie eine Datei mit einer Klasse namens **Program** mit einer Methode namens **Main**, die standardmäßig beim Start der Anwendung ausgeführt wird.

Von der Methode **Main** aus werden wir gleich starten, um unsere erste selbst erstellte Klasse zu instanzieren und zu nutzen.

Eine Klasse pro Datei

Jede Klasse sollte in einer eigenen Datei erstellt werden, um die Übersicht zu gewährleisten. Sollten Sie mehrere Klassen in einer Datei anlegen, finden Sie solche Klassen, deren Name sich von dem der Datei unterscheidet, nicht mehr so schnell über den Projektmappen-Explorer wieder.

Neue Klasse erstellen

Dementsprechend beginnen wir nun, indem wir eine neue Klasse erstellen. Dies erledigen wir über den Menüeintrag **Projekt|Klasse hinzufügen...** Dies öffnet den Dialog **Neues Element** hinzufügen und stellt die Vorlage direkt auf den Eintrag **Klasse** ein. Wir nennen die Klasse **Konto** und klicken dann auf Hinzufügen (siehe Bild 1). Alternativ öffnen Sie den Dialog **Neues Element** mit dem Kontextmenü-Eintrag **Hinzufügen|Klasse...** des Eintrags für das Projekt im Projektmappen-Explorer.

Die neue Klasse erscheint nun sowohl im Projektmappen-Explorer als auch mit dem vollständigen Quellcode im

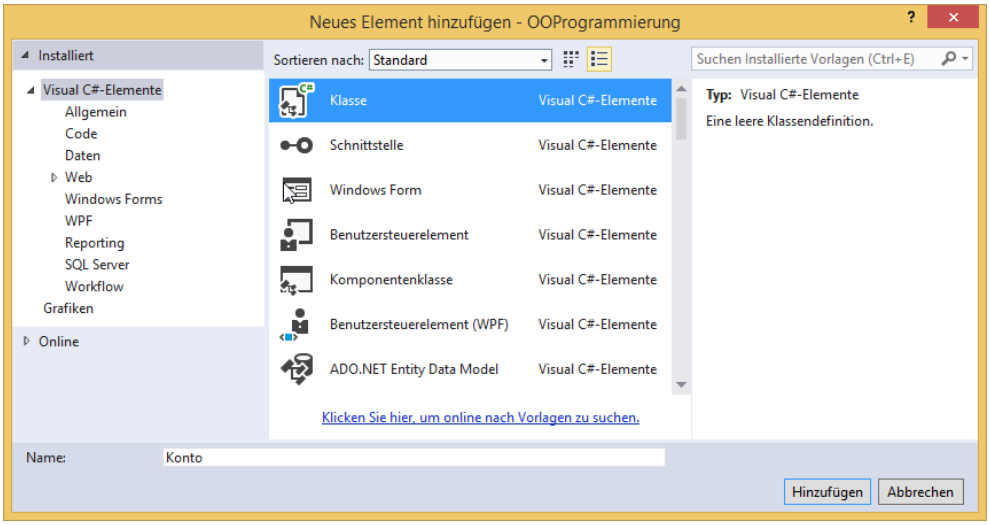


Bild 1: Hinzufügen einer neuen Klasse

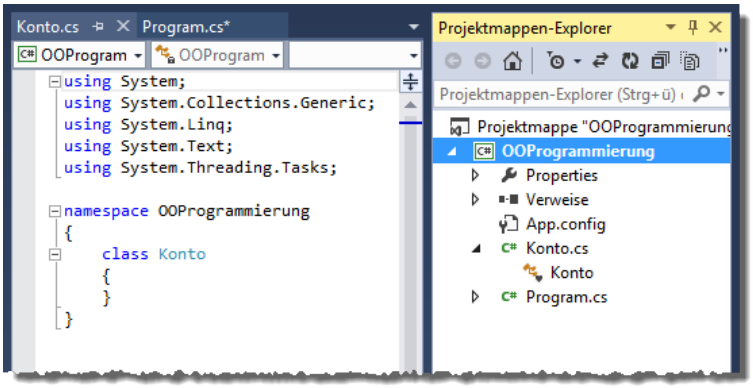


Bild 2: Die neue, noch unberührte Klasse **Konto**

Code-Fenster (siehe Bild 2). Abgesehen von den **using**-Anweisungen, dem **namespace**- und dem **class**-Element ist die Klasse noch jungfräulich:

```
namespace OOProgrammierung {
    class Konto {

    }
}
```

Benennung

Der Klassenname sollte möglichst genau beschreiben, welchem Objekt der Realität die Klasse entspricht – sofern dies gegeben ist. Soll die Klasse eine Person aufnehmen, heißt sie **Person**, soll sie wie in unserem Fall ein Konto aufnehmen, lautet der Klassenname **Konto**.

Wir verwenden kein Präfix, außerdem beginnt der Klassenname mit einem Großbuchstaben. Besteht der Klassenname aus mehreren zusammengesetzten Wörtern, sollten auch die folgenden Wörter jeweils wieder groß geschrieben werden.

Wenn Sie den Namen der Datei mit der Klasse später noch einmal ändern möchten, erledigen Sie da am besten über den Projektmappen-Explorer. Visual Studio ändert dann optional alle Verweise auf diese Klasse. Wenn Sie den Dateinamen **Konto** in **Bankkonto** ändern, benennt Visual Studio auch automatisch die Klasse im Code um.

Objekt instanzieren

Die Klasse ist zwar noch leer (sie hat noch keine Methoden, Eigenschaften et cetera), aber wir können diese dennoch bereits deklarieren und auch instanzieren. Dazu wechseln Sie nun zur Klasse **Program**. Dort erweitern Sie die Methode **Main** um zwei Zeilen:

```
static void Main(string[] args) {
    Konto konto;
    konto = new Konto();
}
```

Die erste Zeile deklariert ein neues Objekt namens **konto** mit dem Typ **Konto** (man beachte die Groß-/Kleinschreibung). Beides können Sie auch in einer einzigen Anweisung erledigen:

```
Konto konto = new Konto();
```

Die Deklaration reserviert den Speicher für das Objekt, die **new**-Anweisung erstellt das Objekt nach dem in der Klasse definierten Bauplan. Mit dem Objekt können wir nun noch nicht viel anfangen, da wir noch keine Methoden oder Eigenschaften hinzugefügt haben.

Sie können auch mehrere Objekte auf Basis der gleichen Klasse erstellen:

```
Konto konto1 = new Konto();
Konto konto2 = new Konto();
```

Sichtbarkeit der Klasse

Wenn Sie mit Access gearbeitet haben, kennen Sie die Klassenbibliotheken, die Sie üblicherweise per Verweis in ein VBA-Projekt eingebunden haben. Diese haben dann, sowohl über den Objektkatalog einsehbare als auch im Code referenzierbare Klassen bereitgestellt. Wenn Sie eine Klasse in einem C#-Projekt erstellen, um von anderen Anwendung darauf zuzugreifen, müssen Sie die Deklaration dieser Klasse mit dem Schlüsselwort **public** versehen:

```
public class Konto {

}
```

Wenn Sie das Schlüsselwort **public** nicht angeben, verwendet die Klasse die Standardeinstellung, welche dem Einsatz des Schlüsselworts **internal** entspricht und den Zugriff von außerhalb der Klassenbibliothek verwehrt.

Objekt freigeben

Objekte belegen Arbeitsspeicher. Unter VBA haben Sie eine Objektvariable durch Setzen auf den Wert **Nothing** freigegeben:

```
VBA-Code:
Set rst = Nothing
```

Unter C# stellen Sie das Objekt auf den Wert **null** ein:

```
konto = null;
```

Damit können Sie nun nicht mehr auf das Objekt zugreifen. Der Speicherplatz wird jedoch erst freigegeben, wenn die Garbage Collection startet und das Objekt löscht. Dies geschieht je nach Auslastung (bei hoher Auslastung seltener) und verfügbarem Speicherplatz (wenn der Platz knapp wird) oder bei Schließen der Anwendung.

Eigenschaften einer Klasse

Unsere Konto-Klasse soll verschiedene Eigenschaften bieten, zum Beispiel **Kontoinhaber**, **Bankleitzahl**, **Kontonummer**, **Kontostand** und **Dispositionsrahmen**. Um diese Eigenschaften nach dem Instanzieren eines Objekts auf Basis der Klasse **Konto** zu lesen und auch zu schreiben, deklarieren Sie diese wie folgt:

```
class Konto {
    public string Kontoinhaber;
    public string Kontonummer;
    public string Bankleitzahl;
    public decimal Kontostand;
    public decimal Dispositionsrahmen;
}
```

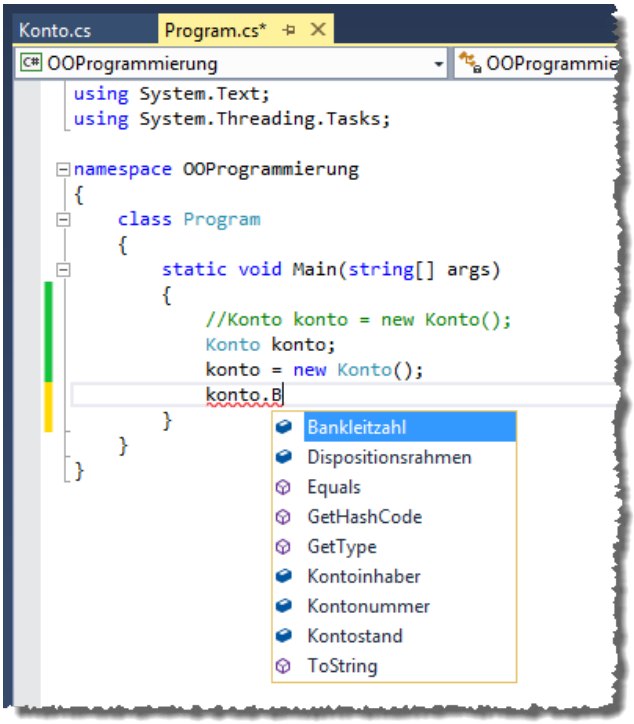


Bild 3: Eigenschaften per IntelliSense

Sie können dann von einer anderen Klasse aus, in diesem Fall **Program**, per IntelliSense auf die Eigenschaften zugreifen (siehe Bild 3). Dort sehen Sie noch einige weitere Methoden, beispielsweise **Equals** oder **GetType**.

Eigenschaften zuweisen und ausgeben

Das folgende Beispiel deklariert und instanziert das Objekt **konto** auf Basis der Klasse **Konto**, weist den Eigenschaften Werte zu und gibt diese dann in der Konsole aus:

```
Konto konto;
konto = new Konto();
konto.Kontoinhaber = "André Minhorst";
konto.Kontonummer = "1234567890";
konto.Bankleitzahl = "87654321";
konto.Kontostand = 100;
konto.Dispositionsrahmen = 1000;
Console.WriteLine("Kontoinhaber: {0}", konto.Kontoinhaber);
```

```
Console.WriteLine("Kontonummer:  {0}", konto.Kontonummer);
Console.WriteLine("Bankleitzahl: {0}", konto.Bankleitzahl);
Console.ReadLine();
```

Datenkapselung

Von VBA kennen Sie es vielleicht bereits: Die Eigenschaften einer Klasse werden nie als öffentliche Variablen deklariert, sondern mit den Schlüsselwort **Private**. Stattdessen erstellen Sie sogenannte **Property Get**- und **Property Let**- (für Werttypen) oder **Property Set**-Methoden (für Referenztypen), um auf die Inhalte der jeweiligen Variablen zuzugreifen. Hier haben wir dann zunächst eine private Variable für den Wert der Eigenschaft deklariert:

```
'VBA-Code:
Private m_Kontostand As Currency
```

Die **Property Let**-Methode erlaubte dann das Setzen des Wertes dieser Variablen von außerhalb der Klasse:

```
'VBA-Code:
Public Property Let Kontostand(curKontostand As Currency)
    m_Kontostand = curKontostand
End Property
```

Mit der **Get**-Methode konnte man den Wert hingegen lesen:

```
'VBA-Code:
Public Property Get Kontostand As Currency
    Kontostand = m_Kontostand
End Property
```

Durch diese Methoden war es möglich, beispielsweise Eingaben auf ihre Gültigkeit hin zu prüfen. Unter C# sieht das ähnlich aus, allerdings ist der Aufbau etwas anders. Die Deklaration ändern wir, indem wir das Schlüsselwort **public** auf **private** ändern. Außerdem soll die Schreibweise mit dem großen Anfangsbuchstaben für die Eigenschaft nach außen hin erkennbar sein. Das heißt, dass wir den Namen der Variablen ändern müssen – Namen für Variablen, Methoden, Eigenschaften et cetera müssen innerhalb eines Namespaces eindeutig sein.

Also ändern wir beim Variablennamen den großen Anfangsbuchstaben in einen kleinen:

```
private string kontoinhaber;
```

Wenn Sie dies zu unübersichtlich finden, können Sie alternativ auch etwa den Namen mit Großbuchstaben verwenden und diesem einen Unterstrich voranstellen – also etwa **_Kontoinhaber**. Außerdem fügen wir eine öffentliche Eigenschaft mit einem **get**- und einem **set**-Block hinzu. Diese wird wieder mit großem Anfangsbuchstaben notiert:

```
public string Kontoinhaber {
    get {
        return kontoinhaber;
    }
    set {
        kontoinhaber = value;
    }
}
```

Sie enthält einen **get**-Block, der mit der **return**-Anweisung den aktuellen Inhalt der Variablen **kontoinhaber** zurückliefert. Der **set**-Block erwartet einen neuen Wert für die Eigenschaft **Kontoinhaber**, den er an die Variable **kontoinhaber** übergibt. Die übrigen Eigenschaften implementieren wir ähnlich.

Dies lässt sich übrigens auch etwas weniger raumgreifend codieren, wie folgende Beispiel zeigt – und wie es vor allem bei größeren Mengen von Get/Set-Methoden sinnvoll ist:

```
public string Kontoinhaber {
    get {return kontoinhaber;}
    set {kontoinhaber = value;}
}
```

Möglichkeiten der Kapselung

Die Kapselung ermöglicht es beispielsweise, die Übergabe von Werten an die Eigenschaften zu validieren und die Werte gegebenenfalls nicht zu übernehmen. Im Falle der Eigenschaft **Dispositionsrahmen** sollen beispielsweise keine positiven Werte eingegeben werden, sondern nur negative Werte oder der Wert **0**. Die Variable deklarieren wir wie auch die anderen Variablen als private Variable. Dadurch kann man von außen nicht mehr direkt auf die Variable zugreifen, sondern nur noch über die Get/Set-Methode:

```
private decimal dispositionsrahmen;
```

Die Get/Set-Methode für die Eigenschaft **Dispositionsrahmen** verwendet einen **get**-Block wie auch die übrigen Get/Set-Methoden. Der **set**-Block sieht jedoch anders aus. Er enthält eine **if**-Bedingung, die prüft, ob der übergebene Wert kleiner als **0** ist. In diesem Fall erscheint eine entsprechende Meldung und der Wert wird nicht in die Variable **dispositionsrahmen** übernommen:

```
public decimal Dispositionsrahmen {
    get {
        return dispositionsrahmen;
    }
    set {
        if (value < 0) {
            Console.WriteLine("Der Dispositionsrahmen darf
                               kein negativer Wert sein.");
        }
        else {
            dispositionsrahmen = value;
        }
    }
}
```

Im Vergleich zu VBA verwendet der **get**-Block die **return**-Anweisung, um den Inhalt der privaten Variablen zurückzuliefern und der **set**-Block nimmt den neuen Wert über die **value**-Variable entgegen.

Die Zuweisung aller notwendigen Werte für das Objekt **konto** auf Basis der Klasse **Konto** inklusive der anschließenden Ausgabe zwecks Test der **get**-Blöcke sieht nun so aus:

```
Konto konto;
konto = new Konto();
konto.Kontoinhaber = "André Minhorst";
konto.Kontonummer = "1234567890";
konto.Bankleitzahl = "87654321";
konto.Kontostand = 100;
konto.Dispositionsrahmen = -1000;
Console.WriteLine("Kontoinhaber:       {0}",
                                  konto.Kontoinhaber);
Console.WriteLine("Kontonummer:        {0}",
                                   konto.Kontonummer);
Console.WriteLine("Bankleitzahl:       {0}",
                                  konto.Bankleitzahl);
Console.WriteLine("Kontostand:         {0}",
                                    konto.Kontostand);
Console.WriteLine("Dispositionsrahmen: {0}",
                               konto.Dispositionsrahmen);
Console.ReadLine();
```

Die Übergabe eines negativen Wertes für den Dispositionsrahmen scheitert hier und zeigt den Text aus Bild 4 in der Konsole an.

Schreib- und Leseschutz

Manche Eigenschaften sollen entweder nicht gelesen oder nicht geschrieben werden können. Für den Kontostand macht es beispielsweise Sinn, diesen nur lesend zu gestalten, aber nicht schreibend. Um den Kontostand zu ändern, kann man alternativ Methoden zur Klasse hinzufügen, mit denen ein Umsatz gebucht und der Kontostand entsprechend geändert wird – mehr dazu weiter unten.

Für das Lesen wie für das Schreiben gilt: Sie müssen einfach nur den **get**- beziehungsweise den **set**-Block weglassen, um den jeweiligen Zugriff zu sperren. Wenn es für eine Set/Get-Methode keinen **set**-Block gibt, dann