HANSER

Leseprobe

Walter Doberenz, Thomas Gewinnus

Visual C# 2012 - Kochbuch

ISBN (Buch): 978-3-446-43438-7

ISBN (E-Book): 978-3-446-43605-3

Weitere Informationen oder Bestellungen unter

http://www.hanser-fachbuch.de/978-3-446-43438-7

sowie im Buchhandel.

Kapitel 1

Kleiner C#-Anfängerkurs

Vor Beginn des Windows-Zeitalters wurden Programme geschrieben, die für jede Ein- und Ausgabe von Daten eine neue Textzeile in einem Bildschirmfenster (Konsole) erzeugten. Dieses Verhalten war ein Erbe aus jener Zeit, als lediglich Fernschreiber für die Ein- und Ausgabefunktionen von Computern zur Verfügung standen.

Aber auch heute noch kann es für den Newcomer durchaus sinnvoll sein, wenn er für seine ersten Schritte die gute alte Konsolentechnik verwendet. So kann er sich doch auf das zunächst Wesentliche, nämlich die Logik von C#-Befehlen, konzentrieren, ohne von der erdrückenden Vielfalt der Windows-Oberfläche abgelenkt zu werden.

Die folgende Serie von sechs absoluten Anfängerbeispielen benutzt zunächst Konsolenanwendungen, um einige grundlegende C#-Sprachelemente zu demonstrieren. Die letzten beiden Beispiele zeigen dann den Übergang zur zeitgemäßen Windows-Programmierung.

HINWEIS: Der "Kleine C#-Crashkurs" kann keinesfalls das Studium einführender C#-Literatur ersetzen (siehe z.B. unser Buch [Visual C# 2012 – Grundlagen und Profiwissen]), sondern ist lediglich als Ergänzung zu verstehen.

R1 Das EVA-Prinzip anwenden

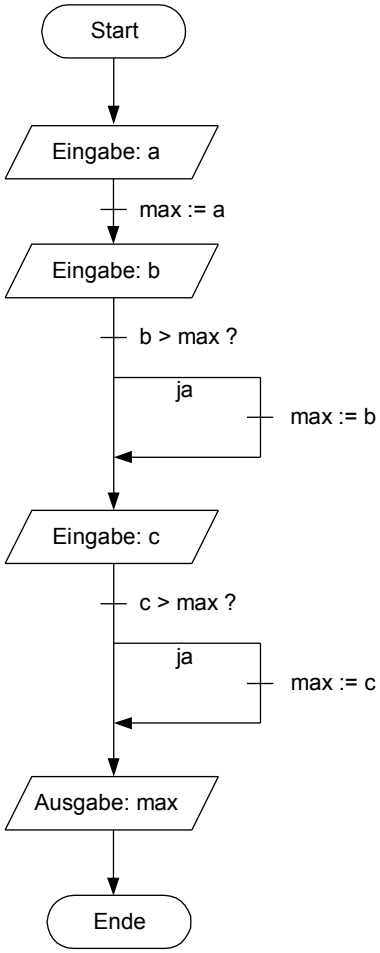
Jeder Weg, und ist er noch so weit, beginnt mit dem ersten Schritt! Für den Anfang soll uns deshalb ein ganz einfaches Beispiel genügen.

Aufgabenstellung

Nacheinander sind drei positive ganze Zahlen einzugeben. Das Programm soll die größte der drei Zahlen ermitteln und das Ergebnis anzeigen.

Lösungsvorschlag

Einen Vorschlag (Algorithmus) für den Programmablauf zeigt der nachfolgende Plan.



Sie erkennen daran das altbekannte EVA-Prinzip eines Programms, wobei die Anweisungen in der Reihenfolge

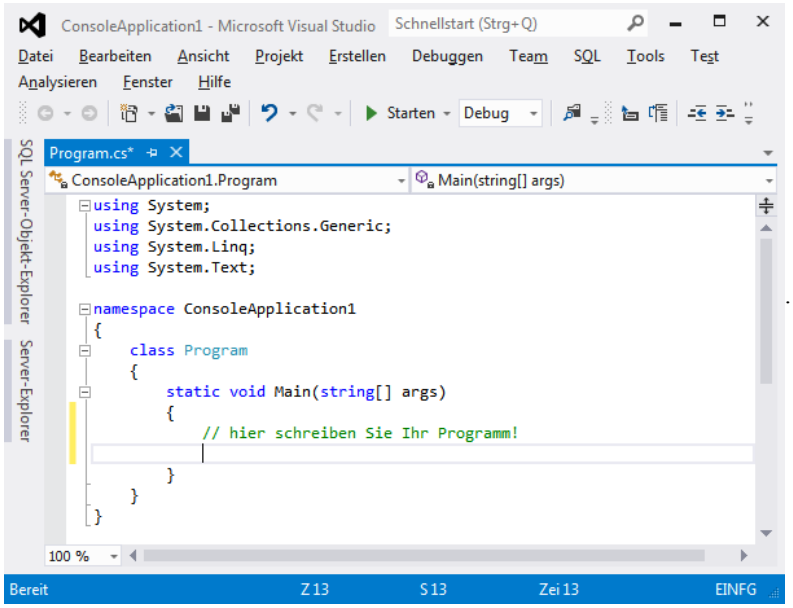
- **E**ingabe,
- **V**erarbeitung,
- **A**usgabe

ausgeführt werden.

Programmierung

Ein Programmablaufplan (PAP), wie ihn die obige Abbildung zeigt, ist zwar heute etwas aus der Mode gekommen, für den Anfänger kann er aber ganz nützlich sein, um die Logik eines C#-Programms besser zu verstehen.

Wählen Sie auf der Startseite von Visual Studio den Link *Neu/Projekt...*. Es öffnet sich das Dialogfenster "Neues Projekt". Wählen Sie links die Vorlage *Visual C#* und in der Mitte den Projekttyp *Konsolenanwendung*. Die am unteren Fensterrand sichtbaren Einträge für *Name* und *Projektmappenname* können auf den Standardeinstellungen *ConsoleApplication1* verbleiben. Nachdem Sie den *Ort* (das Verzeichnis, in welchem Ihr Projekt abgespeichert werden soll) festgelegt haben, erscheint das Fenster des Quellcode-Editors, in welchem bereits ein Codegerüst "vorgefertigt" ist. Ihren eigenen Code fügen Sie in der *Main*-Methode hinzu (zwischen beiden geschweiften Klammern).



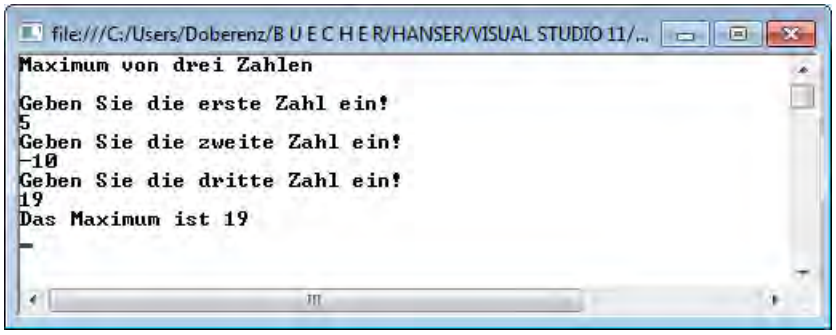
Ergänzen Sie, ohne länger darüber nachzudenken, den Code entsprechend folgendem C#-Listing (die mit "//" eingeleiteten Kommentare können Sie auch weglassen):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            Console.WriteLine("Maximum von drei Zahlen");          // Überschrift
            Console.WriteLine();                                   // Leerzeile
            int a, b, c, max;                                      //  Variablendeklaration
            Console.WriteLine("Geben Sie die erste Zahl ein!");
            a = Convert.ToInt32(Console.ReadLine());               // Eingabe a
            max = a;                                               // Initialisieren von max
            Console.WriteLine("Geben Sie die zweite Zahl ein!");
            b = Convert.ToInt32(Console.ReadLine());               // Eingabe von b
            if (b > max) max = b;                                  // Bedingung
            Console.WriteLine("Geben Sie die dritte Zahl ein!");
            c = Convert.ToInt32(Console.ReadLine());               // Eingabe c
            if (c > max) max = c;                                  // Bedingung
            Console.WriteLine("Das Maximum ist " + max.ToString());     // Ergebnisausgabe
            Console.ReadLine();         // Programm wartet auf die Enter-Taste, um zu beenden
        }
    }
}
```

Test

Kompilieren Sie das Programm (F5-Taste) und überzeugen Sie sich von der Funktionsfähigkeit:



HINWEIS: Durch Drücken der *Enter*-Taste beenden Sie die Anwendung.

Bemerkungen

Für den Anfänger hier noch einmal eine Zusammenstellung einiger wichtiger C#-Grundlagen, wie sie in diesem Beispiel zur Anwendung gekommen sind:

- Ein C#-Programm beginnt mit der *Main*-Methode, die gewissermaßen den Einsprungspunkt des Programms darstellt.

- Jedes C#-Programm besteht aus einer Folge von Anweisungen. Es besteht ein signifikanter Unterschied zwischen Groß- und Kleinschreibung!
- Jede C#-Anweisung wird mit einem Semikolon (;) abgeschlossen, der Zeilenumbruch spielt keine Rolle.
- Die *using*-Anweisungen zu Beginn des Programmes erleichtern den Zugriff auf einige wichtige Programmbibliotheken, die standardmäßig eingebunden werden.
- Die mit einem Doppelslash (*//*) eingeleiteten Anweisungen sind lediglich Kommentare und für den Programmablauf bedeutungslos.
- Vor Beginn eines Programms (bzw. eines in sich abgeschlossenen Teils) sind alle benötigten Variablen zu deklarieren, d.h., ihr Name und ihr Datentyp müssen festgelegt werden.
- Unter dem Begriff "Initialisierung einer Variablen" versteht man das Zuweisen eines Anfangswertes.
- *ReadLine* und *WriteLine* sind die einfachsten Ein-/Ausgabeanweisungen, wie Sie sie allerdings nur bei einer Konsolenanwendung verwenden sollten. Die Endung *...Line* der Befehlswörter bewirkt einen Zeilenvorschub.
- Die *if* -Anweisung. (wenn ... dann ...) ist ein Verzweigungsbefehl und führt eine Anweisung in Abhängigkeit von einer Bedingung aus.

R2 Ein Array definieren und initialisieren

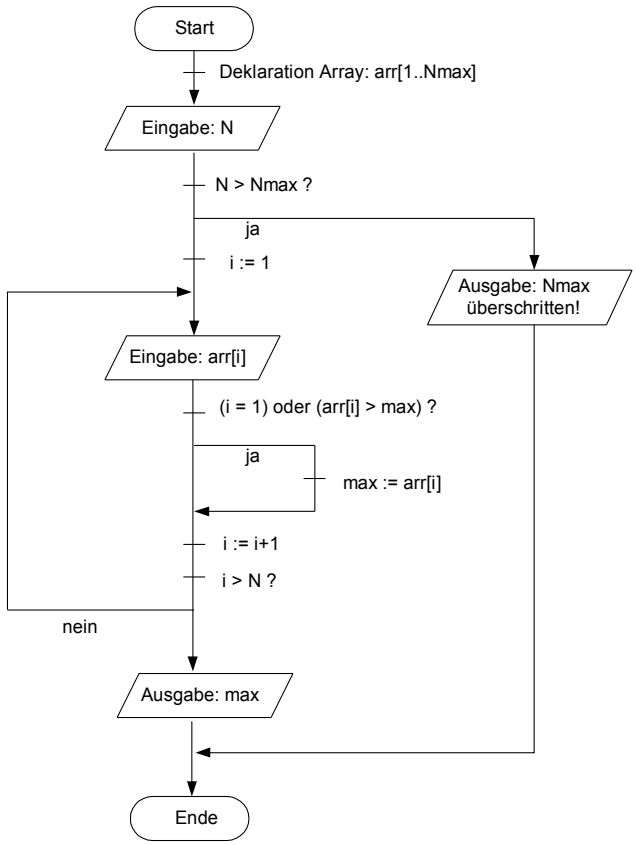
Nachteilig am Vorgängerbeispiel war vor allem die Beschränkung auf drei Zahleneingaben.

Aufgabenstellung

Erweitern Sie das Programm so, dass es die Eingabe einer flexiblen Anzahl von Werten ermöglicht!

Lösungsvorschlag

Der im nachfolgenden PAP dargestellte Algorithmus geht davon aus, dass die Zahlenwerte nicht mehr in Einzelvariablen *a, b, c*, sondern in einem Array gespeichert werden. Stellen Sie sich ein solches Array wie ein Regal mit einzelnen Fächern vor, in denen beliebig große Zahlenwerte abgelegt werden. Die Fächer sind beschriftet mit 1, 2, 3 ... Die Anzahl der Regalfächer beträgt *Nmax*. Diese Konstante ist ausreichend groß zu wählen, damit genügend Reserven für die maximale Anzahl *N* von Zahlenwerten vorhanden sind.



Programmierung

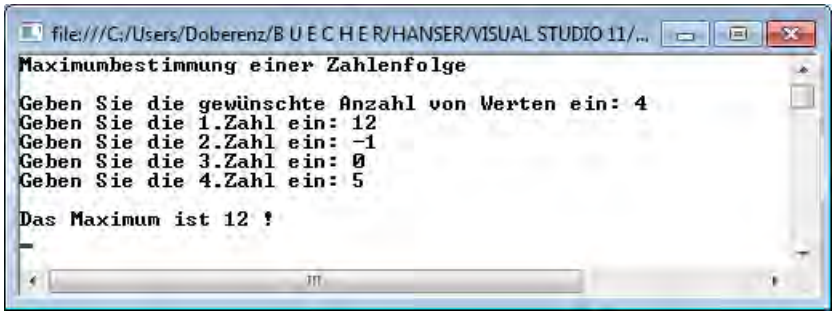
Die Umsetzung des PAP in ein C#-Programm dürfte Ihnen besonders dann leicht fallen, wenn Sie das Vorgängerbeispiel verstanden haben:

```
...
class Program
{
    static void Main(string[] args)
    {
        const int Nmax = 10;                // maximale Anzahl von Werten
        Console.WriteLine("Maximumbestimmung einer Zahlenfolge");      // Überschrift
        Console.WriteLine();                                  // Leerzeile
        int[] arr = new int[10];                              // Array dimensionieren
        int n = 0;
        int max = 0;
        Console.Write("Geben Sie die gewünschte Anzahl von Werten ein: ");
        n = Convert.ToInt32(Console.ReadLine());              // Eingabe n
        if (n > Nmax)                              // Bedingung trifft zu
        {
```

```
            Console.WriteLine("Es dürfen maximal nur " + Nmax.ToString() +
                              " Werte eingegeben werden!");
        }
        else                                        // Bedingung trifft nicht zu
        {
            for (int i = 1; i <= n; i++)                // Beginn der Schleife
            {
                 Console.Write("Geben Sie die " + i.ToString() + ".Zahl ein: ");
                 arr[i-1] = Convert.ToInt32(Console.ReadLine());   // Eingabe i-te Zahl in Array
                 if ((i == 1) | (arr[i - 1] > max)) max = arr[i - 1];
            }                                           // Ende der Schleife
            Console.WriteLine();
            Console.WriteLine("Das Maximum ist " + max.ToString() + " !");     // Ergebnis
        }
        Console.ReadLine();
    }
}
```

Test

Gleich nach Programmstart werden Sie zur Eingabe der Anzahl von Werten aufgefordert, die beliebig groß sein kann.



Bemerkungen

- Die Kommentare im Quelltext beschränken sich nur auf die Neuigkeiten gegenüber dem Vorgängerbeispiel, vor allem auf das Array und die *for*-Schleifenanweisung, die genau *n* mal durchlaufen wird.
- Ein Array-Index steht immer in eckigen Klammern hinter der Arrayvariablen. Der untere Index eines Arrays beginnt mit null, deshalb wird im Feld *arr[0]* die erste Zahl gespeichert.
- Die *ToString*-Funktion, über die jedes .NET-Objekt verfügt, verwandelt den Array-Index *i* (Datentyp *Integer*) in eine Zeichenkette (Datentyp *String*), damit eine Ausgabe über *Write* ermöglicht wird.

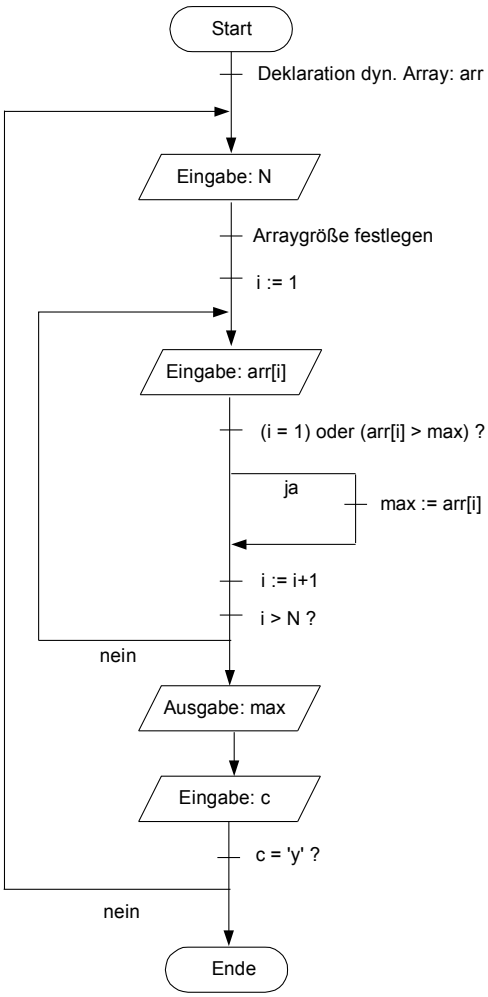
R3 Die Arraygröße zur Laufzeit ändern

Im Vorgängerbeispiel wurde das Programm nach einem einzigen Durchlauf beendet, danach war ein Neustart erforderlich.

Aufgabenstellung

Ergänzen Sie das Programm so, dass es nach Ermittlung des Maximums entweder wieder von vorn beginnt oder aber beendet werden kann!

Lösungsvorschlag



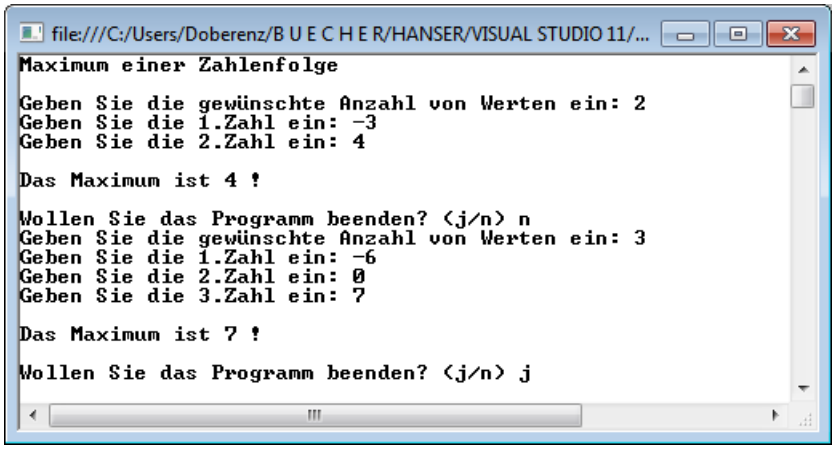
Programmierung

Der Programmablauf weist viele Analogien zum Vorgängerbeispiel auf, deshalb wird nur auf die Neuigkeiten per Kommentar hingewiesen:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Maximum einer Zahlenfolge");          // Überschrift
        Console.WriteLine();                                     // Leerzeile
        int[] arr = null;
        int n = 0;
        int max = 0;
        string c = String.Empty;
        do
        {
            Console.Write("Geben Sie die gewünschte Anzahl von Werten ein: ");
            n = Convert.ToInt32(Console.ReadLine());         // Eingabe n
            arr = new int[n];
            for (int i = 1; i <= n; i++)
            {
                Console.Write("Geben Sie die " + i.ToString() + ".Zahl ein: ");
                arr[i - 1] = Convert.ToInt32(Console.ReadLine());    // Eingabe i-te Zahl
                if ((i == 1) | (arr[i - 1] > max)) max = arr[i - 1];
            }
            Console.WriteLine();
            Console.WriteLine("Das Maximum ist " + max.ToString() + " !");    // Ergebnis
            arr = null;
            Console.WriteLine();
            Console.Write("Wollen Sie das Programm beenden? (j/n) ");
            c = Console.ReadLine();
        }
        while (c != "j");
    }
}
```

Test

Sie können den Berechnungszyklus jetzt beliebig oft wiederholen und dabei die Länge der einzugebenden Zahlenreihe neu festlegen (siehe folgende Abbildung).



HINWEIS: Nach Eingabe von "j" bzw. "n" müssen Sie die *Enter*-Taste drücken!

Bemerkungen

- Mit *null* (eine Zeigervariable auf "nichts") wird der vom Array belegte Speicherplatz wieder freigegeben.
- Die *do ... while*-Schleifenanweisung verlangt am Ende eine Abbruchbedingung.
- In unserem Beispiel dient die *String*-Variable *c* der Entgegennahme einer Benutzereingabe ("j" bzw. "n").

R4 Berechnungen in eine Methode auslagern

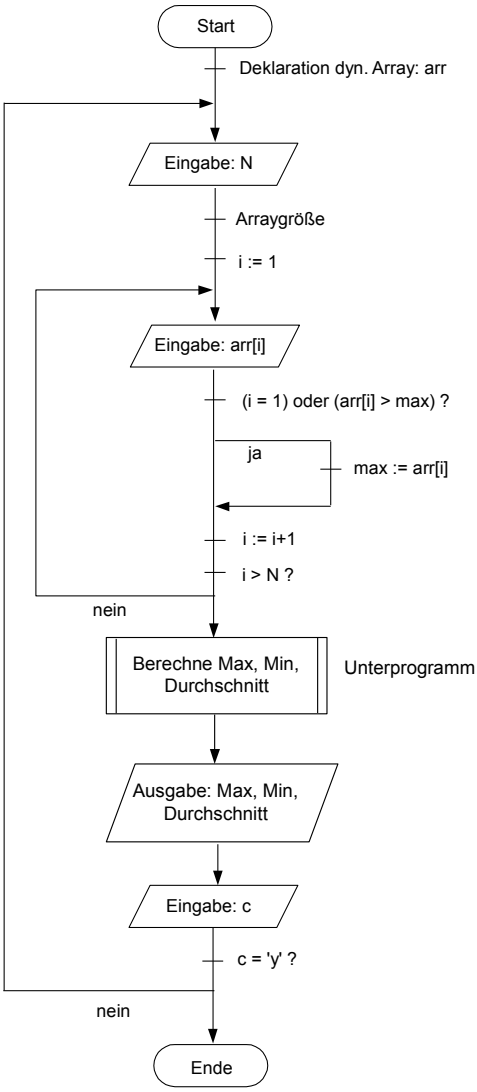
Es gibt kein Programm, das man nicht noch weiter verbessern könnte. Da sich von einer Zahlenreihe weitaus mehr Informationen gewinnen lassen als nur der Maximalwert, sollen Sie noch ein weiteres Problem lösen.

Aufgabenstellung

Ergänzen Sie das Programm, damit auch Minimum und Durchschnitt ausgegeben werden!

Lösungsvorschlag

Trotz der erweiterten Funktionalität erscheint der nachfolgend abgebildete PAP keinesfalls komplizierter als sein Vorgänger zu sein. Dies wurde vor allem durch Auslagern der Berechnungsfunktionen für Maximum, Minimum und Durchschnitt in ein Unterprogramm (eine Methode) erreicht.



Programmierung

Aufgrund der vielen Analogien zum Vorgängerbeispiel wird auch hier nur auf die Besonderheiten per Kommentar hingewiesen:

```
class Program
{
    // globale Variablen deklarieren:
    static int[] arr;
    static int max, min;
    static string av;
    static void berechne()            // hier beginnt die Methode
```

```
    {
        double sum = arr[0];           // lokale Variable

        min = arr[0]; max = arr[0];    // globale Variablen initialisieren
        int n = arr.Length - 1;
        for (int i = 1; i <= n; i++)
        {
            if (arr[i] > max)
                max = arr[i];
            else
                if (arr[i] < min) min = arr[i];
            sum += arr[i];
        }
        double d = sum / (n + 1);     // Durchschnitt berechnen
        av = d.ToString("#0.00");     // globale Variable zuweisen
    }                                 // hier endet die Methode

    static void Main(string[] args)
    {
        Console.WriteLine("Auswerten einer Zahlenfolge");         // Überschrift
        Console.WriteLine();                                      // Leerzeile

        int n = 0;
        string c = String.Empty;
        do
        {
            Console.Write("Geben Sie die gewünschte Anzahl von Werten ein: ");
            n = Convert.ToInt32(Console.ReadLine());          // Eingabe n
            arr = new int[n];
            for (int i = 1; i <= n; i++)
            {
                Console.Write("Geben Sie die " + i.ToString() + ".Zahl ein: ");
                arr[i - 1] = Convert.ToInt32(Console.ReadLine());    // Eingabe i-te Zahl
            }
            Console.WriteLine();
            berechne();                   // Zahlenfolge auswerten (Methodenaufruf)
```

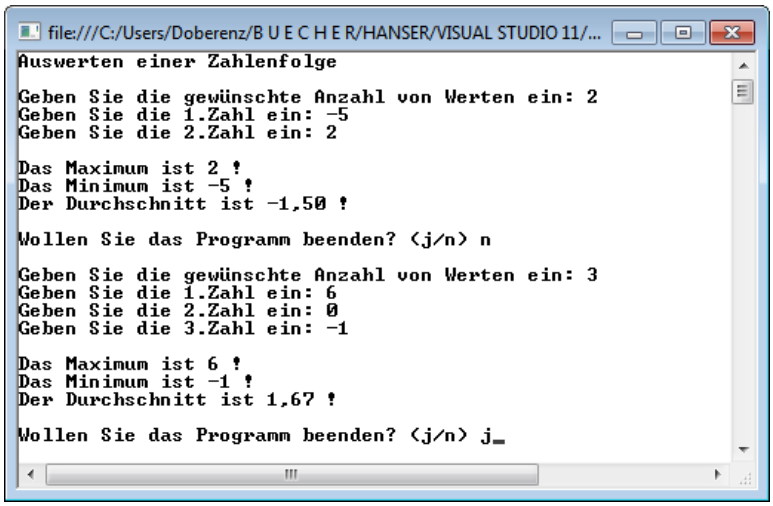
Globale Variablen anzeigen:

```
            Console.WriteLine("Das Maximum ist " + max.ToString() + " !");
            Console.WriteLine("Das Minimum ist " + min.ToString() + " !");
            Console.WriteLine("Der Durchschnitt ist " + av + " !");

            arr = null;
            Console.WriteLine();
```

```
            Console.Write("Wollen Sie das Programm beenden? (j/n) ");
            c = Console.ReadLine();
            Console.WriteLine();
        }
        while (c != "j");
    }
}
```

Test

Dieses Programms liefert weitaus mehr Informationen als seine Vorgänger:



Bemerkungen

- In unserem Quellcode wurden Variablen auf globaler Ebene (gültig innerhalb des gesamten Programms) und auf lokaler Ebene (gültig innerhalb der Methode *berechnen*) benutzt. Eine lokale Variable hat immer Vorrang vor einer gleichnamigen globalen Variablen.
- Die *ToString("#0.00")*-Methode besitzt hier einen so genannten Formatierungsstring als Argument, damit der Durchschnitt (eine Gleitkommazahl!) so in eine Zeichenkette umgeformt wird, dass immer zwei Nachkommastellen angezeigt werden.
- Die *Length*-Eigenschaft der *arr*-Variablen liefert die Anzahl der Array-Elemente. Da die Indizierung eines dynamischen Arrays stets mit null beginnt, ist der höchste Index immer um Eins niedriger.
- Ein Unterprogramm (hier die benutzerdefinierte Methode *berechne*) ist immer dann zweckmäßig, wenn die Übersichtlichkeit des Programms erhöht werden soll, oder aber wenn gleiche Codeabschnitte mehrfach ausgeführt werden sollen. Außerdem wird ein Wiederverwendbarkeit des Quellcodes erleichtert (siehe nächstes Beispiel R5).

- Nicht nur der trostlose schwarze Textbildschirm, auch die mühselige Bedienung, bei der der Rechner die Reihenfolge der Benutzereingaben zwangsweise vorgibt, wird Ihnen ein Dorn im Auge sein. Dies sollte ein wichtiger Grund dafür sein, von der Konsolen- zur Windows-Anwendung überzugehen (siehe folgende Beispiele).

R5 Konsolenprogramm nach Windows portieren

Wir wollen die langweiligen Konsolenanwendungen endlich hinter uns lassen und ab jetzt zeitgemäße Windows-Applikationen verwenden. Auch hier geht es mit einer ganz einfachen Aufgabe los.

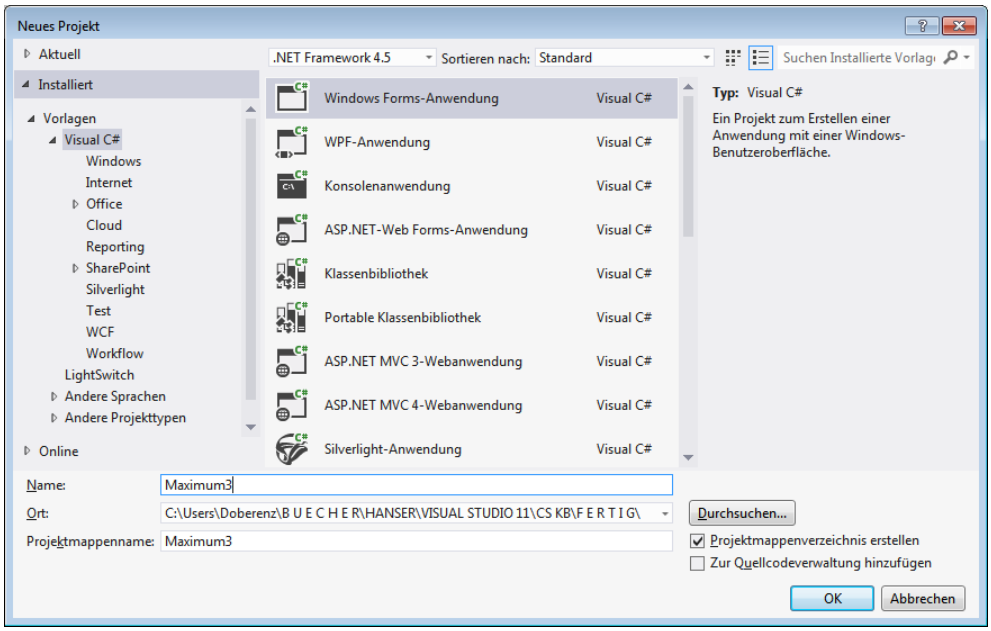
Aufgabenstellung

Lösen Sie mit einer Windows Forms-Anwendung das gleiche Problem (Maximumbestimmung von drei Integer-Zahlen) wie im Beispiel

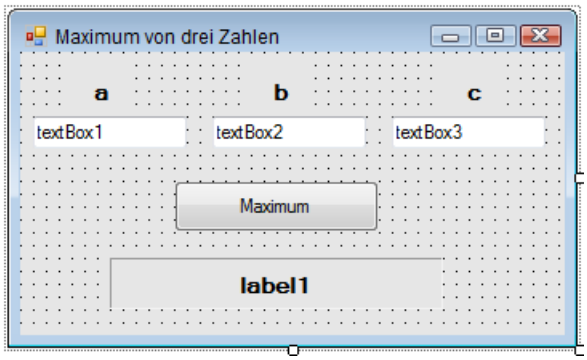
► R1 Das EVA-Prinzip anwenden

Lösungsvorschlag

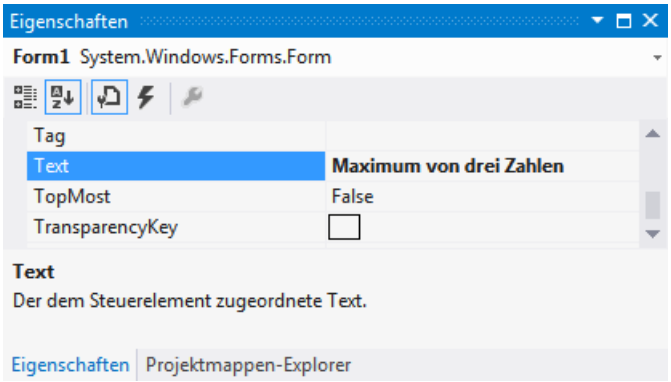
Starten Sie Visual Studio und öffnen Sie ein neues Projekt ("Visual C#", "Windows Forms-Anwendung"). Geben Sie als *Namen* z.B. "Maximum3" ein:



Mit F4 holen Sie sich das Eigenschaftenfenster von *Form1* in den Vordergrund und stellen damit die *Text*-Eigenschaft (das ist die Beschriftung der Titelleiste) des Startformulars *Form1* neu ein: "Maximum von drei Zahlen".



Vom Werkzeugkasten (*Strg+Alt+X*) ziehen Sie die Steuerelemente (3 mal *TextBox*, 1 mal *Button*, 4 mal *Label*) gemäß folgender Abbildung auf *Form1* und stellen auch hierfür bestimmte *Text*-Eigenschaften neu ein:



Programmierung

Durch einen Doppelklick auf *button1* wird automatisch das Codefenster der (partiellen) Klasse *Form1* mit dem bereits vorbereiteten Rahmencode des *Click*-Eventhandlers geöffnet. In diesem Zusammenhang ist ein für den Einsteiger wichtiger Hinweis zu beachten, der auch für die Zukunft gilt:

HINWEIS: In der Regel sollten Sie den Rahmencode der Eventhandler nicht selbst eintippen, sondern immer durch Visual Studio erzeugen lassen!

```
using System;
using System.Windows.Forms;
...
namespace Maximum3
{
    public partial class Form1 : Form
    {
...
        private void button1_Click(object sender, EventArgs e)
        {
              // Hier müssen Ihre C#-Anweisungen eingefügt werden!
        }
    }
}
```

Füllen Sie den Körper des Eventhandlers mit den erforderlichen Anweisungen aus, sodass der komplette Eventhandler schließlich folgendermaßen aussieht:

```
private void button1_Click(object sender, EventArgs e)
{
    try
    {
        int a = Convert.ToInt32(textBox1.Text);     // Eingabe a
        int max = a;                                // Initialisieren von max
        int b = Convert.ToInt32(textBox2.Text);     // Eingabe b
        if (b > max) max = b;                       // Bedingung
        int c = Convert.ToInt32(textBox3.Text);     // Eingabe c
        if (c > max) max = c;                       // Bedingung
        label1.Text = "Das Maximum ist " + max.ToString() + " !";   // Ergebnisausgabe
    }
    catch(Exception ex)
    {
         MessageBox.Show(ex.Message, "Fehler");                    // Fehlermeldung
    }
}
```

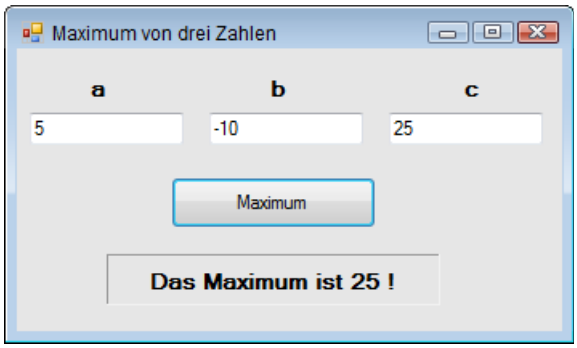
HINWEIS: Beim Vergleich mit der Konsolenanwendung erkennen Sie, dass Ein- und Ausgabe deutlich einfacher und übersichtlicher geworden sind!

Test

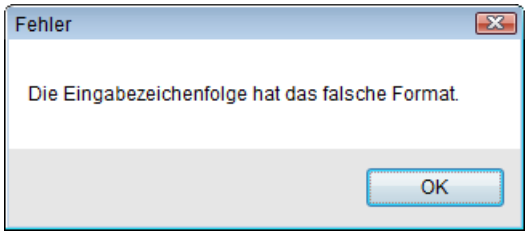
Nachdem Sie das Projekt abgespeichert haben, kompilieren und starten Sie das Programm mit der *F5*-Taste (oder durch Klick auf die entsprechende kleine Schaltfläche mit dem grünen Dreieck auf der Symbolleiste von Visual Studio).

Wenn Sie die Programmbedienung mit der bei einer Konsolenanwendung vergleichen, so stoßen Sie auf ein Hauptmerkmal der Windows-Programmierung: Eine fest vorgeschriebene Reihenfolge für die Benutzereingaben gibt es nicht mehr!

Sie können mit der *Tab*-Taste den Eingabefokus zwischen den einzelnen Komponenten der Bedienoberfläche verschieben!



In unserem Beispiel darf man nur Ganzzahlen eingeben. Dank Fehlerbehandlung mittels *trycatch*-Block stürzt das Programm aber bei falschen Eingaben nicht ab, sondern liefert eine höfliche Meldung:



Bemerkungen

- Genauso wie im Konsolen-Beispiel haben wir es auch auch hier mit C#-Anweisungen zu tun, die sich jetzt allerdings innerhalb einer Ereignisbehandlungsmethode (Event-Handler) befinden.
- Im Unterschied zur Konsolenanwendung (*ReadLine/WriteLine*) ist die Ein-/Ausgabe der Zahlen etwas umständlicher. Deren Werte sind zunächst in der *Text*-Eigenschaft der drei Textboxen enthalten und müssen mit Hilfe der *Convert*-Klasse vom Datentyp *String* in den *Integer*-Datentyp umgewandelt werden. Die Ergebnisausgabe erfolgt umgekehrt mittels der bereits bekannten *ToString*-Methode, deren Ergebnis der *Text*-Eigenschaft von *label1* zugewiesen wird.
- Wo ist – wie bei der Konsolenanwendung – die *class Program* geblieben? Öffnen Sie den Projektmappen-Explorer (Menü *Ansicht|Projektmappen-Explorer*) und doppelklicken Sie auf die Datei *Program.cs*. An deren Inhalt sehen Sie, dass an den C#-Fundamenten nicht gerüttelt wurde. Allerdings beschäftigt sich das Hauptprogramm nur noch mit dem Initialisieren der Windows-Anwendung und dem Erzeugen des Formulars. Der von Ihnen zu schreibende spezifische Code wurde in die Klasse *Form1* ausgelagert.

R6 Werte in einer ListBox anzeigen

Mittlerweile sind Sie auf den Geschmack gekommen und wollen sich an eine etwas anspruchsvollere Windows-Anwendung heranwagen. Schließlich verfügt Visual Studio nicht nur über solch einfache Steuerelemente wie *Label*, *TextBox* und *Button*, sondern über ein ganzes Arsenal attraktiver und leistungsfähiger Controls.

Aufgabenstellung

Verwandeln Sie das Konsolenprogramm aus

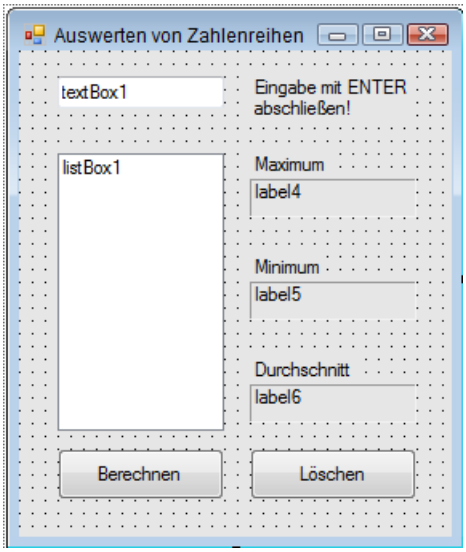
► R4 Berechnungen in eine Methode auslagern

in eine Windows Forms-Applikation und zeigen Sie die Zahlenreihe in einer *ListBox* an. Für die Eingabe sollen nicht nur ganze Zahlen, sondern auch Gleitkommazahlen zulässig sein.

Lösungsvorschlag

Da Sie die allgemeine Vorgehensweise bereits am Vorgängerbeispiel geübt haben, werden sich die folgenden Erläuterungen nur auf das Spezifische beschränken.

Die folgende Abbildung zur Gestaltung der Benutzerschnittstelle ist lediglich als Anregung zu verstehen. Eine *TextBox* dient zur Eingabe der Zahlen, die per *Enter*-Taste in eine *ListBox* übernommen werden sollen. Für die bequeme Bedienung und Anzeige sind weiterhin zwei *Button*- und drei *Label*-Steuerelemente vorgesehen.



Programmierung

In die Klasse *Form1* tragen Sie zunächst folgenden Code ein, wie er Ihnen bereits in ähnlicher Gestalt aus der entsprechenden Konsolenanwendung bekannt ist:

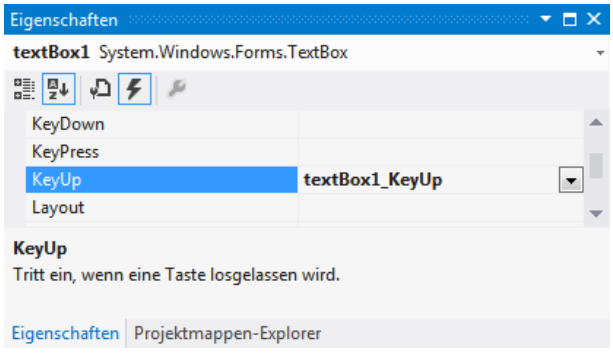
```
public partial class Form1 : Form
{
    private double[] arr;                // globale Variablen deklarieren
    private double max, min;
    private string av;

     private void berechne()
     {
        double sum = arr[0];             // lokale Variable deklarieren und initialisieren
        min = arr[0]; max = arr[0];      // globale Variablen initialisieren
        int n = arr.Length - 1;
        for (int i = 1; i <= n; i++)
        {
            if (arr[i] > max)
                max = arr[i];
            else
                if (arr[i] < min) min = arr[i];
            sum += arr[i];
        }
```

Durchschnitt berechnen:

```
        double d = sum / (n + 1);
        av = d.ToString("#0.00");     // globale Variable zuweisen
    }
```

Der folgende Event-Handler für das *KeyUp*-Ereignis von *textBox1* überträgt dann den Eintrag nach *listBox1*, wenn die *Enter*-Taste gedrückt wird. Anschließend wird der Eintrag in der Textbox gelöscht, um für neue Zahleneingaben bereit zu sein.

HINWEIS: Tippen Sie den Rahmencode dieser Event-Handler nicht per Hand ein, sondern erzeugen Sie ihn über die *Ereignisse*-Seite des Eigenschaftenfensters (F4)!



```
  private void textBox1_KeyUp(object sender, KeyEventArgs e)
    {
        if ((e.KeyCode == Keys.Enter) & (textBox1.Text != String.Empty))   // Enter-Taste
        {
            listBox1.Items.Add(textBox1.Text);     // TextBox => ListBox
            textBox1.Text = String.Empty;          // TextBox löschen
        }
    }
```

Ein weiterer Event-Handler wertet das *Click*-Ereignis von *Button1* aus. Der Inhalt der *ListBox* wird ausgelesen und in das Array kopiert. Da der erste Eintrag einer *ListBox* (genauso wie das erste Feld des Arrays) immer den Index null hat, kann man das Kopieren elegant in einer *for*-Schleife erledigen. Nach Aufruf der Methode *berechne* erfolgt dann die Ergebnisanzeige:

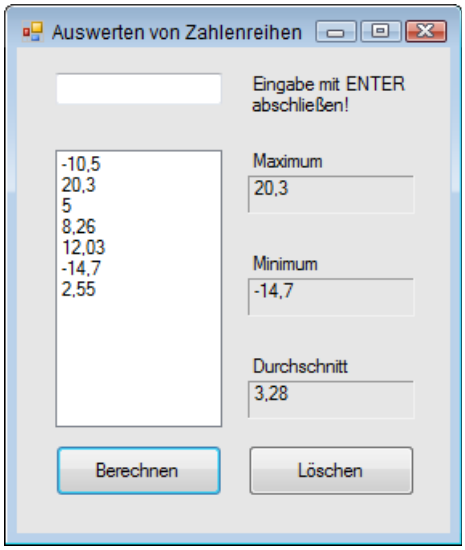
```
    private void button1_Click(object sender, EventArgs e)
    {
        int n = listBox1.Items.Count;     // Anzahl der ListBox-Einträge feststellen
        if (n > 0)                        // mindestens ein Eintrag in der ListBox
        {
            arr = new double[n];          // Array dimensionieren
            for (int i = 0; i < n; i++)
            try
            {
                arr[i] = Convert.ToDouble(listBox1.Items[i]);      // in ListBox kopieren
                 berechne();            // Methodenaufruf
            }
            catch (Exception ex)        // Fehlerbehandlung
            {
                   MessageBox.Show(ex.Message, "Fehler");
            }
            label4.Text = max.ToString();         // Ergebnisanzeige
            label5.Text = min.ToString();
            label6.Text = av;  // Gleitkommazahl formatieren
        }
    }
```

Der letzte Event-Handler ermöglicht das Löschen der gesamten Zahlenreihe, um wieder von vorn beginnen zu können:

```
    private void button2_Click(object sender, EventArgs e)
    {
         listBox1.Items.Clear();
         arr = null;                        // Array freigeben
         label4.Text = String.Empty;        // Anzeige löschen
         label5.Text = String.Empty;
         label6.Text = String.Empty;
    }
}
```

Test

Geben Sie eine beliebig lange Zahlenkolonne ein (jede Zahl mit *Enter*-Taste abschließen) und lassen Sie sich dann das Ergebnis der Auswertung anzeigen:



Bemerkungen

- Vergleichen Sie den Quellcode dieses Programms mit dem der Konsolenanwendungen aus den Vorgängerbeispielen. Nach einiger Praxis dürfte es Ihnen nicht schwer fallen, weitere "alte" Konsolenrogramme auf ähnliche Weise unter Visual Studio "aufzumöbeln". Dabei wird das "Zusammenschieben" der Benutzerschnittstelle nicht das Problem sein, da die Vorgehensweise starke Ähnlichkeiten mit einem Zeichenprogramm hat. Dreh- und Angelpunkt ist vielmehr das Aufbrechen des linearen Programmablaufs (PAP) und seine zweckmäßige Verteilung auf verschiedene Event-Handler.
- In einer Windows Forms-Anwendung hat der klassische Programmablaufplan (PAP) – zumindest außerhalb von Methoden bzw. Event-Handlern – seine Bedeutung weitestgehend verloren, da es keine festgelegte Reihenfolge der Benutzereingaben mehr gibt.
- Wenn Sie auch dieses letzte Beispiel ohne größere Schwierigkeiten gemeistert haben, sind Sie auf dem besten Weg zu einem hoffnungsvollen C#-Programmierer, denn Sie haben bereits ein Gefühl für die wichtigsten sprachlichen Grundlagen entwickelt. Die Zeit ist also reif, um sich an anspruchsvollere Projekte zu wagen.

Kapitel 8

Schnittstellen

R188 Eine einfache E-Mail versenden

Mit Hilfe der Anweisung "*mailto:*" ist es problemlos möglich, eine E-Mail mit Adressangabe, Betreffzeile und E-Mail-Text zu generieren, lediglich auf Dateianhänge müssen Sie verzichten. Aufgerufen wird die Anweisung mit Hilfe der *Start*-Methode eines *Process*-Objekts, siehe dazu

► R239 Ein externes Programm starten

HINWEIS: Das in R240 vorgestellte Programm können Sie für die folgenden Beispiele verwenden, Sie müssen jedoch die Zeile "*proc.WaitForExit()*" auskommentieren.

Folgende Varianten bieten sich an:

BEISPIEL: Eine einfache E-Mail ohne Betreffzeile oder Body-Text.

```
mailto:xyz@abc.com
```

BEISPIEL: Eine E-Mail mit einer Betreffzeile ("Preisanfrage").

```
mailto:xyz@abc.com?subject=Preisanfrage
```

BEISPIEL: Eine E-Mail mit Adresse, Betreffzeile und zusätzlicher Kopie an die Adresse "hans@glueck.com".

```
mailto:xyz@abc.com?subject=Preisanfrage&CC=hans@glueck.com
```

BEISPIEL: Eine E-Mail mit Adresse, Betreffzeile sowie einem E-Mail-Text.

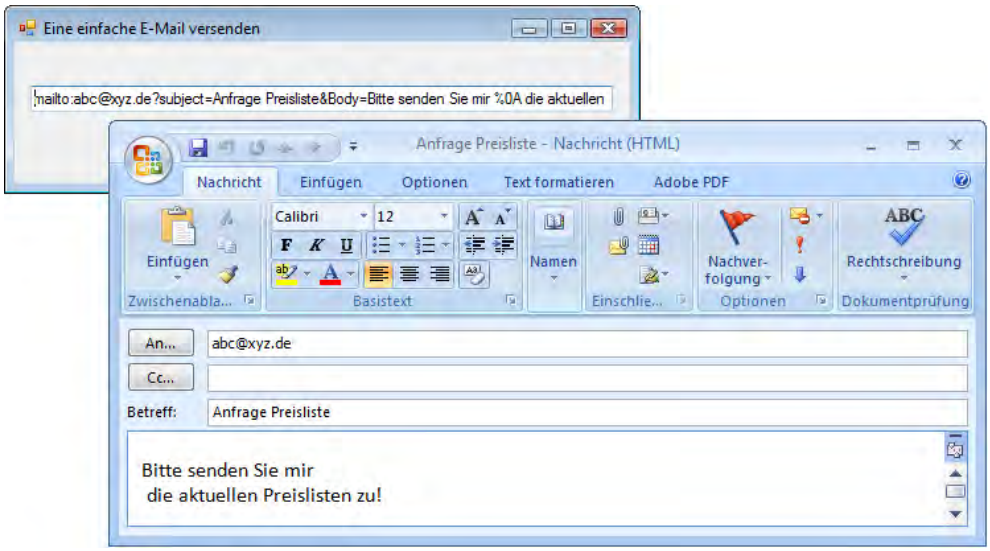
```
mailto:abc@xyz.de?subject=Anfrage Preisliste&Body=Bitte senden Sie mir %0A die aktuellen
Preislisten zu!
```

Wie Sie sehen, ist die Verwendung recht einfach. Es sind lediglich einige Grundregeln zu beachten:

- Die Betreffzeile (*subject*) ist mit einem Fragezeichen "?" von der Adressangabe zu trennen.
- Alle weiteren Optionen sind mit einem "&" voneinander zu trennen.
- Zeilenumbrüche in der Textangabe können Sie mit der Kombination "%0A" realisieren.
- Leerzeichen in der Adressangabe können Sie mit "%20" einfügen.

Test

Tragen Sie obige Anweisungen in das Programm ein und klicken Sie auf die *Start*-Schaltfläche:

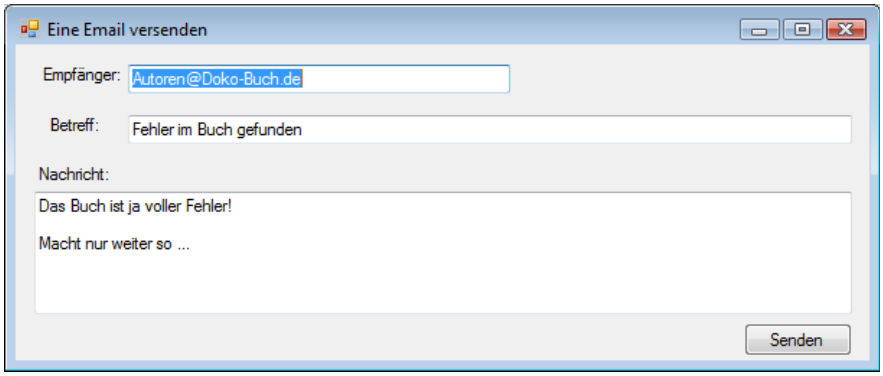


R189 E-Mails mit dem integrierten Mail-Client versenden

Der Ablauf für das Senden einer E-Mail in Windows Forms-Anwendungen entspricht der Vorgehensweise bei ASP.NET-Anwendungen (siehe Kapitel 16). Wir wollen deshalb die theoretischen Grundlagen nicht erneut auswalzen, sondern mit einem kleinen Beispielprogramm die Umsetzung für Windows Forms-Anwendungen demonstrieren.

Oberfläche

Ein Windows *Form*, drei *TextBox*en (Empfänger, Betreff, Nachricht), einige *Label*s für die Beschriftung sowie ein *Button* genügen:



Quelltext

```
using System.Net;
...
    public partial class Form1 : Form
    {
        private void button1_Click(object sender, EventArgs e)
        {
```

Instanz eines E-Mail-Clients erzeugen:

```
            System.Net.Mail.SmtpClient mail = new System.Net.Mail.SmtpClient();
```

Die eigentliche Message:

```
            System.Net.Mail.MailMessage msg = new
                        System.Net.Mail.MailMessage("leser@deutschland.de", textBox1.Text);
            msg.Subject = textBox2.Text;
            msg.Body = textBox3.Text;
```

HINWEIS: Hier müssen Sie Anpassungen entsprechend Ihres Mail-Servers und Ihrer Anmeldeinformationen vornehmen!

```
            mail.Credentials = new NetworkCredential("Hans", "Wurst");
            mail.Host = "server";
```

E-Mail versenden:

```
            mail.Send(msg);
        }
    }
}
```

Test

Nach dem Klick sollte die Mail schon auf dem Weg zu den Autoren sein ...

R190 Die Zwischenablage verwenden

Das *Clipboard*-Objekt kann mit einer ganzen Reihe von Methoden aufwarten, zu jedem der Datentypen *Data*, *Text*, *Audio*, *Image*, *FileDropList* gibt es eine *Contains*-, eine *Get*- und eine *Set*-Methode, z.B. *ContainsText*, *GetText* und *SetText*.

Eine *Contains*-Methode (*true*/*false*) überprüft, ob in der Zwischenablage eine Information im gewünschten Format vorliegt. Die *Get*- und *Set*-Methoden übernehmen das Kopieren bzw. Einfügen. Ein kleines Beispiel zeigt wie es funktioniert.

Oberfläche

In ein Windows *Form* fügen Sie eine *PictureBox*, eine *TextBox* (*MultiLine=True*) sowie vier *Button*s ein (siehe Laufzeitansicht).

Quelltext

```
public partial class Form1 : Form
{
```

Text in die Zwischenablage kopieren:

```
    private void button2_Click(object sender, EventArgs e)
    {
        if (textBox1.SelectedText != String.Empty)
            Clipboard.SetText((textBox1.SelectedText));
        else
            MessageBox.Show("Kein Text selektiert!");
    }
```

Text aus der Zwischenablage einfügen:

```
    private void button4_Click(object sender, EventArgs e)
    {
        if (Clipboard.ContainsText())
            textBox1.Text = Clipboard.GetText();
        else
            MessageBox.Show("Keine geeigneten Daten im Clipboard!");
    }
```

Grafik in die Zwischenablage kopieren:

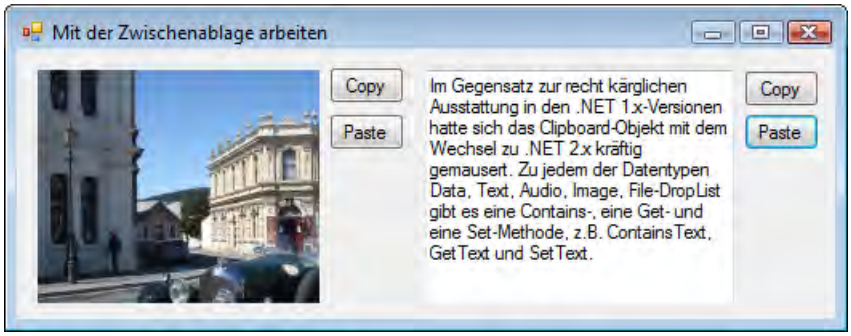
```
    private void button1_Click(object sender, EventArgs e)
    {
        if (pictureBox1.Image != null)
            Clipboard.SetImage(pictureBox1.Image);
        else
            MessageBox.Show("Keine Grafik enthalten!");
    }
```

Grafik aus der Zwischenablage einfügen:

```
        private void button3_Click(object sender, EventArgs e)
        {
            if (Clipboard.ContainsImage())
            {
                System.Drawing.Image pic = Clipboard.GetImage();
                pictureBox1.Image = pic;
            }
            else
                MessageBox.Show("Keine Bitmap im Clipboard!");
        }
    }
```

Test

Kopieren Sie mit einem Zeichenprogramm (z.B. Paint) eine Grafik in die Zwischenablage und versuchen Sie diese in das Programm einzufügen:



R191 Die WIA-Library kennenlernen

Geht es um den Zugriff auf externe Geräte, wie Scanner, Digitalkameras, Webcams etc., bietet sich dem Programmierer mit der Microsoft WIA-Library (*Windows Image Acquisition Automation Layer*) eine recht komfortable Variante an, deren Vorteil unter anderem darin besteht, ohne viel C#-Code auszukommen.

Doch worum geht es eigentlich? Bei WIA handelt es sich um eine komplexe Windows-COM-API, die

- das Auslesen statischer Bilder von Scannern, digitalen Kameras, WebCams,
- das Nachbearbeiten (Größe, Farbe, Drehen ...),
- das Kommentieren (Exif-Informationen)
- und das Konvertieren in diverse Dateiformate (PNG, JPG, GIF, BMP, TIFF) ermöglicht.

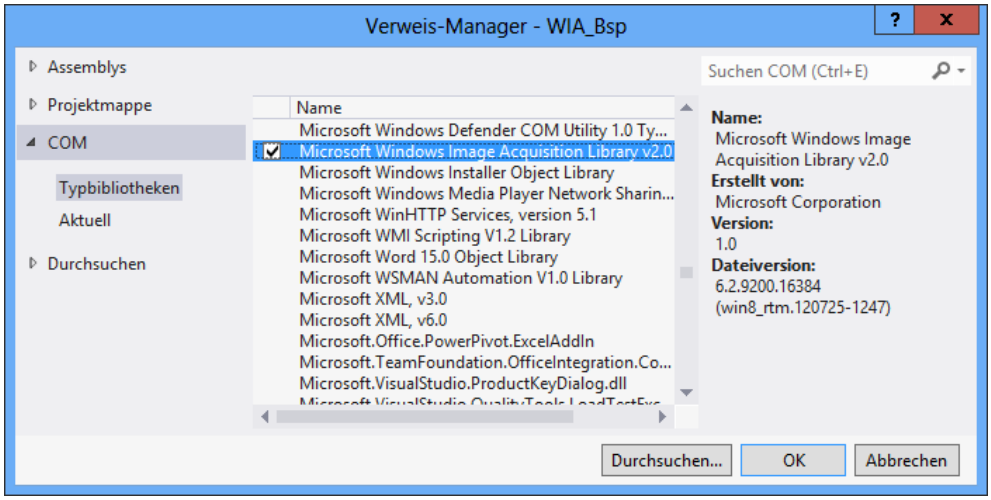
HINWEIS: WIA steht dem Programmierer als COM-Komponente in der Datei *wiaaut.dll* zur Verfügung.

Installation/Vorbereitung

Die Unterstützung beschränkt sich auf die Windows-Version XP ab SP1 und alle später erschienenen Windows Versionen (Vista, 7, 8, Server ...).

Windows Vista/Windows 7/Windows 8

Die komplette und aktuelle WIA-Libary ist bereits auf Ihrem System installiert, Sie können sofort loslegen, indem Sie die *Microsoft Windows Image Acquisition Library v2.0* in Ihr C#-Projekt einbinden. Erstellen Sie dazu ein neues Projekt und fügen Sie die o.g. Library über *Projekt|Verweis hinzufügen* hinzu (siehe folgende Abbildung).



HINWEIS: Ab Visual Studio 2010 wird die PIA[1] (bzw. die nötigen Datentypen) automatisch in die Anwendung eingebunden, eine Weitergabe der extra Assembly ist also nicht mehr nötig.

Windows XP

Hier ist noch etwas Nacharbeit angesagt, da sich auf Ihrem System derzeit nur die Version 1.x befindet. Laden Sie zunächst die aktuelle Version der WIA von der Microsoft-Webseite:

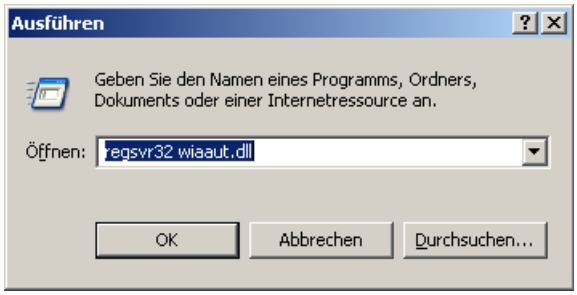
[1] *Primary Interop Assembly*

LINK: `http://www.microsoft.com/downloads/details.aspx?FamilyID=a332a77a-01b8-4de6-91c2-b7ea32537e29`

Entpacken Sie die ZIP-Datei und kopieren Sie enthaltene Datei *wiaaut.dll* in das *\\Windows\System32*-Verzeichnis.

HINWEIS: Ganz nebenbei finden Sie in dieser Datei auch einige VB6-Beispiele und eine umfangreiche Hilfedatei.

Die neue Library registrieren Sie über *Start/Ausführen*:



Binden Sie anschließend die Library, bzw. eine Referenz auf diese (siehe vorhergehender Abschnitt), in Ihr Projekt ein.

Einstieg mit kleiner Beispielanwendung

Auf Grund der umfangreichen Möglichkeiten dieser Library möchten wir Ihnen anhand einer kleinen Beispielanwendung einige wesentliche Features, die für das Einlesen und Weiterverarbeiten von Bildern unabdingbar sind, vorstellen:

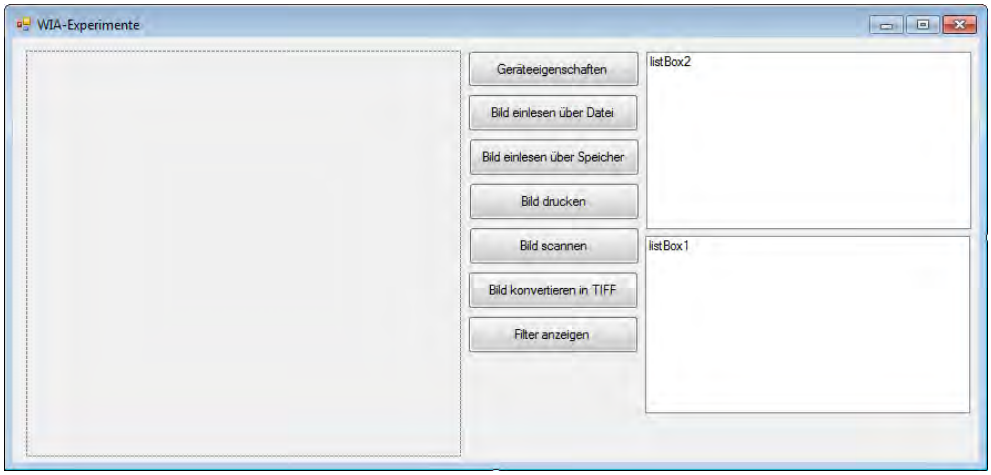
- Erkennen der angeschlossenen Geräte/Gerätetypen
- Auswerten von Ereignissen (Anschließen/Trennen)
- Anzeige der Geräteeigenschaften
- Einlesen von Bildern
- Verwendung des Scan-Assistenten
- Einsatz des Druckassistenten

Oberfläche

Entwerfen Sie zunächst eine Oberfläche entsprechend der folgenden Abbildung. Fügen Sie dazu zwei *ListBoxen* (Anzeige von Statusmeldungen, Anzeige der aktiven Geräte) sowie ein *PictureBox*-

Steuerelement in das Formular ein. Die Bedeutung der Schaltflächen dürfte aus der Beschriftung ersichtlich sein.

Abschließend fügen Sie noch eine *OpenFileDialog*-Komponente ein, mit der wir Bilddateien auswählen wollen.



Quellcode

Ein wichtiger Hinweis vorweg:

HINWEIS: Erzeugen Sie so wenig globale Objekte wie möglich, es besteht immer die Möglichkeit, dass der Anwender die externen Geräte vom PC trennt bzw. diese ausschaltet. In diesem Fall ist die Referenz auf das entsprechende Objekt futsch.

```
using System.IO;
```

Das zentrale Objekt für den Zugriff auf alle Geräte ist der *DeviceManager*:

```
namespace WIA
{
    public partial class Form1 : Form
    {
        WIA.DeviceManager dm;
```

Um die PIA-Klassen einbinden zu können, definieren wir einige wichtige Konstanten direkt, andernfalls müssten wir die externe PIA mitgeben:

```
        const string wiaEventDeviceConnected = "{A28BBADE-64B6-11D2-A231-00C04FA31809}";
        const string wiaEventDeviceDisconnected = "{143E4E83-6497-11D2-A231-00C04FA31809}";
        const string wiaFormatBMP = "{B96B3CAB-0728-11D3-9D7B-0000F81EF32E}";
        const string wiaFormatPNG = "{B96B3CAF-0728-11D3-9D7B-0000F81EF32E}";
        const string wiaFormatGIF = "{B96B3CB0-0728-11D3-9D7B-0000F81EF32E}";
```

```
const string wiaFormatJPEG = "{B96B3CAE-0728-11D3-9D7B-0000F81EF32E}";
const string wiaFormatTIFF = "{B96B3CB1-0728-11D3-9D7B-0000F81EF32E}";
```

Mit dem Laden des Formulars wird zunächst der *DeviceMananger* instanziiert, nachfolgend können zwei wichtige Ereignisse angemeldet werden:

```
private void Form1_Load(object sender, EventArgs e)
{
    dm = new DeviceManager();
    dm.RegisterEvent(wiaEventDeviceConnected, "*");
    dm.RegisterEvent(wiaEventDeviceDisconnected, "*");
    dm.OnEvent += new _IDeviceManagerEvents_OnEventEventHandler(dm_OnEvent);
    anzeige();
}
```

HINWEIS: Die Verwendung von *RegisterEvent* ist unbedingt erforderlich, andernfalls wird das betreffende Ereignis nicht an die Ereignismethode weitergeleitet.

Reagieren auf das Verbinden/Trennen von Geräten

Nutzen Sie die folgende Ereignisbehandlung, um mit Ihrer Anwendung auf das Hinzufügen bzw. Entfernen von WIA-Geräten zu reagieren:

```
void dm_OnEvent(string EventID, string DeviceID, string ItemID)
{
    if (EventID == wiaEventDeviceDisconnected) listBox1.Items.Add("Gerät getrennt");
    if (EventID == wiaEventDeviceConnected) listBox1.Items.Add("Gerät angeschlossen");
    anzeige();
}
```

HINWEIS: Nur angemeldete Ereignisse werden ausgelöst!

Ermitteln der verfügbaren Geräte

Die folgende Routine aktualisiert die Anzeige der vorhandenen Geräte. Dazu werden alle vom *DeviceManager* in der Collection *DeviceInfos* aufgelisteten Geräte abgefragt und mit *Connect* instanziiert:

```
private void anzeige()
{
    WIA.Device dev;
    listBox2.Items.Clear();
    foreach (WIA.DeviceInfo di in dm.DeviceInfos)
    {
        dev = di.Connect();
```

Konnte eine Instanz gebildet werden, wird der Name des Geräts in die Liste eingetragen:

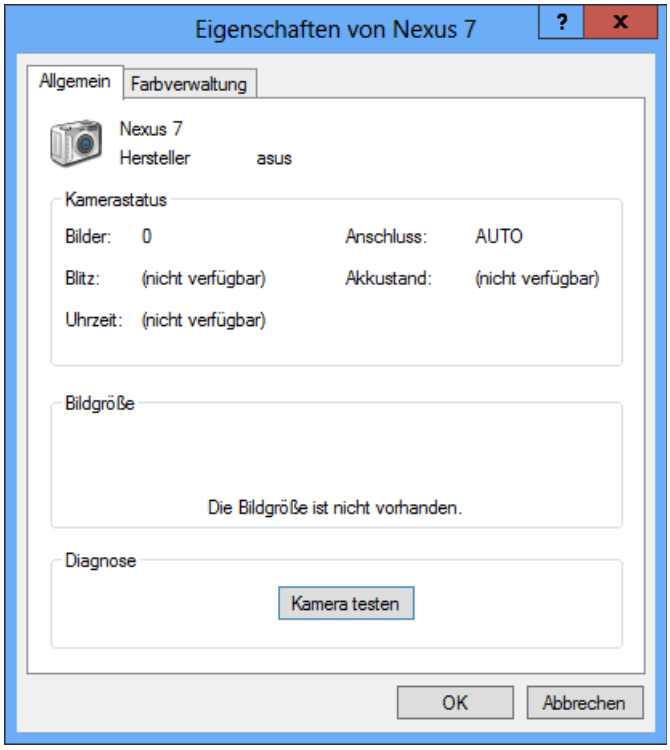
```
                if (dev != null) listBox2.Items.Add(di.Properties["Name"].get_Value());
            }
```

Die Schaltflächen unserer Anwendung werden nur freigegeben, wenn mindestens ein Gerät installiert ist:

```
            button1.Enabled = listBox2.Items.Count > 0;
            button2.Enabled = listBox2.Items.Count > 0;
            button3.Enabled = listBox2.Items.Count > 0;
            button4.Enabled = listBox2.Items.Count > 0;
            button5.Enabled = listBox2.Items.Count > 0;
            if (listBox2.Items.Count > 0) listBox2.SelectedIndex = 0;
        }
```

Anzeige der Geräteeigenschaften

Erfreulicherweise stellt die WIA-Library bereits einen großen Fundus an Dialogen bereit. Unter anderem befindet sich darunter auch ein Dialog zur Anzeige der Geräteeigenschaften, den Sie mit einem *CommonDialog*-Objekt (nicht mit den entsprechenden Dateidialogen verwechseln) und der Methode *ShowDeviceProperties* anzeigen können:



```
        private void button1_Click(object sender, EventArgs e)
        {
            WIA.Device dev;
            WIA.CommonDialog dlg = new WIA.CommonDialog();
```

Zunächst wird eine *Device*-Instanz erzeugt (ist dem physischen Gerät zugeordnet):

```
            dev = dm.DeviceInfos[listBox2.SelectedIndex + 1].Connect();
            dlg.ShowDeviceProperties(dev);
        }
```

Alternativ können Sie auch einzelne Eigenschaften über die *Properties*-Auflistung abrufen:

BEISPIEL: Ausgabe der Properties in einer *ListBox*

```
            WIA.Device dev;
            dev = dm.DeviceInfos[listBox2.SelectedIndex + 1].Connect();
            foreach (WIA.Property prop in dev.Properties)
            {
                listBox1.Items.Add(prop.Name.ToString() + ": " + prop.get_Value());
            }
```

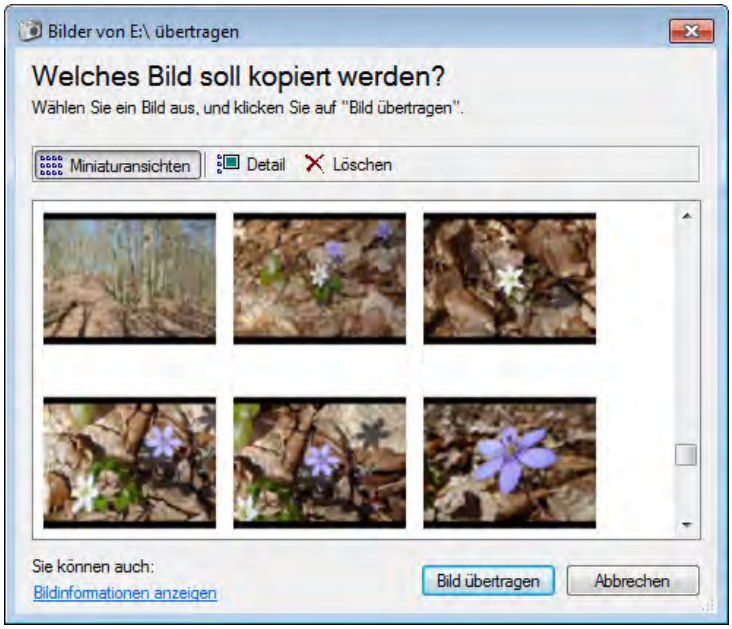
Die Ausgabe:

```
Item Name: Root
Full Item Name: 0018\Root
Item Flags: 76
Unique Device ID: {EEC5AD98-8080-425f-922A-DABF3[
Manufacturer: MATSHITA
Description: E:\
Type: 131072
Port: AUTO
Name: E:\
Server: local
Remote Device ID:
UI Class ID: {00000000-0000-0000-0000-000000000000}
```

Ein Bild einlesen

Das eigentliche Ziel unserer Bemühungen war und ist immer noch das Einlesen eines Bildes, was wir jetzt in Angriff nehmen.

Wie schon beim vorhergehenden Anzeigen von Eigenschaften können wir auch hier auf einen integrierten Dialog zugreifen, mit dem der Anwender das eigentliche Bild auswählt bzw. die Eigenschaften einstellt (siehe folgende Abbildung).



Anzeige des Dialogs:

```
private void button2_Click(object sender, EventArgs e)
{
    WIA.CommonDialog dlg = new WIA.CommonDialog();
    WIA.ImageFile img;
    img = dlg.ShowAcquireImage();
```

Der Rückgabewert ist ein *ImageFile*-Objekt, dessen Dateiformat Sie über die *FileExtension*-Eigenschaft in Erfahrung bringen können.

Für die weitere Vorgehensweise bieten sich zwei Varianten an:

- Sie speichern das Bild als Datei und laden diese gegebenenfalls später.
- Sie übertragen direkt die binären Bilddaten an Ihre Anwendung und nutzen diese beispielsweise für eine *PictureBox*.

An dieser Stelle beschränken wir uns zunächst auf die erste Variante:

```
        string TempFileName = Path.Combine(Path.GetTempPath(), Path.GetRandomFileName()) +
                              "." + img.FileExtension;
        img.SaveFile(TempFileName);
        using (FileStream fs = new FileStream(TempFileName, FileMode.Open, FileAccess.Read))
        {
            pictureBox1.Image = Image.FromStream(fs);
            fs.Close();
        }
        File.Delete(TempFileName);
    }
```

HINWEIS: Die Verwendung der *Image.FromFile*-Methode wäre zwar mit weniger Schreibaufwand verbunden, allerdings haben Sie nachfolgend keinen Zugriff auf die Datei und somit können Sie diese temporäre Datei auch nicht löschen.

Alternativ hier der Weg über einen *MemoryStream*:

```
WIA.CommonDialog dlg = new WIA.CommonDialog();
WIA.ImageFile img;
img = dlg.ShowAcquireImage();
```

Hier ermitteln wir die Bilddaten,

```
WIA.Vector v = img.FileData;
```

kopieren diese in ein Byte-Array

```
byte[] bytes = (byte[])v.get_BinaryData();
```

und wandeln dieses in einen *MemoryStream* um:

```
MemoryStream ms = new MemoryStream(bytes);
```

Letzter Schritt ist die Anzeige des Bildes:

```
pictureBox1.Image = Image.FromStream(ms);
```

Haben Sie statt einer Digitalkamera einen Scanner angeschlossen, erscheint der bekannte Scan-Dialog von Windows[1].

[1] Unsere treuen Stammleser werden sich vermutlich noch an unser allererstes Buch (siehe Abbildung) von 1995 erinnern.

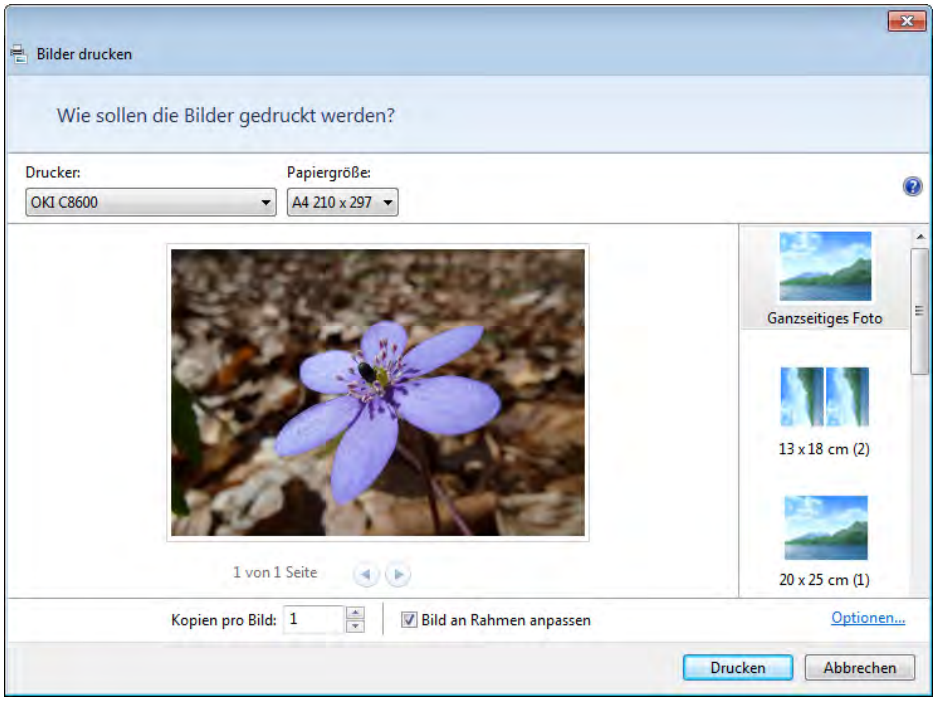
HINWEIS: In diesem Dialog können Sie auch spezielle Einstellungen (Größe, DPI, Farbe) vornehmen.

Doch Vorsicht:

HINWEIS: Gerade hier sollten Sie auf den Datentransfer per Arbeitsspeicher verzichten, können doch die Bilder sehr groß werden..

Bild(er) drucken per Assistent

Neben der reinen Anzeige können Sie per WIA auch gleich die Ausgabe realisieren, ohne sich große Gedanken um ein ansprechendes Interface machen zu müssen. Ein entsprechender Assistent steht bereits "ab Werk" zur Verfügung:




```
private void button3_Click(object sender, EventArgs e)
{
    WIA.CommonDialog dlg = new WIA.CommonDialog();
```

Wird ein Bild per *OpenFileDialog* ausgewählt:

```
            if (openFileDialog1.ShowDialog() == System.Windows.Forms.DialogResult.OK)
            {
```

... zeigen wir den Windows-Druckassistenten an:

```
                dlg.ShowPhotoPrintingWizard(openFileDialog1.FileName);
            } else
               MessageBox.Show("Kein Bild gewählt!");
        }
```

Den Scanner-Assistent aufrufen

Nicht genug der Assistenten, auch für das Scannen in eine Datei steht ein entsprechender Assistent bereit:

```
        private void button4_Click(object sender, EventArgs e)
        {
            WIA.Device dev;
            WIA.CommonDialog dlg = new WIA.CommonDialog();
            dev = dm.DeviceInfos[listBox2.SelectedIndex + 1].Connect();
```

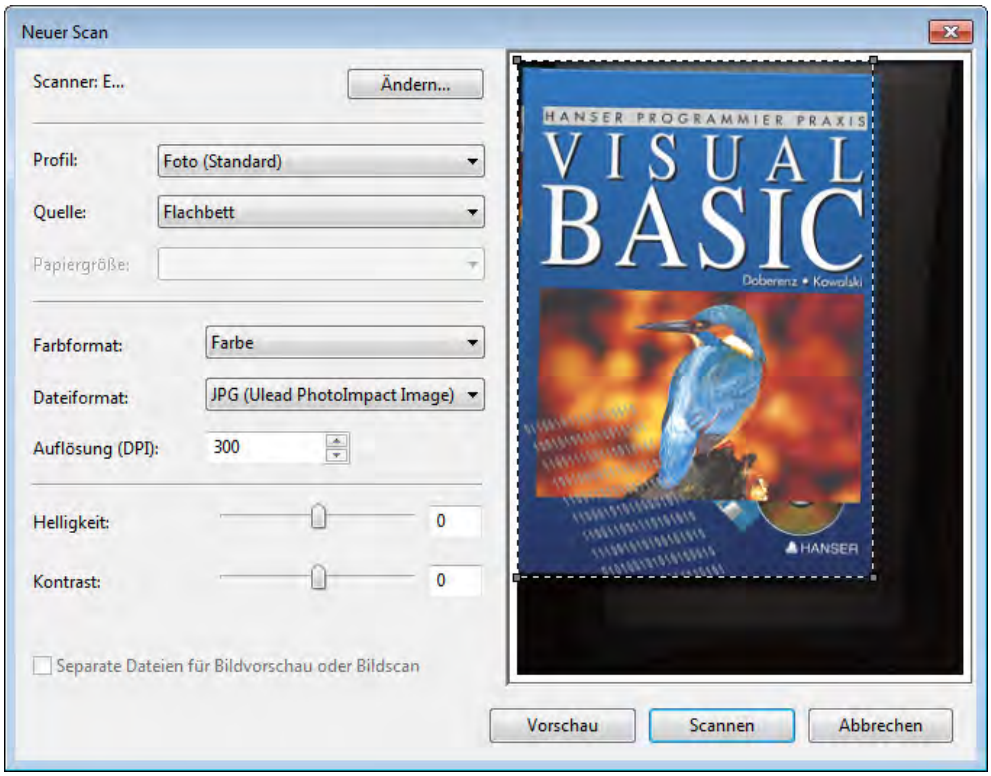
Nur wenn es sich um einen Scanner handelt, wird der Dialog angezeigt:

```
            if (dev.Type == WIA.WiaDeviceType.ScannerDeviceType)
                dlg.ShowAcquisitionWizard(dev);
            else
                MessageBox.Show("Kein Scanner vohanden!");
         }
```

Wie Sie sehen, können Sie per *Device.Type*-Eigenschaft den Gerätetyp in Erfahrung bringen:

Konstante	Bedeutung
ScannerDeviceType	Scanner, nur diese können Sie auch mit dem Scanner-Assistenten verwenden.
CameraDeviceType	Digitale Kameras die per WIA eingebunden sind. Achtung, hier haben Sie es meist mit Verzeichnisstrukturen zu tun!
VideoDeviceType	WebCam etc.
UnspecifiedDeviceType	Unbekannt, wird meist bei der Auswahl von Geräten angegeben, um alle Geräte abzufragen.

Der Scandialog unterscheidet sich in diesem Fall etwas vom "ShowAcquireImage"-Dialog:



Mehr dazu im Rezept

► R193 Auf den Scanner zugreifen

Grafikbearbeitung mit WIA

Neben dem reinen Import bietet sich WIA auch für die Bearbeitung von Bildern an. Nachdem diese in einem *ImageFile*-Objekt vorliegen, können diese mit dem *ImageProcess*-Objekt verarbeitet werden. Dazu erstellen Sie zunächst eine Instanz des *ImageProcess*-Objekts und weisen per *Filters.Add*-Methode spezielle Verarbeitungsfilter zu. Anschließend können Sie diese noch konfigurieren.

BEISPIEL: Konvertieren einer eingescannten Grafik (ins TIFF-Format)

```
WIA.CommonDialog dlg = new WIA.CommonDialog();
WIA.ImageFile img = new ImageFile();
WIA.ImageProcess proc;
if (openFileDialog1.ShowDialog() == System.Windows.Forms.DialogResult.OK)
{
```

Bild einlesen:

```
            img.LoadFile(openFileDialog1.FileName);
```

Konverter erzeugen:

```
            proc = new WIA.ImageProcess();
```

Filter zuweisen:

```
            proc.Filters.Add(proc.FilterInfos["Convert"].FilterID);
```

Filter konfigurieren

```
            proc.Filters[1].Properties["FormatID"].set_Value(wiaFormatTIFF);
```

Filter anwenden:

```
            img = proc.Apply(img);
            img.SaveFile(@"c:\test.tiff");
        }
    }
```

BEISPIEL: Verfügbare Filter anzeigen

```
    private void button7_Click(object sender, EventArgs e)
    {
        WIA.ImageProcess proc = new WIA.ImageProcess();
       foreach (WIA.FilterInfo fil in proc.FilterInfos)
           listBox1.Items.Add(fil.Name);
    }
```

Namen der verfügbaren Filter:

Natürlich können Sie mit den WIA-Objekten noch weit mehr anfangen, aber für einen ersten Einstieg dürften die bisherigen Ausführungen sicherlich genügen.

R192 Auf eine Webcam zugreifen

Haben Sie schon einmal daran gedacht, Bilder Ihrer Webcam per C#-Programm abzurufen und gegebenenfalls auf einem Server abzulegen? Wenn ja, finden Sie mit Hilfe der Microsoft WIA-Library eine recht einfache Lösung.

Zu den Grundlagen der WIA siehe Rezept

► R191 Die WIA-Library kennenlernen

Oberfläche

Ein einfaches Windows-Form mit einer *PictureBox* und einem *Timer,* den Sie bitte auf einen nicht zu kleinen Wert einstellen (z.B. 10000 für 10 Sekunden).

HINWEIS: Binden Sie zusätzlich die "Microsoft Windows Image Acquisition Library v2.0" in Ihr C#-Projekt ein (*Projekt|Verweis hinzufügen|COM*).

Quelltext

Binden Sie für die Stream-Unterstützung den folgenden Namespace ein:

```
using System.IO;
...
    public partial class Form1 : Form
    {
```

Die Konstante für die Aufnahme von Bildern:

```
        const string wiaCommandTakePicture = "{AF933CAC-ACAD-11D2-A093-00C04F72DC3C}";
```

Eine Referenz für das WIA-Gerät, in unserem Fall eine Webcam:

```
        WIA.Device Camera;
...
```

Mit dem Laden des Formulars wählen wir zunächst das gewünschte Gerät aus. Sind mehrere Geräte vorhanden, wird ein Auswahldialog angezeigt, sonst wird automatisch das erste Gerät gewählt:

```
        private void Form1_Load(object sender, EventArgs e)
            try
            {
                Camera = (new WIA.CommonDialog()).ShowSelectDevice();
```

Testen auf VideoDevice:

```
                if (Camera.Type == WIA.WiaDeviceType.VideoDeviceType)
                    timer1.Enabled = true;
                else
                {
                    MessageBox.Show("Keine Webcam vorhanden!");
                    Application.Exit();
                }
            }
            catch (Exception)
            {
```

```
                MessageBox.Show("Keine Webcam vorhanden!");
                Application.Exit();
            }
        }
```

Ist die Zeit abgelaufen, wird der Schnappschuss ausgelöst:

```
        private void timer1_Tick(object sender, EventArgs e)
        {
            timer1.Enabled = false;
            this.Text = "ACHTUNG: AUFNAHME";
```

Ein Kommando an das Gerät senden:

```
            WIA.Item item = Camera.ExecuteCommand(wiaCommandTakePicture);
```

Das erzeugte Bild abrufen und per Speicher an die *PictureBox* weiterleiten:

```
            WIA.ImageFile img = item.Transfer();
            if (img != null)
            {
                WIA.Vector vector = img.FileData;
                pictureBox1.Image = Image.FromStream(new
                                    MemoryStream((byte[])vector.get_BinaryData()));
            }
            this.Text = "-";
            timer1.Enabled = true;
        }
```

Test

Nach dem Start müssen Sie sich einige Sekunden gedulden, dann sollte auch bei Ihnen ein Bild im Fenster erscheinen:

R193 Auf den Scanner zugreifen

Nachdem wir uns im vorhergehenden Rezept schon etwas mit der WIA beschäftigt haben, wollen wir uns im Rahmen dieses Beispiels noch einmal intensiver mit dem Zugriff auf Scanner befassen.

Zwei grundsätzliche Varianten sind bei Verwendung der WIA möglich:

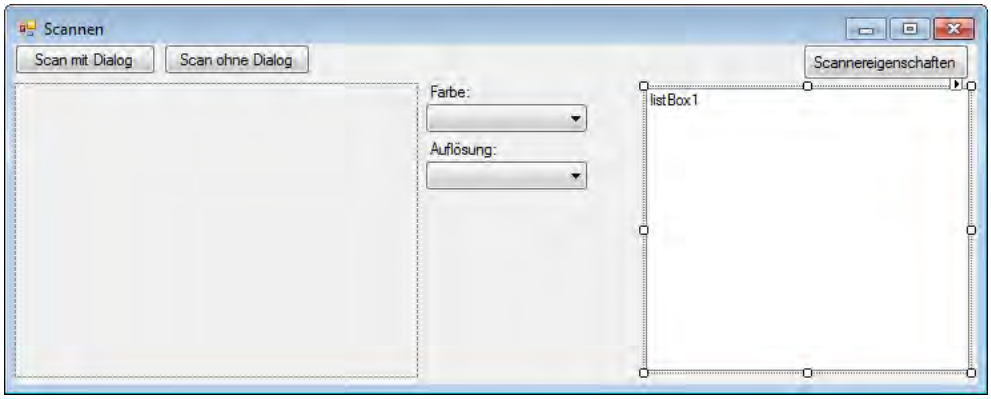
- Anzeige des Scanner-Assistenten und Übernahme des Bildes mit den dort getroffenen Einstellungen.
- Setzen der Scanner-Parameter mit eigenen Dialogen/Komponenten und direktem Abruf des Bildes.

Wie Sie es sich vermutlich denken können, ist die zweite Variante mit wesentlich mehr Programmieraufwand verbunden, müssen Sie sich doch recht intensiv mit den einzelnen Scanner-Parametern auseinandersetzen.

Im Weiteren wollen wir Ihnen beide Versionen vorstellen, zusätzlich beschäftigen wir uns noch mit dem Abruf der Standardeinstellungen.

Oberfläche

Wir beschränken uns auf eine *PictureBox* zur Anzeige, eine *ListBox* für die Scanner-Eigenschaften sowie zwei *ComboBox*en für die Auswahl der Farbeinstellungen und der DPI-Zahl. Last, but not least, sind auch noch drei Schaltflächen nötig:



Weisen Sie der Farb-*ComboBox* die Werte "Schwarz/Weiss", "Graustufe" und "Farbe" zu, die *ComboBox* für die DPI-Zahl konfigurieren Sie mit "75, 100, 150, 200, 300, 600".

HINWEIS: Binden Sie zusätzlich die "Microsoft Windows Image Acquisition Library v2.0" in Ihr C#-Projekt ein (*Projekt|Verweis hinzufügen|COM*).

Quellcode

```
using System.IO;
...
    public partial class Form1 : Form
    {
```

Einige Konstanten für die Auswahl des Bildformats:

```
        const string wiaFormatBMP = "{B96B3CAB-0728-11D3-9D7B-0000F81EF32E}";
        const string wiaFormatPNG = "{B96B3CAF-0728-11D3-9D7B-0000F81EF32E}";
        const string wiaFormatGIF = "{B96B3CB0-0728-11D3-9D7B-0000F81EF32E}";
        const string wiaFormatJPEG = "{B96B3CAE-0728-11D3-9D7B-0000F81EF32E}";
        const string wiaFormatTIFF = "{B96B3CB1-0728-11D3-9D7B-0000F81EF32E}";
...
```

Zunächst beschäftigen wir uns mit der Anzeige des Scannerdialogs:

```
        private void button1_Click(object sender, EventArgs e)
        {
            WIA.CommonDialog dlg = new WIA.CommonDialog();
            WIA.ImageFile img = null;
            img = dlg.ShowAcquireImage(WIA.WiaDeviceType.ScannerDeviceType,
                                       WIA.WiaImageIntent.GrayscaleIntent,
                                       WIA.WiaImageBias.MinimizeSize,
                                       wiaFormatJPEG, false, false, false);
```

HINWEIS: Machen Sie sich keine Illusionen, es ist (fast) egal, was Sie dem Dialog übergeben, es wird immer die Standardansicht angezeigt.

Wurde ein Bild zurückgegeben, müssen wir dieses an unsere Anwendung weiterreichen:

```
            if (img != null)
            {
                WIA.Vector vector = img.FileData;
                pictureBox1.Image = Image.FromStream(new
                          MemoryStream((byte[])vector.get_BinaryData()));
            }
        }
```

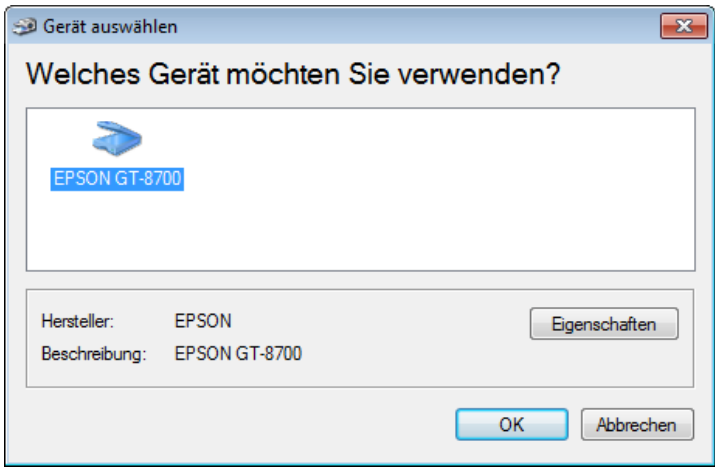
Die etwas flexiblere Variante für den Zugriff auf den Scanner nutzt die WIA-Dialoge lediglich zur Auswahl eines Gerätes. nachfolgend wird das Gerät konfiguriert und es wird das gewünschte Bild abgerufen. Doch der Reihe nach ...

```
        private void button2_Click(object sender, EventArgs e)
        {
```

Ein Gerät abrufen:

```
            WIA.CommonDialog dlg = new WIA.CommonDialog();
            WIA.Device Scanner = dlg.ShowSelectDevice(WIA.WiaDeviceType.ScannerDeviceType,
                                                      false, false);
```

Der angezeigte Dialog:



Aus der zweiten *ComboBox* lesen wir die DPI-Zahl ab:

```
int dpi = Convert.ToInt16(comboBox2.SelectedItem);
```

Je nach Auswahl in der ersten *ComboBox* weisen wir die Farbauswahl zu:

```
switch (comboBox1.SelectedIndex)
{
    case 0:
        Scanner.Items[1].Properties["Current Intent"].set_Value(4);
        break;
    case 1:
        Scanner.Items[1].Properties["Current Intent"].set_Value(2);
        break;
    default:
        Scanner.Items[1].Properties["Current Intent"].set_Value(1);
    break;
}
```

Übergeben wird diese Information über die *Properties*-Collection des ersten Items. Gleiches trifft auch auf die anderen Scan-Optionen zu:

Dots per inch/Horizontal:

```
Scanner.Items[1].Properties["Horizontal Resolution"].set_Value(dpi);
```

Dots per inch/Vertikal:

```
Scanner.Items[1].Properties["Vertical Resolution"].set_Value(dpi);
```

Linke obere Ecke für den Scanbereich:

```
Scanner.Items[1].Properties["Horizontal Start Position"].set_Value(0);
Scanner.Items[1].Properties["Vertical Start Position"].set_Value(0);
```

Größe des Scanbereichs (Inch * dpi):

```
Scanner.Items[1].Properties["Horizontal Extent"].set_Value(8.5 * dpi);
Scanner.Items[1].Properties["Vertical Extent"].set_Value(11 * dpi);
```

Übertragen des Bildes in ein *ImageFile*-Objekt:

```
WIA.ImageFile img = Scanner.Items[1].Transfer();
```

War dies erfolgreich, müssen wir das *ImageFile*-Objekt "nur noch" in unserer *PictureBox* anzeigen:

```
        if (img != null)
        {
            WIA.Vector vector = img.FileData;
            pictureBox1.Image = Image.FromStream(new
                                    MemoryStream((byte[])vector.get_BinaryData()));
        }
    }
```

Ganz nebenbei zeigen wir in der *ListBox* noch die möglichen Eigenschaften des Scanners an:

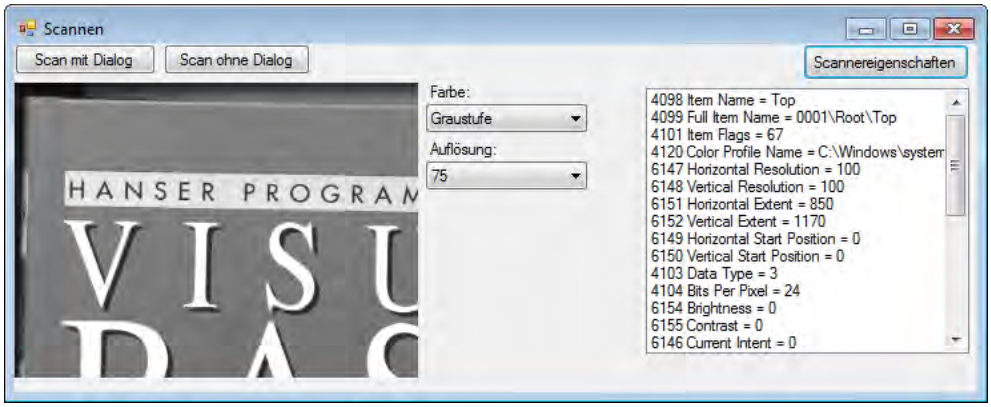
```
    private void button4_Click(object sender, EventArgs e)
    {
        WIA.CommonDialog dlg = new WIA.CommonDialog();
        WIA.Device Scanner = dlg.ShowSelectDevice(WIA.WiaDeviceType.ScannerDeviceType,
                             false, false);

        foreach (dynamic p in Scanner.Items[1].Properties)
        {
            listBox1.Items.Add(p.PropertyID +" " + p.Name + " = " + p.Value);
        }
    }
```

Das mögliche Ergebnis (geräteabhängig):

```
4101 Item Flags = 67
4120 Color Profile Name = C:\Windows\system
6147 Horizontal Resolution = 100
6148 Vertical Resolution = 100
6151 Horizontal Extent = 850
6152 Vertical Extent = 1170
6149 Horizontal Start Position = 0
6150 Vertical Start Position = 0
4103 Data Type = 3
4104 Bits Per Pixel = 24
6154 Brightness = 0
6155 Contrast = 0
6146 Current Intent = 0
4112 Pixels Per Line = 850
4114 Number of Lines = 1170
4105 Preferred Format = {B96B3CAA-0728-11D
```

Test

Nach dem Start können Sie beide Scan-Varianten ausprobieren. Experimentieren Sie ruhig auch einmal mit den Properties des *Item*-Objekts um andere Auflösungen oder Scanbereiche zu bestimmen.



Bemerkung

Die Parameter für *ShowAcquireImage*:

```
SYNTAX:  ImageFile ShowAcquireImage([DeviceType], [Intent], [Bias], [FormatID],
                                    [AlwaysSelectDevice], [UseCommonUI], [CancelError])
```

Die Bedeutung:

Parameter	Beschreibung
DeviceType	Werte der *WiaDeviceType*-Enumeration, Default = *UnspecifiedDeviceType*, für Scanner nutzen Sie bitte *ScannerDeviceType*.
Intent	Werte der *WiaImageIntent*-Enumeration (Wert egal)
Bias	Werte der *WiaImageBias*-Enumeration (Wert egal)
FormatID	Das zurückgegebene Bildformat: wiaFormatBMP = "{B96B3CAB-0728-11D3-9D7B-0000F81EF32E}" wiaFormatPNG = "{B96B3CAF-0728-11D3-9D7B-0000F81EF32E}" wiaFormatGIF = "{B96B3CB0-0728-11D3-9D7B-0000F81EF32E}" wiaFormatJPEG = "{B96B3CAE-0728-11D3-9D7B-0000F81EF32E}" wiaFormatTIFF = "{B96B3CB1-0728-11D3-9D7B-0000F81EF32E}"
AlwaysSelectDevice	Soll das Gerät jedesmal ausgewählt werden?
UseCommonUI	Soll der Dialog angezeigt werden (das ist immer der Fall)?
CancelError	Soll ein Fehler ausgelöst werden, wenn der Nutzer Abbruch wählt?

R194 OpenOffice.org Writer per OLE steuern

Angesichts der reichlich vorhandenen Lösungen zur MS Word-Automatisierung entsteht für manchen Programmierer sicher auch die Frage, wie sich Open*Office.org Writer*, der Freeware-Pendant von Word, automatisieren lässt. Das vorliegende Beispiel soll einen ersten Ansatz aufzeigen.

Leider ist die Programmierung von OpenOffice-Anwendungen per C#/Visual Basic nicht gerade trivial, eine fast schon chaotisch wirkende Programmierschnittstelle macht es dem eingefleischten MS-Programmierer nicht gerade leicht.

Wir wollen Ihnen etwas Arbeit abnehmen und kapseln deshalb einige der wichtigsten Grundfunktionen in einer eigenen Klasse *OOWriter*.[1] Mit dieser lassen sich dann die Standardaufgaben relativ leicht lösen.

Ärger mit den dynamischen Objekten

Wer sich jetzt, wie die Autoren, voller Elan auf die Umsetzung stürzt, wird sich bald in einem undurchdringlichen Gestrüpp aus Invoke-Aufrufen, Parameterübergaben etc. wiederfinden.

Auch der neue *dynamic*-Datentyp ist hier keine echte Hilfe (zugewiesene Objekte haben keine der gewünschten Methoden etc.).

Nach diversen Experimenten beim Erstellen der notwendigsten Klassen haben wir die Arbeit abgebrochen und uns einer "echten" dynamischen Programmiersprache zugewendet. Sie ahnen es vielleicht, wir sprechen von Visual Basic. Doch das soll Sie als C#-Programmierer nicht daran hindern, eine mit VB erstellte Library in Ihr Projekt einzubinden, um damit in den Genuss von echtem Late-Binding zu kommen.

HINWEIS: Aus diesem Grund wundern Sie sich bitte nicht, dass an dieser Stelle zunächst die VB-Library erstellt und beschrieben wird, bevor wir zu unserer C#-Beispielanwendung kommen.

VB-Library "OOWLib"

Herzlich willkommen in der Welt der VB-Programmierer!

Library erstellen

Erstellen Sie trotz VB zunächst ein C#-Windows Forms-Projekt und fügen Sie in die übergeordnete Solution eine ***VB-Klassenbibliothek*** mit dem Namen *OOWLib* ein (Kontextmenü *Hinzufügen neues Projekt*).

[1] Sollte ein Leser den Elan aufbringen, die Klasse wesentlich zu erweitern, sind die Autoren für jede Art von Feedback dankbar.

Löschen Sie den schon vorhandenen Klassenrumpf und erstellen Sie eine neue Klasse *cOOWriter*, auf deren Details wir im Folgenden eingehen wollen.

Quelltext (Klasse cOOWriter)

Unsere neue VB-Klasse:

```
Public Class cOOWriter
```

Einige interne Objekte[1]:

```
    Private oStarOffice As Object
    Private oDesk As Object
    Private oDoc As Object
    Private oVC As Object
```

Wir erstellen ein neues, leeres Dokument:

```
    Public Sub NewDocument()
        Dim args(-1) As Object
        oDesk = oStarOffice.createInstance("com.sun.star.frame.Desktop")
```

Das aktuelle Dokument:

```
        oDoc = oDesk.LoadComponentFromURL("private:factory/swriter", "_blank", 0, args)
```

Der sichtbare Cursor[2]:

```
        oVC = oDoc.CurrentController.ViewCursor()
    End Sub
```

Alternativ greifen wir auf eine vorhandene Datei zu:

```
    Public Sub OpenDocument(ByVal doc As String, Optional ByVal ro As Boolean = False,
                            Optional ByVal Passwort As String = "")
        Dim args(2) As Object
        args(0) = MakePropertyValue("ReadOnly", ro)
        args(1) = MakePropertyValue("Password", Passwort)
        args(2) = MakePropertyValue("Hidden", False)
        oDesk = oStarOffice.createInstance("com.sun.star.frame.Desktop")
        oDoc = oDesk.LoadComponentFromURL(ConvertToUrl(doc), "_blank", 0, args)
        oVC = oDoc.CurrentController.ViewCursor()
    End Sub
```

HINWEIS: Sie müssen eine der beiden o.g. Methoden ausführen, bevor Sie das Dokument bearbeiten können.

[1] Wer in der Star-Basic-Programmierung versiert ist, kann diese Objekte auch für den externen Zugriff freigeben und nutzen.

[2] Writer kennt auch noch Textcursor, die unabhängig vom sichtbaren Cursor agieren können.

Ausgabe von Text:

```
Public Sub InsertString(ByVal str As String)
    oDoc.Text.InsertString(oVC, str, False)
End Sub
```

Ein Zeilenumbruch:

```
Public Sub NewLine()
    oDoc.Text.InsertControlCharacter(oVC, 0, False)
End Sub
```

Ein neuer Absatz:

```
Public Sub NewParagraph()
    oDoc.Text.InsertControlCharacter(oVC, 5, False)
End Sub
```

Ein Seitenumbruch:

```
Public Sub PageBreak()
    oDoc.Text.InsertControlCharacter(oVC, 0, False)
    oVC.BreakType = 4
End Sub
```

Eine Dokumentvorlage nutzen:

```
Public Sub SetStyle(ByVal StyleName As String)
    oVC.ParaStylename = StyleName
End Sub
```

Fettschrift:

```
Public Property Bold() As Boolean
    Get
        Return (oVC.CharWeight = 150)
    End Get
    Set(ByVal value As Boolean)
        If value Then
            oVC.CharWeight = 150
        Else
            oVC.CharWeight = 100
        End If
    End Set
End Property
```

Eine Schriftart auswählen:

```
Public Sub SetFont(ByVal Fontname As String, Optional ByVal Size As Long = 12)
    oVC.CharFontname = Fontname
    If Size > 0 Then oVC.CharHeight = Size
End Sub
```

An den Anfang des Dokuments springen:

```
Public Sub GotoStart()
    oVC.JumpToFirstPage()
End Sub
```

An das Ende des Dokuments springen:

```
Public Sub GotoEnd()
    oVC.JumpToLastPage()
    oVC.jumpToEndOfPage()
End Sub
```

Datei als PDF sichern:

```
Public Sub SaveAsPDF(ByVal filename As String)
    Dim args(1) As Object
    args(0) = MakePropertyValue("FilterName", "writer_pdf_Export")
    oDoc.storeToURL(ConvertToUrl(filename), args)
End Sub
```

Dokument schließen:

```
Public Sub CloseDocument()
    oDoc.Close(True)
End Sub
```

Die folgenden internen Funktionen stammen aus dem Internet[1] und erleichtern den Zugriff auf die OpenOffice-Objekte:

Eine Objekt-Eigenschaft erzeugen:

```
Private Function MakePropertyValue(ByVal cName, ByVal uValue) As Object
    Dim oPropertyValue As Object
    oPropertyValue = oStarOffice.Bridge_GetStruct("com.sun.star.beans.PropertyValue")
    oPropertyValue.Name = cName
    oPropertyValue.value = uValue
    MakePropertyValue = oPropertyValue
End Function
```

Pfadangaben in Urls umwandeln:

```
Private Function ConvertToUrl(ByVal strFile As String) As String
    strFile = Replace(strFile, "\", "/")
    strFile = Replace(strFile, ":", "|")
    strFile = Replace(strFile, " ", "%20")
    strFile = "file:///" + strFile
    ConvertToUrl = strFile
End Function
```

Ein spezielles Service-Objekt erzeugen:

```
Private Function CreateUnoService(ByVal strServiceName) As Object
```

[1] http://www.kalitech.fr/clients/doc/VB_APIOOo_en.html

```
        Dim oServiceManager As Object
        oServiceManager = CreateObject("com.sun.star.ServiceManager")
        CreateUnoService = oServiceManager.createInstance(strServiceName)
    End Function
```

Im Konstruktor erstellen wir das zentrale *ServiceManager*-Objekt:

```
    Public Sub New()
        oStarOffice = CreateObject("com.sun.star.ServiceManager")
    End Sub
End Class
```

Kompilieren

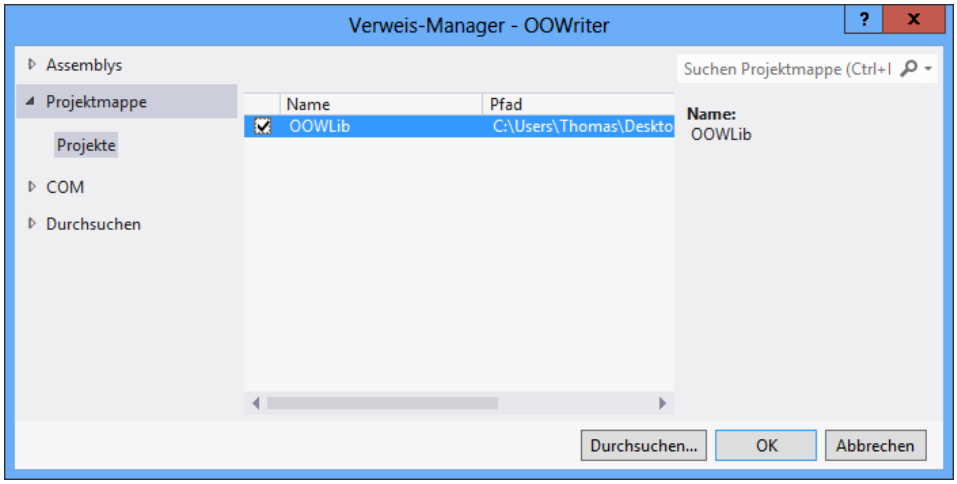
HINWEIS: Kompilieren Sie abschließend die Klassenbibliothek um die gewünschte Assembly zu erstellen.

C#-Anwendungsprogramm

Damit endet auch schon unser kurzer Ausflug in die VB-Regionen und wir kehren wieder zu unserer vertrauten Programmiersprache zurück.

Einbinden der VB-Assembly

Wählen Sie in der Solution das C#-Projekt und fügen Sie eine Referenz auf die oben erstellte VB-Klassenbibliothek hinzu (*Verweis hinzufügen|Projekte*).



Nachfolgend kommen wir schon in den Genuss der vollen Funktionalität unserer Assembly, wir müssen den Namespace nur noch importieren.

Oberfläche

Die Oberfläche beschränkt sich auf einen *Button* und einen *FileSaveDialog*.

Quelltext

Um das Beispiel nicht auf ein simples "Hello World" zu beschränken, wollen wir den Inhalt einer XML-Datei formatiert ausgeben.

HINWEIS: Die dazu nötige XML-Datei finden Sie in den Beispieldaten zum Buch.

Import der beiden benötigten Namespaces:

```
using OOWLib;           // unsere VB-Library
using System.Xml.Linq;
...
    public partial class Form1 : Form
    {
```

Eine Instanz für die Elemente der XML-Datei:

```
        private XElement ArtikelListe;

        public Form1()
        {
            InitializeComponent();
```

Hier laden wir die XML-Beispieldaten:

```
            ArtikelListe = XElement.Load("Artikel.xml");
        }
```

Mit dem Klick auf die Schaltfläche beginnen die hektischen Aktivitäten:

```
        private void button1_Click(object sender, EventArgs e)
        {
```

Eine Instanz des Writers erstellen[1]:

```
            cOOWriter oow = new cOOWriter();
```

Ein neues Dokument anlegen:

```
            oow.NewDocument();
```

Formatvorlage *Überschrift1* auswählen und Textausgabe:

```
            oow.SetStyle("Überschrift 1");
            oow.InsertString("Lagerdaten");
```

[1] Im Gegensatz zu Word/Excel etc. wird der Writer standardmäßig eingeblendet, Sie müssen diesen also nicht erst sichtbar machen.

Neuer Absatz und Formatvorlage *Standard:*

```
            oow.NewParagraph();
            oow.SetStyle("Standard");
```

Zeilenvorschub:

```
            oow.NewLine();
```

Hier folgt die Schleife für alle *Artikel*-Elemente in den XML-Daten:

```
            foreach (XElement Artikel in ArtikelListe.Elements())
            {
```

Fettschrift:

```
                oow.Bold = true;
```

Textausgabe:

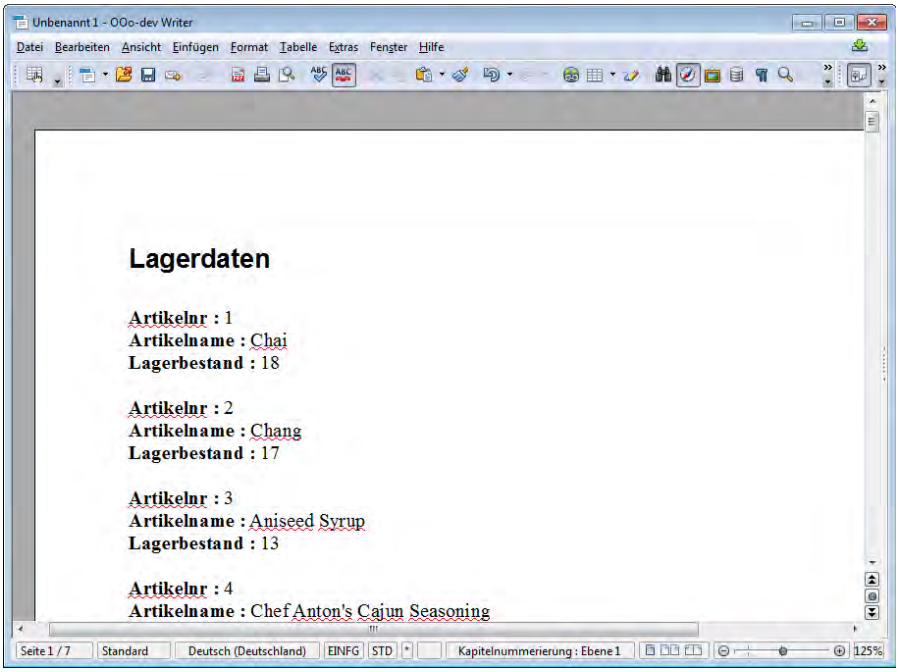
```
                oow.InsertString("Artikelnr : ");
                oow.Bold = false;
                oow.InsertString(Artikel.Element("ArtikelNr").Value);
                oow.NewLine();
                // Textausgabe:
                oow.Bold = true;
                oow.InsertString("Artikelname : ");
                oow.Bold = false;
                oow.InsertString(Artikel.Element("Artikelname").Value);
                oow.NewLine();
                // Textausgabe:
                oow.Bold = true;
                oow.InsertString("Lagerbestand : ");
                oow.Bold = false;
                oow.InsertString(Artikel.Element("Lagerbestand").Value);
                oow.NewLine();
                oow.NewLine();
            }
```

Last, but not least wollen wir das erzeugte Writer-Dokument als PDF-Datei speichern:

```
            if (saveFileDialog1.ShowDialog() == System.Windows.Forms.DialogResult.OK)
                oow.SaveAsPDF(saveFileDialog1.FileName);
            oow = null;
        }
    }
}
```

Test

Nach dem Klick auf die Schaltfläche sollten Sie folgende Ausgabe erhalten (vorausgesetzt, Sie haben *OpenOffice.org* installiert):



Auch der Export ins PDF-Format dürfte problemlos realisiert werden, Sie können sich mit dem Acrobat Reader vom Erfolg überzeugen.

Bemerkung

Das Rezept dürfte ein gutes Beispiel dafür sein, wie sinnvoll sich die MS-Programmiersprachen ergänzen und wie einfach der Datenaustausch zwischen zwei .NET-Programmiersprachen realisierbar ist.

HINWEIS: Auch wenn Sie mit obigem Beispiel recht schnell zu brauchbaren Ergebnissen kommen, viele OpenOffice-Funktionen sind nur umständlich erreichbar (z.B. Zwischenablage).

Hilfe finden Sie in diesem Fall unter:

LINK: `http://forum.openoffice.org/en/forum/`

R195 Mit OLE-Automation auf MS Access zugreifen

Da VBA nach wie vor auch für Microsoft Access als Programmiersprache eingesetzt wird, muss man leider auch in Kauf nehmen, dass die VBA-Fähigkeiten im Vergleich zu den Möglichkeiten der modernen .NET-Programmiersprachen, wie Visual Basic oder Visual C#, ziemlich einge-

schränkt sind. Insbesondere trifft dies auf verteilte Umgebungen zu, d.h., Intranet- und Internet-Applikationen lassen sich mit VBA – wenn überhaupt – nur sehr umständlich entwickeln.

Um von C# aus auf Access Datenbanken zugreifen zu können, gibt es hauptsächlich zwei Alternativen:

- Automation
- und ADO.NET.

Automation erlaubt bekanntlich die quasi "Fernsteuerung" einer Anwendung von einer anderen Anwendung aus. Sie sollten Automation dann verwenden, wenn Sie an Access-spezifischen Features interessiert sind, wie z.B. die Druck- oder Vorschau-Funktionen für Access-Reports, das Anzeigen und Manipulieren eines Access-Formulars oder der Aufruf eines Makros.

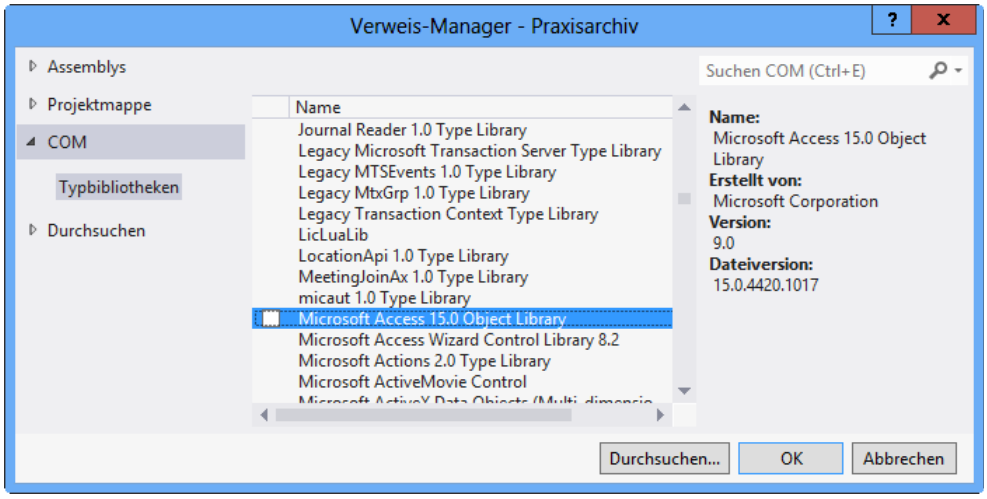
Im vorliegenden Beispiel werden wir zeigen, wie Sie auf einen Access-Report per OLE-Automation zugreifen können. Um es nicht ganz so einfach zu machen, rufen wird den Report allerdings über ein Access-Formular auf, in welches Bereichsgrenzen einzugeben sind.

HINWEIS: Für das Nachvollziehen dieses Beispiels muss Ihnen *Microsoft Access* zur Verfügung stehen!

Oberfläche

Erstellen Sie eine eine neue Windows Forms-Anwendung. Auf dem Hauptformular platzieren Sie eine Schaltfläche.

Über das Menü *Projekt|Verweis hinzufügen ...* öffnen Sie den entsprechenden Dialog, wo Sie auf der COM-Seite einen Verweis auf die *Microsoft Access 15.0 Object Library* einrichten:



Quelltext

Um unliebsamen Problemen mit dem Namespace *Microsoft.Office.Interop.Access* aus dem Weg zu gehen (Überschneidung bei Bezeichnern) vergeben wir einen Alias-Namen:

```
using ACC = Microsoft.Office.Interop.Access;
using System.Runtime.InteropServices;
...
    public partial class Form1 : Form
    {
```

Die Schaltfläche "Access aufrufen":

```
        private void button1_Click(object sender, EventArgs e)
        {
```

Unsere Hauptroutine hat die Aufgabe, die Datenbank *Test.mdb* aufzurufen und dort das Formular "Bericht aufrufen" anzuzeigen. Demonstriert wird außerdem die Manipulation von Steuerelementen:

```
            ACC.Application oAccess = null;
            ACC.Form oForm = null;
            string dbPath = null;   // Datenbankpfad zu Test.mdb
```

Eine neue Access-Instanz für die Automation starten:

```
            oAccess = new ACC.Application();
```

Dafür sorgen, dass Access sichtbar ist:

```
            oAccess.Visible = true;
```

Datenbankpfad ermitteln (die Datenbank befindet sich hier der Einfachheit wegen im Anwendungsverzeichnis!):

```
            dbPath = Application.StartupPath + "\\Test.mdb";
```

Datenbank öffnen (im Shared Mode)[1]:

```
            oAccess.OpenCurrentDatabase(filepath: dbPath, Exclusive: false);
```

Eventuell geöffnete Formulare schließen:

```
            foreach (ACC.Form oFormWithinLoop in oAccess.Forms)
            {
                oForm = oFormWithinLoop;
                oAccess.DoCmd.Close(ObjectType: ACC.AcObjectType.acForm,
                                    ObjectName: oFormWithinLoop.Name,
                                    Save: ACC.AcCloseSave.acSaveNo);
            }
```

[1] Wir verwenden benannte Parameter ...

```
if (oForm != null)
    Marshal.ReleaseComObject(oForm);
oForm = null;
```

Das Formular im Datenbankfenster auswählen und den Fokus erteilen:

```
oAccess.DoCmd.SelectObject(ObjectType: ACC.AcObjectType.acForm,
            ObjectName: "Bericht aufrufen", InDatabaseWindow: true);
```

Das Formular anzeigen und seine Beschriftung editieren:

```
oAccess.DoCmd.OpenForm(FormName: "Bericht aufrufen", View: ACC.AcFormView.acNormal);
oForm = oAccess.Forms["Bericht aufrufen"];
oForm.Caption = "Access-Automation mit C#";
```

Die Controls-Auflistung verwenden, um die Beschriftung der Schaltfläche *Befehl1* zu editieren:

```
dynamic oCtl = oForm.Controls["Befehl1"];
oCtl.Caption = "Report aufrufen";
Marshal.ReleaseComObject(oCtl);
oCtl = null;
oForm.SetFocus();
```

Das Objekt freigeben:

```
Marshal.ReleaseComObject(oForm);
oForm = null;
```

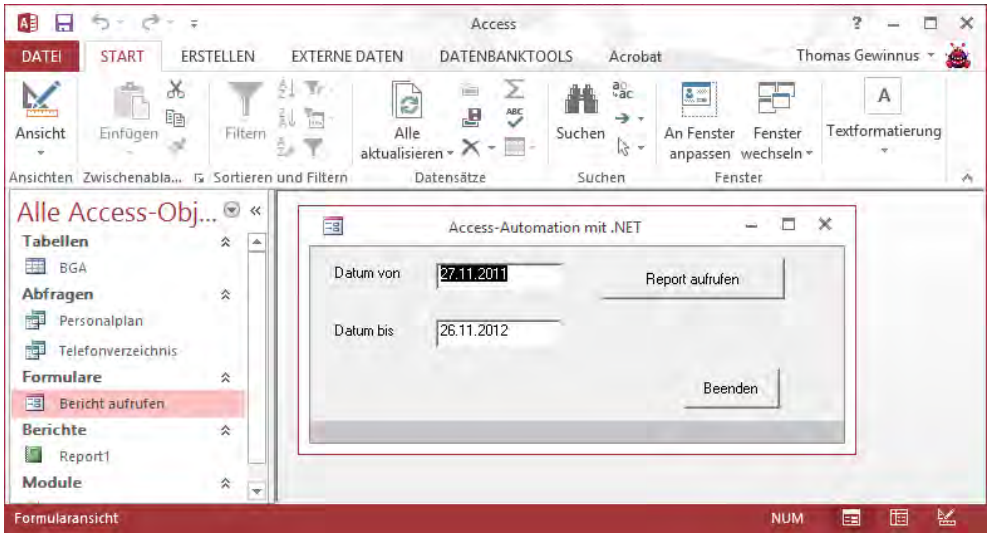
Das *Application*-Objekt freigeben und erlauben, dass Access durch den Anwender geschlossen wird:

```
if (!oAccess.UserControl)
    oAccess.UserControl = true;
Marshal.ReleaseComObject(oAccess);
oAccess = null;
```

Test

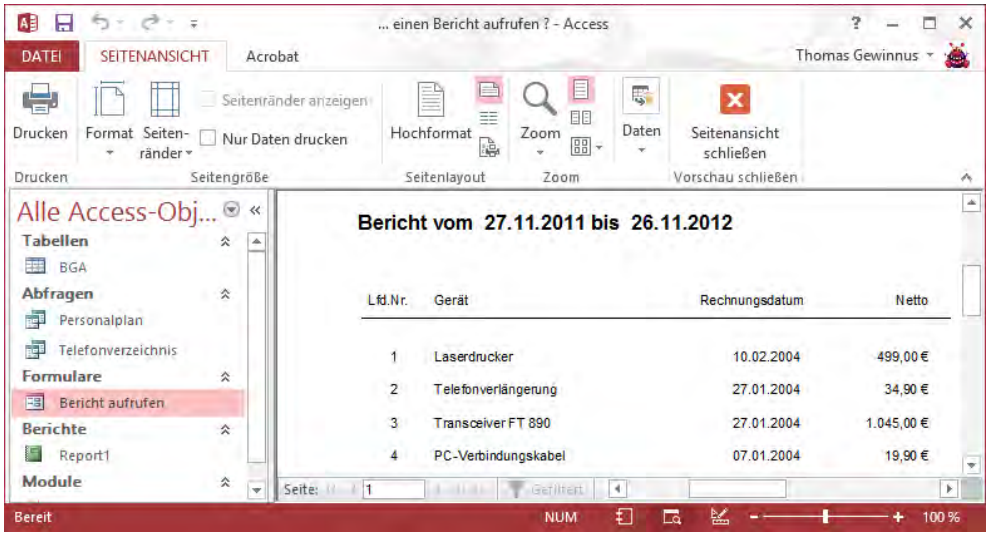
Die Datenbank *Test.mdb* (siehe Buchbeispiele) bietet keine Besonderheiten, sie besteht lediglich aus der Tabelle *BGA* und einem Formular, von welchem aus über eine Schaltfläche der Bericht *Report1* aufgerufen wird.

Nach dem Kompilieren des C#-Projekts und Betätigen der Schaltfläche *Access aufrufen* wird wie von Geisterhand Microsoft Access gestartet und es erscheint das gewünschte Formular, von welchem aus Sie dann die in Access implementierten Druck- und Vorschaufunktionen aufrufen.



Auch an den geänderten Beschriftungen von Formular und Schaltfläche sehen Sie, dass die Automation funktioniert hat.

Nach Betätigen der Schaltfläche "Report aufrufen" erscheint zunächst die Vorschau, von welcher aus Sie dann den Druck starten können:



Bemerkungen

- Natürlich wäre es auch möglich, den Druckvorgang bzw. die Druckvorschau direkt aus C# heraus zu starten. Wir haben uns jedoch für das "Dazwischenschalten" eines Access-Formulars entschieden, um den Quellcode überschaubarer zu halten.
- Auf die gleiche Weise wie hier für ein Access-Formular demonstriert, lassen sich auch alle anderen Access-Objekte (Tabellen, Berichte, Abfragen, Module) von einer beliebigen .NET-Anwendung aus über Automation quasi "fernsteuern".

R196 Ein Managed Add-In programmieren und einbinden

Sie möchten mit MS Access einige Funktionen realisieren, die mit dem boardeigenen VBA nicht realisierbar sind. Spontan fallen uns da vor allem Webdienste, Remoting, Windows Forms, WPF etc. ein.

Einen direkten Weg für den Zugriff auf obige Funktionen gibt es nicht, aber eine kleine und recht effiziente Hintertür haben die Microsoft-Entwickler offen gelassen. Die Rede ist von der Möglichkeit, Managed Add-Ins in Microsoft Access zu integrieren, die Sie zum Beispiel in C# oder VB programmieren können.

HINWEIS: Diese Add-Ins können Sie nur mit Visual Studio 2010 erstellen. In Visual Studio 2012 ist die entsprechende Unterstützung **nicht mehr vorhanden**.

Visual Studio 2010 bietet zu diesem Zweck den Projekttyp *Gemeinsames Add-In* an, Sie müssen "nur noch" eine geeignete Schnittstelle zwischen der Access-Anwendung und dem Add-In schaffen. Wer jetzt befürchtet, sich mit Unmengen an Deklarationen etc. herumschlagen zu müssen, der sei beruhigt, diese Arbeit wird von einem Assistenten weitestgehend erledigt, es bleibt wirklich nur die reine Schnittstelle übrig.

Doch welche Form der Interaktion zwischen Access-Anwendung und Add-In ist überhaupt möglich? Eigentlich gibt es da kaum Einschränkungen, nach der Übergabe einer Referenz können Sie zum Beispiel auf

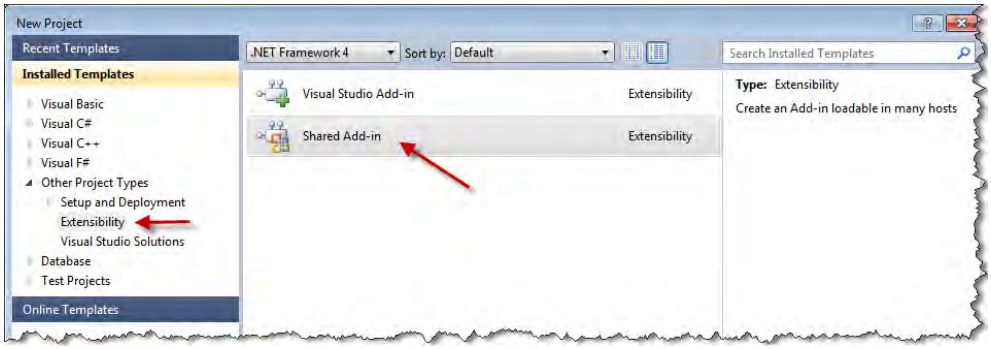
- die Anwendung (per *Application*-Objekt),
- geöffnete Formulare und Berichte (per *Me*-Referenz),
- Steuerelemente (Eigenschaften, Methoden, Ereignisse)

zugreifen.

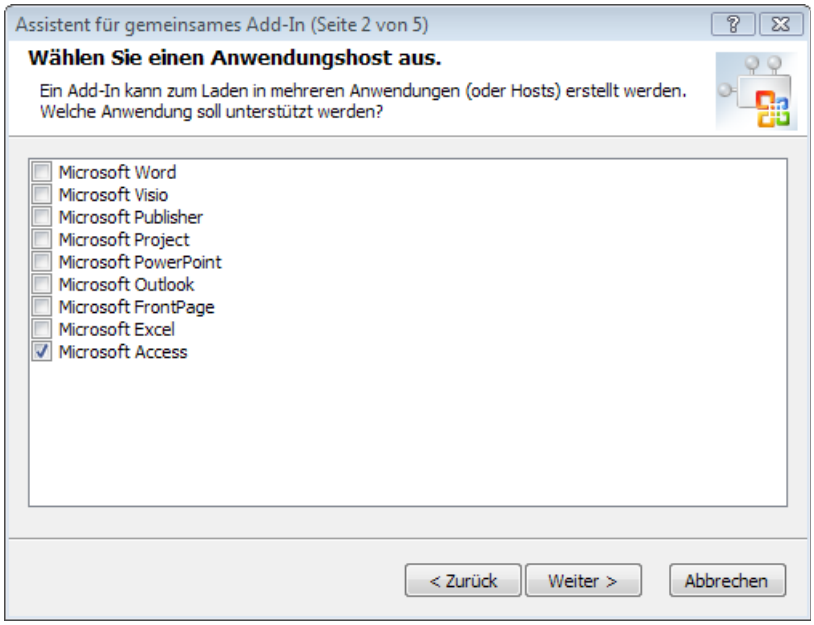
Ein kleines Beispielprogramm soll das Zusammenspiel demonstrieren. Dazu werden wir per Add-In zwei Methoden bereitstellen, eine Access-Schaltfläche konfigurieren und ein .NET-Formular zur Konfiguration eines Access-Kombinationsfeldes anzeigen. Doch der Reihe nach.

Entwurf des Add-Ins

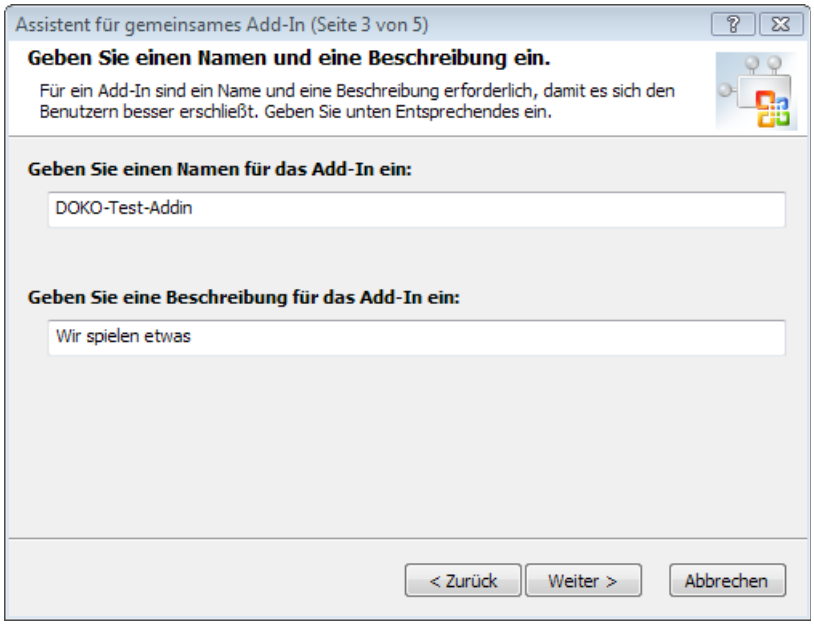
Starten Sie Visual Studio und wählen Sie *Datei|Neu|Projekt|Andere Projekttypen|Erweiterungen|Gemeinsames Add-In*:



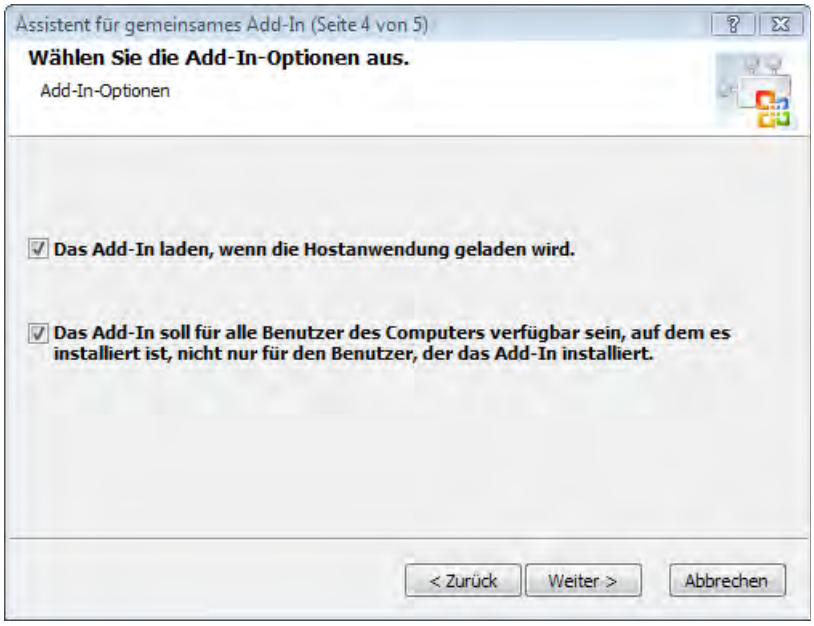
Nachfolgend erscheint ein Assistent, wählen Sie hier als Programmiersprache "Visual C#". Als Anwendungshost (Schritt 2) genügt uns Microsoft Access:



Dritter Schritt ist das Festlegen von Name und Beschreibung des Add-Ins:



Die beiden folgenden Optionen lassen wir für die Entwurfszeit markiert, auf diese Weise wird das Add-In automatisch beim Start von Access geladen und steht jederzeit zur Verfügung.



Nach einigem Festplattenklappern sollte ein neues Projekt in Visual Studio erscheinen, das aus dem eigentlichen Add-In und einem Setup-Projekt besteht (mehr dazu später).

Verweise einrichten

So wie das Projekt derzeit konfiguriert ist, ist die Programmierung sicher nicht komfortabel, fehlen doch alle Datentypen/Objekte und Konstanten, die wir für die Access-Programmierung benötigen. Fügen Sie also noch einen Verweis auf die *Microsoft Access 12.0 Object Library* hinzu.

Klicken Sie dazu im Projektmappen-Explorer auf den Add-In-Knoten (Kontextmenü *Verweis hinzufügen*) und wählen Sie im folgenden Dialogfenster in der Rubrik "COM" die oben genannte Library aus. Nachfolgend sollten die erforderlichen Verweise in Ihrem Projekt auftauchen.

Einen weiteren Verweis müssen Sie in der Rubrik ".NET" auswählen, es handelt sich um die Assembly *System.Windows.Forms*.

Quellcode Add-In

Damit können wir uns dem Quellcode des Add-Ins zuwenden. Das im Folgenden abgedruckte Listing ist um die fett hervorgehobenen Zeilen ergänzt bzw. erweitert worden:

```
namespace DokoAddIn
{
 using System;
 using Extensibility;
 using System.Runtime.InteropServices;
```

Die Unterstützung für den Access-Namespace:

```
    using Access = Microsoft.Office.Interop.Access;
    using System.Windows.Forms;

 [GuidAttribute("D7ED4A40-A614-414D-B40C-340643B6B8EF"), ProgId("DokoAddIn.Connect")]
 public class Connect : Object, Extensibility.IDTExtensibility2
 {
```

In der folgenden Variablen speichern wir eine Referenz auf die aufrufende Access-Anwendung:

```
        private Access.Application AccessApp;
        private Microsoft.Office.Core.COMAddIn addInInstance;
```

Hier werden einige Verweise auf Access-Objekte gespeichert:

```
        private Access.CommandButton _button1;
        private Access.ComboBox _combo1;
        private Access.Form _form;
```

Die erste Kontaktaufnahme zwischen Add-In und Access:

```
   public void OnConnection(object application, Extensibility.ext_ConnectMode connectMode,
                            object addInInst, ref System.Array custom)
   {
```

Wir speichern die Verweise ab (Typisierung nötig, da nur *Object* übergeben wird):

```
            AccessApp = (Access.Application) application;
```

```
            addInInstance = (Microsoft.Office.Core.COMAddIn)addInInst;
```

Der Add-In-Instanz wird das aktuelle Objekt (*this*) übergeben:

```
            addInInstance.Object = this;
    }
```

Über obiges *addInInstance.Object* greifen wir auch in Access auf die Member des Add-Ins zu.

Eine erste Methode für unser Add-In:

```
        public void Info()
        {
            MessageBox.Show("Hallo User");
        }
```

Mit der folgenden Methode verbinden wir die Access-Objekte mit unserem Add-In. Dazu übergeben wir in Access die Referenzen auf die gewünschten Objekte:

```
        public void ControlsAnbinden(Access.CommandButton btn, Access.ComboBox cb,
                                     Access.Form frm)
        {
```

Abspeichern der Referenzen:

```
            _button1 = btn;
            _form = frm;
            _combo1 = cb;
```

Ab hier dürfte sich das Programm kaum von einem VBA-Programm unterscheiden, wir arbeiten mit den Access-Objekten mit dem "kleinen" Unterschied, dass es sich um C# handelt:

```
            _button1.Caption = "Hier geht die Post ab ...";
```

Die folgende Deklaration ist zwar nicht für C# nötig, Access feuert aber nicht das Ereignis, wenn die folgende Zuweisung fehlt:

```
            _button1.OnClick = "[Event Procedure]";
            _form.OnClose = "[Event Procedure]";
```

Wir füllen eine Access-ComboBox:

```
            _combo1.RowSourceType = "Wertliste";
            _combo1.AddItem("Zeile 1");
            _combo1.AddItem("Zeile 2");
            _combo1.AddItem("Zeile 3");
            _combo1.AddItem("Zeile 4");
            _combo1.AddItem("Zeile 5");
            _combo1.AddItem("Zeile 6");
```

Noch die Schaltfläche beschriften und die beiden Ereignisprozeduren anbinden:

```
            _button1.Caption = "ENDE"; // Nicht schön, aber möglich
            _button1.Click += _button1_click;
            _form.Close += _form_Close;
        }
```

Die Ereignisbehandlung für den *Button*:

```
        private void _button1_click()
        {
            string s = null;
```

Eine einfache Meldung anzeigen (nur zur Kontrolle):

```
            MessageBox.Show("Button_Click");
```

Wir instanziieren ein WinForm und füllen es mit Daten aus einem Access-Formular:

```
            Form1 f = new Form1();
            f.textBox1.Text = _combo1.RowSource.Replace(";", "\r\n");
```

Anzeige des Dialogs:

```
            f.ShowDialog();
```

Auswerten und Anpassen der *ComboBox* in Access:

```
            _combo1.RowSource = "";
            s = f.textBox1.Text.Replace("\r\n", ";");
            _combo1.RowSource = s;
        }
```

Hier könnten wir auf das Schließen des Access-Formulars reagieren:

```
        private void _form_Close()
        {
            MessageBox.Show("Und jetzt ist das Formular zu ....");
        }
'''
}
```

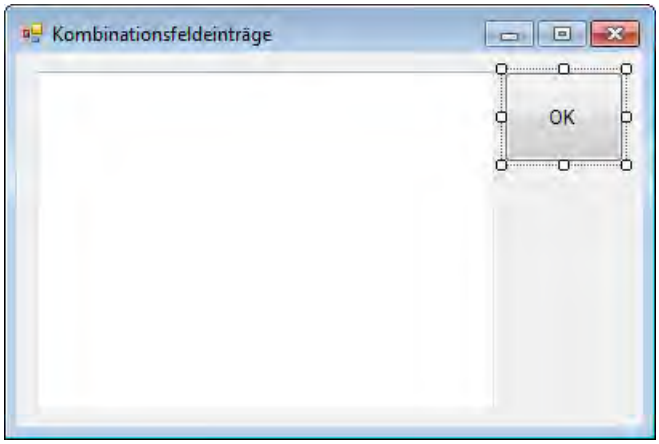
HINWEIS: Wir hätten obige Aufgabenstellung sicher auch mit Webservices oder Remoting bzw. den Zugriff auf den SQL Server bereichern können, aber das Beispiel wird dadurch sicher nicht übersichtlicher.

Oberfläche Add-In

Wie Sie dem obigen Listing entnehmen konnten, rufen wir auch ein .NET-Formular auf. Dieses müssen wir zunächst erstellen (Menü *Projekt/Windows Form hinzufügen*). Fügen Sie eine *TextBox* und einen *Button* ein und legen Sie die *Multiline*-Eigenschaft der *TextBox* mit *True* fest.

HINWEIS: Vergessen Sie nicht die *Modifiers*-Eigenschaften auf *Public* festzulegen!

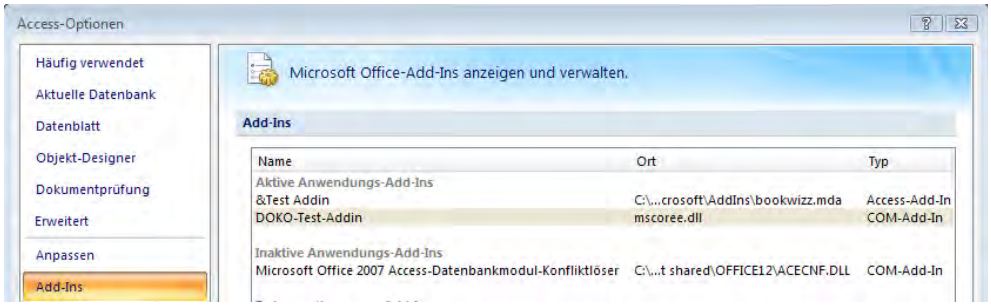
Damit wir das Formular auch deutlich von einem Access-Formular unterscheiden können, setzen wir die *Opacity*-Eigenschaft (Transparenz) auf 50%.



Erstellen der Access-Anwendung

Kompilieren und Einbinden

Nach dem Kompilieren des Add-Ins in Visual Studio (Menü *Erstellen*) können Sie sich bereits in Microsoft Access (*Optionen*) von der Anwesenheit des neuen Add-Ins überzeugen:



Doch wie können wir auf das Add-In zugreifen?

Erster Test

Eine kurze VBA-Routine (öffnen Sie dazu den VBA-Editor) zeigt die Vorgehensweise, bevor wir uns an ein eigenes Formular wagen:

```
Sub test()
  With COMAddIns("DOKOAddIn.Connect")
     .Connect = True
```

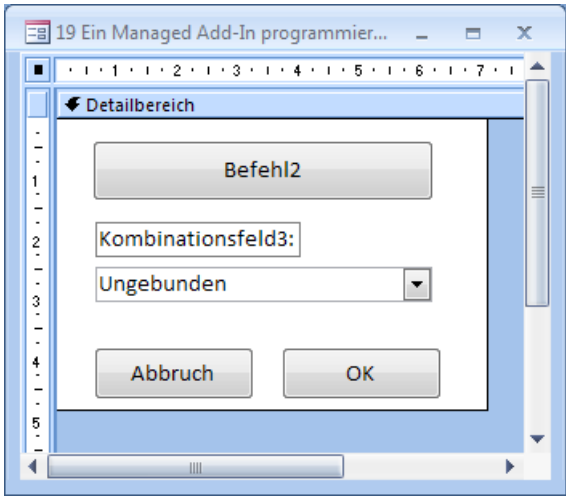
Hier rufen wir unsere Methode auf:

```
    .Object.info
  End With
End Sub
```

Nach dem Start mit F5 sollte unsere .NET-*MessageBox* erscheinen.

Das Access-Formular

Erstellen Sie ein einfaches Access-Formular nach der Vorlage in folgender Abbildung. Das Kombinationsfeld bzw. dessen Inhalt werden wir zur Laufzeit mit den Werten aus dem Add-In füllen.

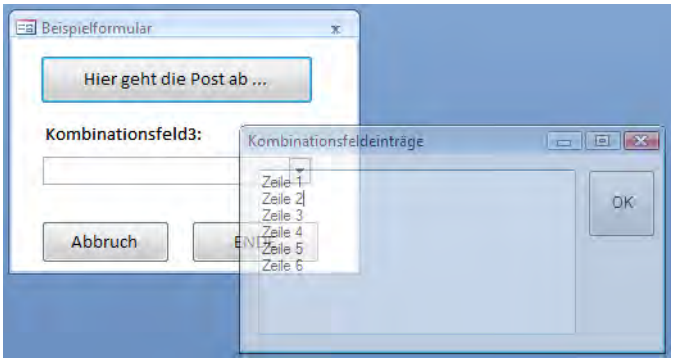


Mit dem Laden des Access-Formulars wird folgender Code ausgeführt:

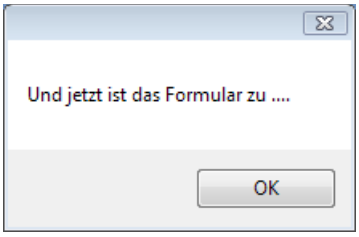
```
Private Sub Form_Load()
  With COMAddIns("DOKOAddIn.Connect")
    .Connect = True
    .Object.ControlsAnbinden Befehl2, Kombinationsfeld1, Me     ' Referenzen übergeben
  End With
End Sub
```

Test

Nach dem Starten des Formulars und dem Klick auf die obere Schaltfläche (beachten Sie die geänderte Beschriftung!) wird unser halbtransparentes .NET-Formular angezeigt:



Auch beim Schließen des Formulars erscheint die Meldung aus dem Add-In:



Bemerkungen

Wie Sie gesehen haben, ist eine nahtlose Integration in die Access-Umgebung möglich.

HINWEIS: Verwenden Sie im Add-In ADO-Objekte, dürfte auch der Datenzugriff auf die gerade geöffnete Datenbank kein Problem sein.

Sicher konnten wir hier zu dieser komplexen Materie nur einen ersten Einblick gewähren, das relativ einfach umsetzbare Grundprinzip dürfte jedoch erkennbar geworden sein.

Mit der kostenpflichtigen Software Add-in Express™ können Sie auch in Visual Studio 2012 Add-Ins für ältere Office-Versionen erstellen:

LINK: `http://www.add-in-express.com/add-in-net/index.php`

R197 Zugriff auf die serielle Schnittstelle

Zahlreiche Mess- und Elektrogeräte[1] sind auch heute noch mit der klassischen seriellen Schnittstelle ausgestattet. In Verbindung mit dem zu Visual Studio mitgelieferten *SerialPort*-Steuerelement können Sie beliebige Programme schreiben, mit denen sich diese Geräte steuern lassen.

[1] Sogar der Hometrainer des Autors verfügt über einen seriellen RS232-Port.

Um eine sinnvolle praktische Anwendung des *SerialPort*-Controls mit geringstem Aufwand zu demonstrieren, verwenden wir hier als Peripheriegerät ein Digitalvoltmeter (z.B. DT9602R) mit serieller Schnittstelle[1]. Es dürfte aber auch jedes andere DVM mit serieller Schnittstelle geeignet sein. Die folgende Abbildung zeigt das DVM beim Messen einer Batteriespannung, oben sieht man das Kabel, welches zum seriellen COM-Port des PC führt.

Oberfläche

Auf das Startformular *Form1* setzen Sie ein mittels *BorderStyle*-, *Font*- und *Color*- Eigenschaften stattlich herausgeputztes *Label*, welches als Anzeigeelement dienen soll. Weiterhin benötigen Sie als Herzstück natürlich das *SerialPort*-Control sowie einen simplen *Button* zum Beenden.

Beim *SerialPort*-Control können Sie es meist bei den im Eigenschaftenfenster vorgegebenen Standardeigenschaften belassen (*DataBits = 8; Parity = None; StopBits = One*; ...) belassen. Lediglich die Eigenschaften *PortName* und *BaudRate* müssen Sie individuell anpassen. Letzteren Wert können Sie in der Regel dem Datenblatt des DVM entnehmen (z.B. Baudrate 2400 für das verwendete DT9602R). Und noch etwas sollten Sie nicht vergessen:

HINWEIS: In unserem Fall ist die *DtrEnable*-Eigenschaft des *SerialPort*-Controls auf *True* zu setzen!

Quellcode

```
using System;
using System.Windows.Forms;
using System.IO.Ports;

public partial class Form1 : Form
```

[1] Preisgünstige Digitalvoltmeter mit RS232-Schnittstelle gibt es schon für ca. 30 Euro bei (fast) jedem Elektronikversand.

```
{
   ...
```

Beim Laden des Formulars wird der Port geöffnet:

```
   private void Form1_Load(object sender, EventArgs e)
   {
        serialPort1.Open();
   }
```

Messdaten sind eingetroffen:

```
   private void serialPort1_DataReceived(object sender, SerialDataReceivedEventArgs e)
   {
        try
        {
            int count = serialPort1.BytesToRead;   // Anzahl Bytes im Empfangspuffer des Ports
```

Um den Inhalt des Empfangspuffers komplett als Zeichenkette auszulesen, könnte man die *ReadExisting*-Methode verwenden. Für die Übertragung von Zahlenwerten (Messdaten etc.) scheint es aber zweckdienlicher zu sein, wenn wir von der Text- auf die Byte-Ebene hinabsteigen. Die im Folgenden vorgestellte Variante benutzt die *Read*-Methode der *SerialPort*-Komponente, um eine bestimmte Anzahl von Bytes aus dem Empfangspuffer in ein Byte-Array zur weiteren Verarbeitung einzulesen:

```
            byte[] ba = new byte[count];
            serialPort1.Read(ba, 0, count);
```

Es folgt das Aufsplitten des übergebenen Byte-Arrays in zwei Strings (Messwert und Messbereich). In unserem Beispiel entsteht ein String aus acht ASCII-Zeichen. Dabei bilden die ersten fünf Zeichen den Messwert. Danach folgt ein Leerzeichen. Die letzten beiden Zeichen verweisen auf den Messbereich:

```
            string data1 = System.Text.Encoding.Default.GetString(ba, 0, 5);      // Messwert
            string data2 = System.Text.Encoding.Default.GetString(ba, 6, 2);      // Messbereich
```

Messwert in Gleitkommazahl parsen:

```
            float f = float.Parse(data1);
```

Messbereichsabhängige Umrechnung und Anzeige:

```
            label1.Invoke(new EventHandler(delegate       // Umschaltung auf UI-Thread
            {
                switch (data2)
                {
                    case "11": label1.Text = f / 1000 + " V"; break;
                    case "21": label1.Text = f / 100 + " V"; break;
                    case "41": label1.Text = f / 10 + " mV"; break;
                    default: label1.Text = String.Empty; break;
                }
            }
            ));
        }
```

```
        catch { }
    }
```

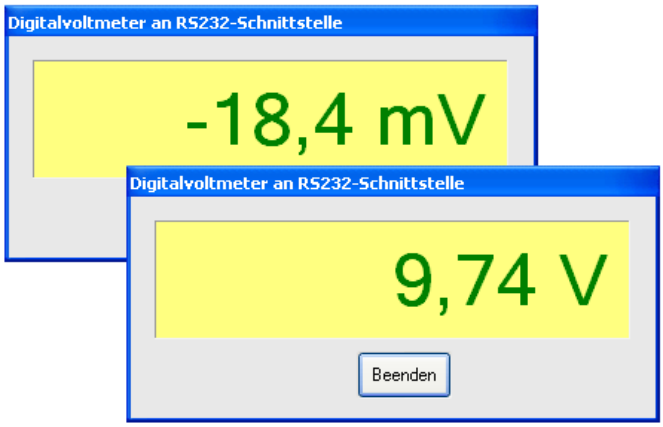
Beim Beenden des Programms sollte auch der Port wieder geschlossen werden:

```
    private void button1_Click(object sender, EventArgs e)
    {
        serialPort1.Close();
        this.Close();
    }
}
```

Test

Verbinden Sie Ihr DVM über das im Zubehör enthaltene Kabel mit der seriellen COM1-Schnittstelle des PC (andernfalls *PortName*-Eigenschaft im Quellcode anpassen). Stellen Sie den Messbereichsschalter des DVM auf Gleichspannung (VOLT DC) ein und aktivieren Sie die Übertragung per RS232 (siehe Bedienungsanleitung des DVM).

In Abhängigkeit von der Wandlungsrate des DVM (die des DT9602R ist hier mit ca. 1/sek nicht sonderlich berauschend) sollte das Programm nun kontinuierlich die aktuellen Messwerte anzeigen.



Bemerkungen

- Gefahren für Mensch und Computer beim Experimentieren sind so gut wie ausgeschlossen, da die RS232-Schnittstelle bei Messgeräten meist über Optokoppler herausgeführt wird.
- Falls keine Unterlagen zum DVM verfügbar sind: Um die Codierung der Messwerte und Messbereiche experimentell herauszufinden, kann das in unserem Buch [Visual C# 2012 Grundlagen und Profiwissen] beschriebene Terminalprogramm benutzt werden (nicht vergessen: Häkchen bei DTR setzen!). Durch einige Versuche mit unterschiedlichen Spannungen lässt sich meist leicht feststellen, wie die einzelnen Ascii- Zeichen zu interpretieren sind.

- Für eine komplette Steuerung (Modelleisenbahn[1]) eignet sich z.B. das preisgünstige PC-Messmodul M 232, es verfügt über 6 Analogeingänge (10-Bit-Auflösung) sowie 8 digitale Ein-/Ausgänge. Davon kann ein Eingang als Zählereingang für ein 16-Bit-Register benutzt werden.
- Neuere PCs verfügen zugunsten zahlreicher USB-Ports meist nur noch über einen einzigen seriellen COM-Port. Hier lassen sich mit Hilfe handelsüblicher *USB zu SUB-D9-Adapter* weitere serielle Ports realisieren.

R198 Sound per MCI aufnehmen

Im vorliegenden Rezept wollen wir uns der Aufnahme von WAV-Dateien widmen, wobei (mittels unverwaltetem Code) auf Windows-Bordmittel, d.h. auf die "gute alte" MCI-Schnittstelle, zurückgegriffen wird.

Oberfläche

Ein Formular mit einer Schaltfläche, die Sie mit START beschriften, genügt.

Quellcode

```
...
using System.IO;
using System.Runtime.InteropServices;

 public partial class Form1 : Form
 {
   ...
```

Die folgenden zwei API-Deklarationen greifen auf die MCI-Schnittstelle Ihres PCs zu:

```
       [DllImport("winmm.dll", CharSet = CharSet.Auto)]
       private static extern int mciSendString(string lpstrCommand,
                               StringBuilder lpstrReturnString, int uReturnLength,
                               IntPtr hwndCallback);

       [DllImport("winmm.dll", CharSet = CharSet.Auto)]
       private static extern int mciGetErrorString(int dwError,
                                 StringBuilder lpstrBuffer, int uLength);
```

MCI-Fehlermeldung dekodieren (wir erhalten ausführliche Fehlerbeschreibungen):

```
       private string getMciError(int errCode)
       {
           StringBuilder errMsg = new StringBuilder(255);
           if (mciGetErrorString(errCode, errMsg, errMsg.Capacity) == 0)
```

[1] Das Kind im Mann freut sich!

```
            return "MCI-Fehler " + errCode;
        else
            return errMsg.ToString();
    }
```

Mit Hilfe der API-Funktion *GetShortPathName* können wir den kurzen Pfadnamen zum Speicherort der WAV-Datei ermitteln, da ansonsten (z.B. wegen Leerzeichen im Dateinamen etc.) die WAV-Datei von *mciSendString* nicht gespeichert werden kann:

```
[DllImport("kernel32.dll", CharSet = CharSet.Auto)]
private static extern int GetShortPathName(string lpszLongPath,
                        StringBuilder lpszShortPath, int cchBuffer);

private string shortPathName()
{
    string shortPath = string.Empty;
    long len = 0;
    StringBuilder buffer = new StringBuilder(256);
    string s = Directory.GetCurrentDirectory();
    len = GetShortPathName(s, buffer, 256);
    shortPath = buffer.ToString();
    return shortPath;
}
```

Nach all diesen Vorbereitungen kann es nun endlich losgehen. Bereits beim Laden des Formulars wird die MCI-Schnittstelle angesprochen:

```
private void Form1_Load(object sender, EventArgs e)
{
```

Eine neue (leere) Sound-Aufnahme eröffnen:

```
    string mciString = "open new type waveaudio alias myAlias";
    int res = mciSendString(mciString, null, 0, IntPtr.Zero);
    if (res != 0)
        MessageBox.Show(getMciError(res),
                        "MCI-Fehler beim Öffnen des Geräts ('Open New'-Befehl)",
                        MessageBoxButtons.OK, MessageBoxIcon.Error);
```

Parameter einstellen: 8000 Abtastungen pro Sekunde mit je 1 Byte:

```
    mciString = "set myAlias time format ms bitspersample" +
                " 8 channels 1 samplespersec 8000 bytespersec 8000";
    res = mciSendString(mciString, null, 0, IntPtr.Zero);
    if (res != 0)
        MessageBox.Show(getMciError(res),
            "MCI-Fehler beim Zuweisen der Parameter ('Set-Befehl)",
            MessageBoxButtons.OK, MessageBoxIcon.Error);
}
```

Da es sich bei obigen Parametern um die Standardwerte handelt, könnte in unserem Fall dieser MCI-Aufruf auch weggelassen werden.

Und schließlich der START/STOPP-Button:

```
        private void button1_Click(object sender, EventArgs e)
        {
            if (button1.Text == "START")  // auf START geklickt
            {
```

Die Aufnahme beginnt:

```
                string mciString = "record myAlias";
                int res = mciSendString(mciString, null, 0, IntPtr.Zero);
                if (res != 0)
                    MessageBox.Show(getMciError(res),
                          "MCI-Fehler beim Starten der Aufnahme ('Record-Befehl)",
                          MessageBoxButtons.OK, MessageBoxIcon.Error);
                button1.Text = "STOPP";
            }
            else    // auf STOPP geklickt
            {
```

Die Sound-Datei wird im aktuellen Verzeichnis abgespeichert:

```
                string mciStr = "save myAlias " + shortPathName() + "\\Test.wav";
                int res = mciSendString(mciStr, null, 0, IntPtr.Zero);
                if (res != 0)
                    MessageBox.Show(getMciError(res),
                            "MCI-Fehler beim Speichern der Datei ('Save'-Befehl)",
                            MessageBoxButtons.OK, MessageBoxIcon.Error);
                button1.Text = "START";
            }
        }
    }
```

Test

Vergewissern Sie sich, dass an Ihren PC ein Mikrofon angeschlossen ist und dass es korrekt funktioniert (siehe Bemerkungen).

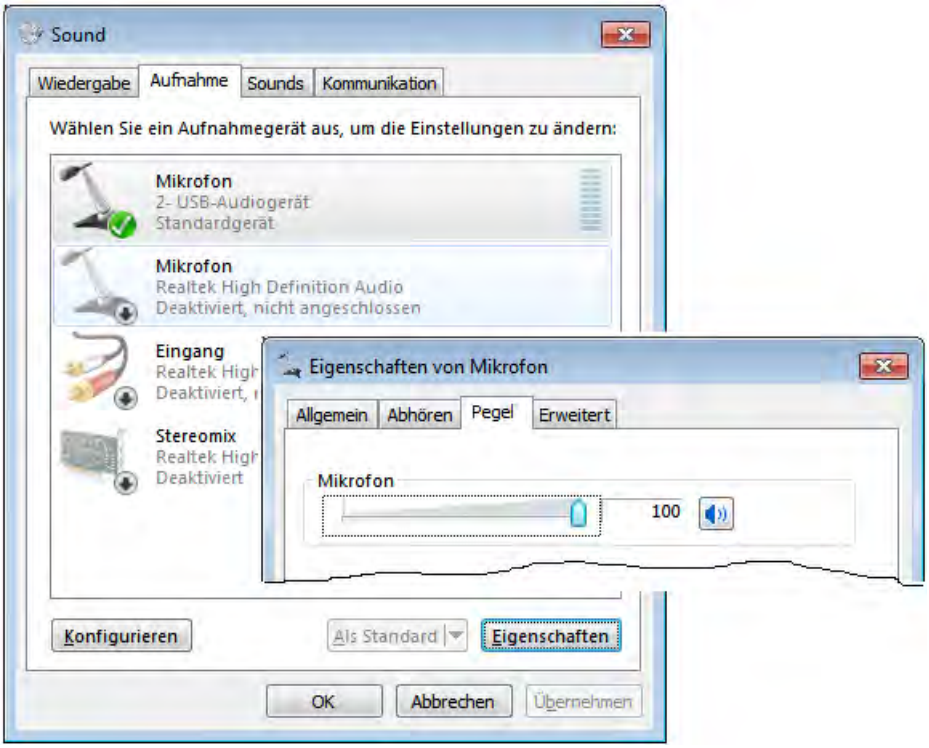
Nach Klick auf die START-Schaltfläche ändert sich deren Beschriftung in STOPP und die Aufnahme beginnt. Jetzt können Sie zum Beispiel eine Probe Ihrer Gesangskünste abliefern. Nach Aufnahmestopp findet sich die Sounddatei *Test.wav* im aktuellen Projektverzeichnis (bzw. im *.../bin/Debug*-Unterverzeichnis Ihres Projektordners).

Nach Doppelklick auf die Sounddatei dürfte sich normalerweise der Windows Media Player öffnen und Sie können sich das Ergebnis Ihrer gesanglichen Darbietungen anhören, oder aber Sie verwenden zur Wiedergabe das Rezept

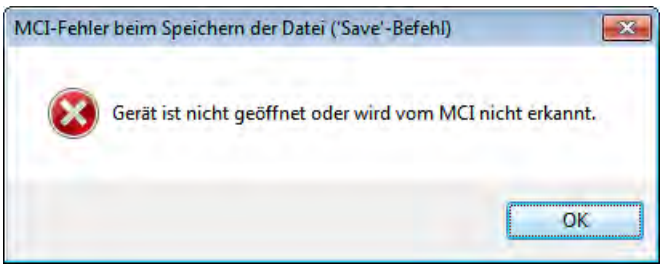
► R201 Sound-und Video-Dateien per MCI abspielen

Bemerkungen

- Über die Systemsteuerung (oder auch die Taskbar) haben Sie Zugriff auf die Dialoge für Mikrofonauswahl und Pegeleinstellungen:



- Bei schlechter Soundqualität sollten Sie vergleichsweise zunächst mit dem im Windows-Zubehör enthaltenen Sound Recorder einige Probeaufnahmen machen.
- Die ausführlichen MCI-Meldungstexte helfen Ihnen bei der Fehlersuche, zum Beispiel wenn ein falscher oder doppelt vergebener Alias verwendet wurde:



R199 Mikrofonpegel anzeigen

Bei Soundaufnahmen wünscht man sich meist auch eine Anzeige des Mikrofonpegels, z.B. in Form eines Aussteuerungsbalkens. Im Folgenden wollen wir das Vorgängerrezept R198 mit einer einfachen Pegelanzeige ergänzen.

Oberfläche

Zum Formular des Vorgängerbeispiels fügen Sie eine *ProgressBar* (*Maximum* = 128) und einen *Timer* (*Interval* = 10) hinzu.

Quellcode

Ergänzen Sie zunächst den *Load*-Eventhandler des Formulars wie folgt:

```
private void Form1_Load(object sender, EventArgs e)
{
     ...
```

Leider kann ein geöffnetes Audiogerät nicht gleichzeitig eine Sound-Datei aufnehmen und den Pegel anzeigen. Wir öffnen deshalb eine zweite Sound-Aufnahme mit unterschiedlichem Aliasnamen:

```
          mciString = "open new type waveaudio alias myAlias2";
          res = mciSendString(mciString, null, 0, IntPtr.Zero);
          if (res != 0)
              MessageBox.Show(getMciError(res),
                              "MCI-Fehler beim Öffnen des Geräts ('Open New'-Befehl)",
                              MessageBoxButtons.OK, MessageBoxIcon.Error);
```

Auf eine Parameterzuweisung können wir in unserem Fall verzichten, da mit den Standardwerten gearbeitet wird.

```
          timer1.Start();
   }
```

In der zweiten Programmergänzung wird im 10ms Takt der Pegel abgerufen:

```
private void timer1_Tick(object sender, EventArgs e)
{
```

Der aktuelle Pegel kann mittels *status*-Befehl über das *level*-Flag abgefragt werden:

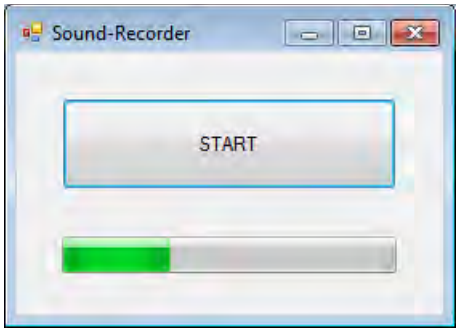
```
          string mciString = "status myAlias2 level";
          StringBuilder buffer = new StringBuilder(20);
          int res = mciSendString(mciString, buffer, buffer.Capacity, IntPtr.Zero);
          if (res != 0)
              MessageBox.Show(getMciError(res),
                         "MCI-Fehler bei der Status-Abfrage  ('status ... level'-Befehl)",
                         MessageBoxButtons.OK, MessageBoxIcon.Error);
```

Der Input-Level ist ein vorzeichenloser Wert. Für die von uns gewählte 8-Bit Abtastgenauigkeit liegt er zwischen 0 und 127 (0x7F), bei 16-Bit Abtastung zwischen 0 und 32.767 (0x7FFF):

```
            byte level = byte.Parse(buffer.ToString());    // 0 ... 127
            progressBar1.Value = level;
}
```

Test

Die Aussteuerungsanzeige funktioniert sowohl vor als auch während der Aufnahme.



R200 Pegeldiagramm aufzeichnen

So richtig professionell wirkt ein Sound Recorder erst dann, wenn man den Pegelverlauf in Echtzeit auf einem Diagramm verfolgen kann. Diese, u.a. auch bei medizinischen Überwachungsgeräten übliche Darstellung wollen wir anhand einer Programmerweiterung des Vorgängerrezepts

► R199 Mikrofonpegel anzeigen

demonstrieren.

Oberfläche

Ergänzen Sie das vorhandene Formular durch eine *PaintBox* und ein repräsentatives *Label*. Gestalten Sie die Oberfläche entsprechend der Laufzeitabbildung am Schluss dieses Rezepts.

Quellcode-Ergänzungen

Um wegen der zahlreichen Ergänzungen die Übersicht nicht zu verlieren, empfiehlt es sich insbesondere bei diesem Rezept, den kompletten Code von den Buchdaten in die Entwicklungsumgebung zu laden und in seiner Gesamtheit zu analysieren.

```
...
using System.Drawing;                    // für die Grafikoperationen
```

Zunächst sind einige globale Variablen hinzuzufügen:

```
private Graphics g = null;               // Graphics-Objekt für PaintBox
private byte[] BA = null;                // Puffer-Array für die Pegelwerte
private float xmax,                      // Breite des Diagramms (Pixel)
              y0,                        // y-Koordinate der Diagramm-Mittellinie (Pixel)
               v;                        // Streckungsfaktor der Amplitude
private int i = 0;                       // Zähler für Timer-Ticks ab Start
private DateTime dStart;                 // Startzeit der Aufnahme
private int msec = 0;                    // seit Start abgelaufene Zeit in Millisekunden
```

Die Pegelabfrage soll alle 10 ms stattfinden, die Länge des aufzuzeichnenden Diagramms beträgt 1 Minute:

```
private const int iv = 10;               // Timer-Intervall (ms)
private const int tmax = 60000;          // Länge der Zeitachse (ms)
```

Initialisieren der globalen Variablen:

```
private void Form1_Load(object sender, EventArgs e)
{
     ...
     g = pictureBox1.CreateGraphics();
     iv = timer1.Interval;              // Voreinstellung 10ms
```

Die erforderliche Größe des Puffer-Arrays ergibt sich aus obigen Konstanten:

```
     BA = new Byte[tmax / iv];          // 6000 gespeicherte Werte (ca. 60sek)
```

Abmessungen des Diagramms:

```
     xmax = pictureBox1.Width;
     y0 = pictureBox1.Height / 2;
```

Der vom MCI gelieferte Maximalpegel beträgt 127. Da bei Vollaussteuerung der obere bzw. untere Rand der *PaintBox* nicht ganz erreicht werden soll, bauen wir in den Streckungsfaktor eine kleine Reserve ein:

```
     v = y0 / 140;
     ...
}
```

Eine Ergänzung im *Click*-Eventhandler des Buttons:

```
private void button1_Click(object sender, EventArgs e)
{
    if (button1.Text == "START")
    {
              ...
```

Nach dem Klick auf START beginnt eine neue Aufnahme, d.h., die Startzeit wird zugewiesen, der Zähler zurückgesetzt und das alte Diagramm gelöscht:

```
            dStart = DateTime.Now;
            i = 0;
            pictureBox1.Refresh();
    }
    ...
}
```

Bei einem Timerintervall von 10ms und 6000 Durchläufen ist die Endzeit in der Regel deutlich länger als die erwarteten 60 Sekunden[1]. Wollen Sie eine exakte Anzeige der seit Start abgelaufenen Sekunden hinzufügen, so kommen Sie um ein *TimeSpan*-Objekt nicht herum.

Der Aufbau der Grafik erfolgt synchron mit der Pegelabfrage. Der vorhandene *Tick*-Eventhandler des *Timer*s ist deshalb wie folgt zu ergänzen:

```
private void timer1_Tick(object sender, EventArgs e)
{
   ...
```

Die folgende Zeile ist kein Fehler, denn während die Soundaufnahme läuft, hat der Button die Beschriftung STOPP:

```
    if (button1.Text == "STOPP")
    {
        TimeSpan ts = new TimeSpan(DateTime.Now.Ticks - dStart.Ticks);
        int sec = Convert.ToInt32(ts.TotalSeconds);
```

Anzeige der abgelaufenen Zeit (Sekunden):

```
            label1.Text = sec.ToString();
```

Abbruchkriterium bei 60sek:

```
            if (sec == tmax / 1000)
            {
```

Speichern am Schluss:

```
                saveFile();
                button1.Text = "START";
                return;
            }
```

Um die x-Koordinate exakt berechnen zu können, muss die Auflösung zumindest im Millisekundenbereich liegen:

```
            msec = Convert.ToInt32(ts.TotalMilliseconds);
            float x = msec * xmax / tmax;
```

[1] Windows ist kein Echtzeit-Betriebssystem und Prozesse mit höherer Priorität können solche mit niedrigerer Priorität ausbremsen.

Pegel in Diagramm eintragen:

```
                g.DrawLine(new Pen(Color.Red, 0.1F), x, y0 - level * v, x, y0 + level * v);
```

Abspeichern des Pegels im Puffer-Array:

```
                BA[i] = level;
                i++;
    }
}
```

Damit nach einem Verdecken des Formulars das Diagramm nicht auf Nimmerwiedersehen verschwindet, muss es im *Paint*-Eventhandler der *PaintBox* komplett neu gezeichnet werden:

```
private void pictureBox1_Paint(object sender, PaintEventArgs e)
 {
    Graphics g = e.Graphics;
```

Abstand zwischen zwei gespeicherten Werten auf Position *i*:

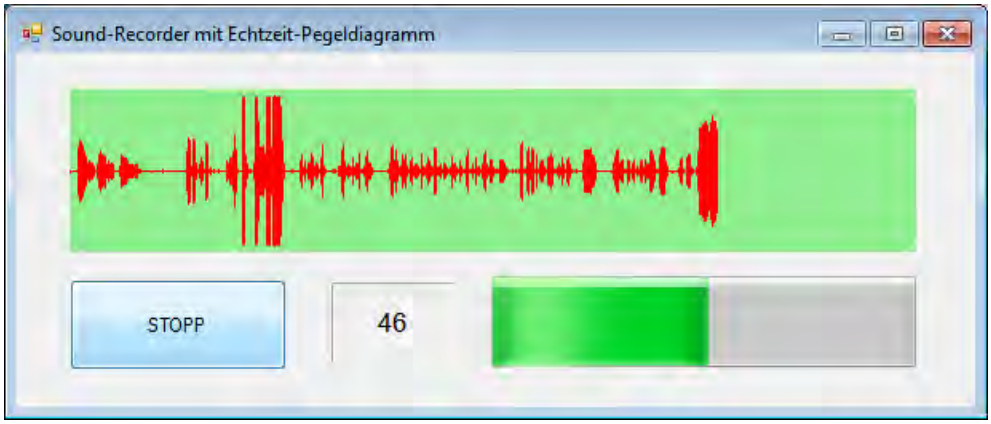
```
    float dx = xmax / tmax * msec / i;
```

Der Pufferspeicher wird bis zur Position *i* ausgelesen und angezeigt:

```
    for (int j = 0; j < i; j++)
        g.DrawLine(new Pen(Color.Red, 0.1F),j*dx, y0 - BA[j] * v, j*dx, y0 + BA[j] * v);
 }
```

Test

Jetzt können Sie unmittelbar während der Sound-Aufnahme den kontinuierlichen Pegelverlauf "live" beobachten.



Bei STOPP, bzw. nach Ablauf von 60 Sekunden, wird die Datei *Test.wav* automatisch im Projektverzeichnis abgelegt.

R201 Sound-und Video-Dateien per MCI abspielen

Für die Wiedergabe von Sound und Videos in Windows Forms-Anwendungen gibt es bereits einige vorgefertigte .NET- oder auch ActiveX-Steuerelemente (*SoundPlayer*, *WindowsMediaplayer*, ...).

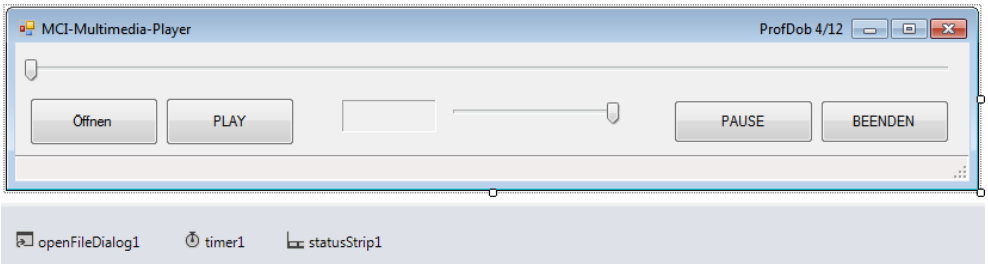
Wollen Sie aber individuelle Wünsche realisieren, wie zum Beispiel eine eigene Benutzeroberfläche, oder einfach nur mit verschiedenen Befehlen experimentieren, so können Sie die MCI-Schnittstelle direkt ansprechen. Diese allerdings ist nicht Teil von .NET, sondern gehört zum Uralt-Bestand von Windows (seit 3.1), d.h., die Fremdbibliothek *winmm.dll* muss importiert werden.

Mittels einfacher Kommandos (*Open*, *Play*, ...) können Sie dann auf alle verfügbaren Multimedia-Geräte zugreifen, für die ein MCI-Treiber existiert.

Das vorliegende Rezept demonstriert dies anhand eines einfachen "selbstgestrickten" Players für Sound- und Videodateien unterschiedlichster Formate (Wave, MP3, Midi, AVI, WMV, MPEG etc.).

Oberfläche

Öffnen Sie eine neue Windows Forms-Anwendung und gestalten Sie die Oberfläche von *Form1* mit vier *Button*s, zwei *TrackBar*s und je einem *Label*, *OpenFileDialog*, *Timer* und *StatusStrip*. Setzen Sie die *TopMost*-Eigenschaft von *Form1* auf *True*.



Klasse C_MCI

Den Zugriff auf die MCI-Schnittstelle kapseln wir in einer separaten Klasse *C_MCI*, die wir über das Menü *Projekt/Klasse hinzufügen...* erzeugen:

```
...
using System.IO;
using System.Text;
```

Vergessen Sie nicht, für den (unmanaged) Zugriff auf die Windows-API den folgenden Namespace einzubinden:

```
using System.Runtime.InteropServices;

namespace MciPlayer
{
    public class C_MCI
```

```
{
```

Die beiden erforderlichen MCI-Funktionen sind Teil der Datei *winmm.dll*. Dabei definiert der (optionale) *CharSet*-Parameter die automatische Übergabe von Zeichenfolgen (Marshalling) entsprechend des jeweiligen Zielbetriebssystems.

```
[DllImport("winmm.dll", CharSet = CharSet.Auto)]
```

Die Funktion *mciSendString* ist das Herzstück des Programms, sie sendet unsere Befehle an die MCI-Schnittstelle:

```
private static extern int mciSendString(string lpstrCommand,
                                        StringBuilder lpstrReturnString,
                                        int uReturnLength, IntPtr hwndCallback);
```

Die MCI-Funktion *mciGetErrorString* gibt uns Fehlertexte für einen bestimmten Fehlercode zurück:

```
[DllImport("winmm.dll", CharSet = CharSet.Auto)]
private static extern int mciGetErrorString(int dwError,
                                       StringBuilder lpstrBuffer, int uLength);
```

Ein bestimmtes MCI-Gerät wird innerhalb der Befehlszeichenkette über einen frei definierbaren Alias identifiziert:

```
private string alias = "mciAlias";
```

Eine private Zustandsvariable zeigt an, ob das angesprochene MCI-Gerät geöffnet ist:

```
private bool isOpen = false;
```

Der Lesezugriff:

```
public bool IsOpen
{
    get
    {
        return this.isOpen;
    }
}
```

Mit der Funktion *getMciError* kapseln wir den Aufruf der API-Funktion *mciGetErrorString* und erhalten damit zum Fehlercode den entsprechenden Fehlertext:

```
private string getMciError(int errCode)
{
    StringBuilder errMsg = new StringBuilder(255);
    if (mciGetErrorString(errCode, errMsg, errMsg.Capacity) == 0)
        return "MCI-Fehler " + errCode;
    else
        return errMsg.ToString();
}
```

Alles beginnt mit dem Befehl zum Öffnen einer Multimedia-Datei:

```
public void Open(string filename)
{
```

Vorher schließen wir eine eventuell zuvor noch geöffnete Datei:

```
    if (this.IsOpen) this.Close();
```

Nun wird überprüft, ob die übergebene Multimedia-Datei tatsächlich existiert:

```
    if (File.Exists(filename) == false)
       MessageBox.Show("Die Datei '" + filename + "' ist nicht vorhanden!", "Fehler",
                       MessageBoxButtons.OK, MessageBoxIcon.Error);
```

Jetzt kommen wir zum wichtigsten Parameter, dem MCI-Befehlsstring. Dieser besteht aus einzelnen Kommandos, die durch Leerzeichen voneinander getrennt sind. Da aber auch der Dateipfad Leerzeichen enthalten kann, führt das häufig zu unergründbaren Fehlern. Wir müssen deshalb den Dateinamen in doppelte Anführungszeichen einschließen, also gewissermaßen "String im String" realisieren, dazu müssen die Anführungszeichen durch Voranstellen eines Backslash (\) "escapt" werden.

Der komplette MCI-Befehlsstring zum Öffnen einer Datei besteht aus dem *Open*-Kommando, dem Dateinamen, dem Medientyp und dem Alias:

```
    string mciStr = "open \"" + filename + "\" type mpegvideo alias " + this.alias;
```

Nun kann das MCI-Gerät geöffnet werden:

```
    int res = mciSendString(mciStr, null, 0, IntPtr.Zero);
```

Zur Fehleranzeige verwenden wir hier einfachheitshalber ein Meldungsfenster, könnten aber stattdessen auch eine eigene Fehlerklasse definieren und eine Ausnahme werfen (*throw MCI_Exception*):

```
    if (res != 0)
        MessageBox.Show(getMciError(res),
                        "MCI-Fehler beim Öffnen des Geräts ('Open'-Befehl)",
                        MessageBoxButtons.OK, MessageBoxIcon.Error);
```

HINWEIS: Eine Fehlerbehandlung ähnlich der obigen haben wir bei allen MCI-Aufrufen implementiert, im Folgenden aber aus Platzgründen darauf verzichtet (siehe Buch-Beispieldaten).

Das Zeitformat für Positionsangaben auf Millisekunden festlegen:

```
    res = mciSendString("set " + this.alias + " time format ms", null, 0, IntPtr.Zero);
```

Unsere Multimedia-Datei ist geöffnet:

```
    this.isOpen = true;
}
```

Was nun folgt, ist die Kapselung anderer wichtiger MCI-Kommandos als Eigenschaften oder Methoden.

Die folgende Eigenschaft liefert die Abspiellänge der Multimedia-Datei (in Millisekunden):

```
public int Length
{
    get
    {
        StringBuilder buffer = new StringBuilder(255);
        int res = mciSendString("status " + this.alias + " length", buffer,
                                buffer.Capacity, IntPtr.Zero);
        return Convert.ToInt32(buffer.ToString());
    }
}
```

Die gesamte Multimedia-Datei abspielen:

```
public void Play()
{
    Play(0, this.Length);
}
```

Die Multimedia-Datei zwischen zwei Positionen abspielen:

```
public void Play(int from, int to)
{
    int res = mciSendString("Play " + this.alias + " From " + from + " To " +
                            to, null, 0, IntPtr.Zero);
}
```

Die folgende Eigenschaft liest/schreibt die aktuelle Abspielposition (Millisekunden):

```
public int Position
{
    get
    {
        StringBuilder buffer = new StringBuilder(261);
        int res = mciSendString("status " + this.alias + " position",
                                buffer, buffer.Capacity, IntPtr.Zero);
        if (res != 0)
           return Convert.ToInt32(buffer.ToString());
          // oder return int.Parse(buffer.ToString());
    }
    set
    {
        int res = mciSendString("seek " + this.alias + " to " + value, null,
                                0, IntPtr.Zero);
        res = mciSendString("play " + this.alias, null, 0, IntPtr.Zero);
    }
}
```

Eine Pause:

```
        public void Pause()
        {
```

Den Abspielvorgang unterbrechen

```
            int res = mciSendString("Pause " + this.alias, null, 0, IntPtr.Zero);
        }
```

Den Abspielvorgang fortsetzen:

```
        public void Resume()
        {
            int res = mciSendString("Resume " + this.alias, null, 0, IntPtr.Zero);
        }
```

Die Lautstärke festlegen:

```
        public void volume(int wert)
        {
            int res = mciSendString("setaudio " + this.alias + " volume to " + wert,
                                    null, 0, IntPtr.Zero);
        }
```

Den Abspielvorgang stoppen und auf den Anfang zurücksetzen:

```
        public void Stop()
        {
            int res = mciSendString("Stop " + this.alias, null, 0, IntPtr.Zero);
        }
```

Ein geöffnetes MCI-Gerät schließen:

```
        public void Close()
        {
            if (this.isOpen)
            {
                int res = mciSendString("Close " + this.alias, null, 0, IntPtr.Zero);
                this.isOpen = false;
            }
        }
    }
}
```

Form1

Nun steht der Realisierung unseres Multimedia-Players nichts mehr im Weg, und wir können die eingangs entworfene Bedienoberfläche beispielsweise mit folgendem Code hinterlegen:

```
...
namespace MciPlayer
{
```

```
public partial class Form1 : Form
{
   public Form1()
   {
        InitializeComponent();
```

Diese Werte könnten Sie auch direkt im Eigenschaftenfenster zuweisen:

```
        trackBar1.Maximum = 10000;
        trackBar1.LargeChange = 1;
        track>Bar2.Maximum = 1000;
        trackBar2.Value = 1000;
        trackBar2.LargeChange = 100;
        timer1.Interval = 100;
   }
```

Eine Instanz unseres Multimedia-Players erzeugen:

```
    private C_MCI myMCI = new C_MCI();
```

Korrekturfaktor für *TrackBar*-Anzeige:

```
    private float kf = 1;
```

Eine Multimedia-Datei öffnen:

```
    private void button1_Click(object sender, EventArgs e)
    {
            if (openFileDialog1.ShowDialog() == DialogResult.OK)
            {
                myMCI.Open(openFileDialog1.FileName);
```

Bei der Berechnung des Korrekturfaktors ist eine Typkonvertierung erforderlich, da sonst die Nachkommastellen abgeschnitten werden:

```
                kf = Convert.ToSingle(myMCI.Length) / trackBar1.Maximum;
                toolStripStatusLabel1.Text = openFileDialog1.FileName;
                button2.Text = "PLAY";
            }
    }
```

Die Schaltfläche PLAY/STOPP:

```
    private void button2_Click(object sender, EventArgs e)
    {
        if (button2.Text == "PLAY")
        {
            myMCI.Play();
            timer1.Enabled = true;
            button2.Text = "STOPP";
        }
        else
        {
            myMCI.Stop();
```

```
            timer1.Enabled = false;
            trackBar1.Value = 0;
            label1.Text = "00:00";
            button2.Text = "PLAY";
        }
    }
```

Die Schaltfläche PAUSE/WEITER:

```
    private void button3_Click(object sender, EventArgs e)
    {
        if (button3.Text == "PAUSE")
        {
            myMCI.Pause();
            button3.Text = "WEITER";
        }
        else
        {
            button3.Text = "PAUSE";
            myMCI.Resume();
        }
    }
```

Es folgt eine Hilfsfunktion für die Bereitstellung der laufenden Zeitanzeige in Minuten und Sekunden. Anhand der Reglerposition von *trackBar1* werden zunächst die Sekunden ermittelt. Mittels Integer- und Modulo-Division werden dann die Minuten und restlichen Sekunden berechnet und durch einen Doppelpunkt getrennt:

```
    private string getMinutes()
    {
        int sekunden = Convert.ToInt32((trackBar1.Value * kf / 1000));
        return (sekunden / 60).ToString("00") + ":" + (sekunden % 60).ToString("00");
    }
```

Im *Tick*-Event des *Timer*s werden der Regler der *Trackbar* vorwärts bewegt und die Zeitanzeige aktualisiert:

```
    private void timer1_Tick(object sender, EventArgs e)
    {
        int val = Convert.ToInt32(myMCI.Position / kf);
        if (val < trackBar1.Maximum)
        {
            trackBar1.Value = val;
            this.label1.Text = getMinutes();
        }
```

Das Ende unserer MM-Datei ist erreicht:

```
        else
        {
            myMCI.Stop();
            timer1.Enabled = false;
```

```
                trackBar1.Value = 0;
                label1.Text = "00:00";
                button2.Text = "PLAY";
            }
        }
```

Wenn Sie am Regler der *TrackBar* ziehen, können Sie die Abspielposition verändern:

```
        private void trackBar1_Scroll(object sender, EventArgs e)
        {
            try
            {
                this.myMCI.Position = Convert.ToInt32(trackBar1.Value * kf);
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message, Application.ProductName,
                                MessageBoxButtons.OK, MessageBoxIcon.Error);
            }
        }
```

Die Lautstärke lässt sich mit der kleinen *TrackBar* verändern:

```
        private void trackBar2_ValueChanged(object sender, EventArgs e)
        {
            myMCI.Volume(trackBar2.Value);
        }
```

Die Schaltfläche zum Beenden:

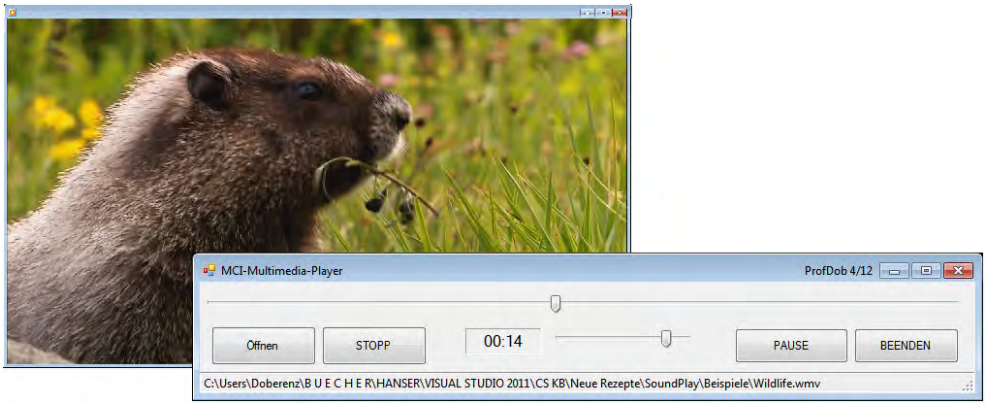
```
        private void button4_Click(object sender, EventArgs e)
        {
            this.Close();
        }

        private void Form1_FormClosing(object sender, FormClosingEventArgs e)
        {
            myMCI.Close();
        }
    }
}
```

Test

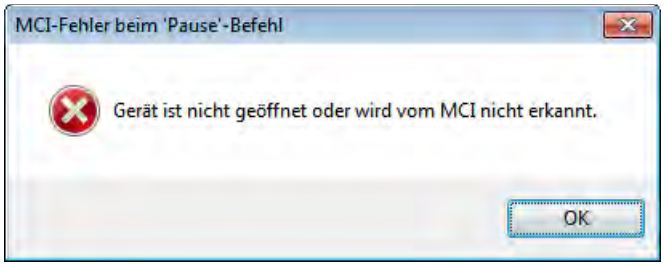
Ring frei für Ihre Experimente! Starten Sie das Programm und laden Sie über die *Öffnen*-Schaltfläche eine beliebige Sound- oder Video-Datei (*.wav, .wma, .mp3, .midi, .avi, .wmv*)[1]. Spulen Sie mit dem oberen Regler vor und zurück und testen Sie die übrigen Bedienfunktionen.

[1] Einige Beispieldateien finden sich im Windows-Zubehör.

Die Abbildung zeigt eine Momentaufnahme beim Abspielen der zu Windows mitgelieferten Videodatei *Wildlife.wmv*.



Auf fehlerhafte MCI-Kommandos werden Sie durch aussagekräftige Fehlermeldungen hingewiesen, zum Beispiel:



Bemerkungen

- In einer Windows Forms-Anwendung könnten Sie MCI auch durch die DirectX-Bibliothek ersetzen. Letztere müsste dann aber in vielen Fällen separat installiert werden, während hingegen jeder Computer von Haus aus auch über die erforderlichen MCI-Treiber verfügt.
- MCI bietet eine enorme Vielfalt von Befehlen für unterschiedlichste Dateiformate und Geräte, über die Sie sich am besten bei

  LINK: http://msdn.microsoft.com/en-us/library/windows/desktop/dd743572(v=vs.85).aspx

  informieren. So können Sie zum Beispiel mit dem folgenden Aufruf die Position und die Abmessungen des Fensters für die Video-Wiedergabe individuell festlegen:

```
mciSendString("put " + this.alias + " window at " + x + " " + y + " " + width + " " +
                                                                    height, null, 0, IntPtr.Zero);
```

R202 Eine C-DLL in C# einbinden

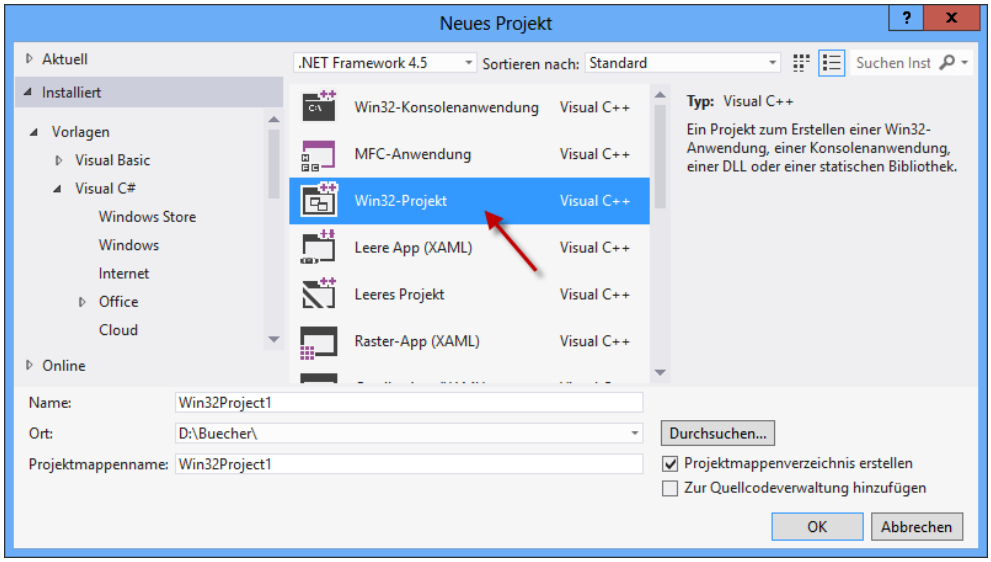
Aus verschiedensten Gründen kann es mitunter notwendig sein, dass Sie von Ihrer C#-Anwendung aus auch auf unverwalteten (unmanaged) Code zugreifen müssen. Sei es, dass Sie eine aus Altbeständen noch vorhandene und in C++ geschriebene DLL nutzen oder aber einen Teil der Programmlogik auslagern wollen, um eine Offenlegung Ihres wertvollen C#-Quellcodes zu verhindern oder zumindest zu erschweren[1].

Die "Mehrzweckwaffe" Visual Studio 2012 erlaubt Ihnen unter anderem auch das Erstellen von C++-DLLs, was wir am Beispiel einer einfachen Sortierroutine, die wir in ein C#-Projekt einbinden werden, demonstrieren wollen.

HINWEIS: Das vorliegende Rezept befasst sich mit dem Programmieren einer "echten" DLL, also nicht mit ActiveX-DLLs bzw. COM-Komponenten.

DLL-Projekt erstellen

Starten Sie die Entwicklungsumgebung Visual Studio 2012 und klicken Sie auf *Neues Projekt...*. Wählen Sie links unter der Vorlage *Andere Sprachen* die Sprache *Visual C++* und in der mittleren Spalte *Win32-Projekt*.

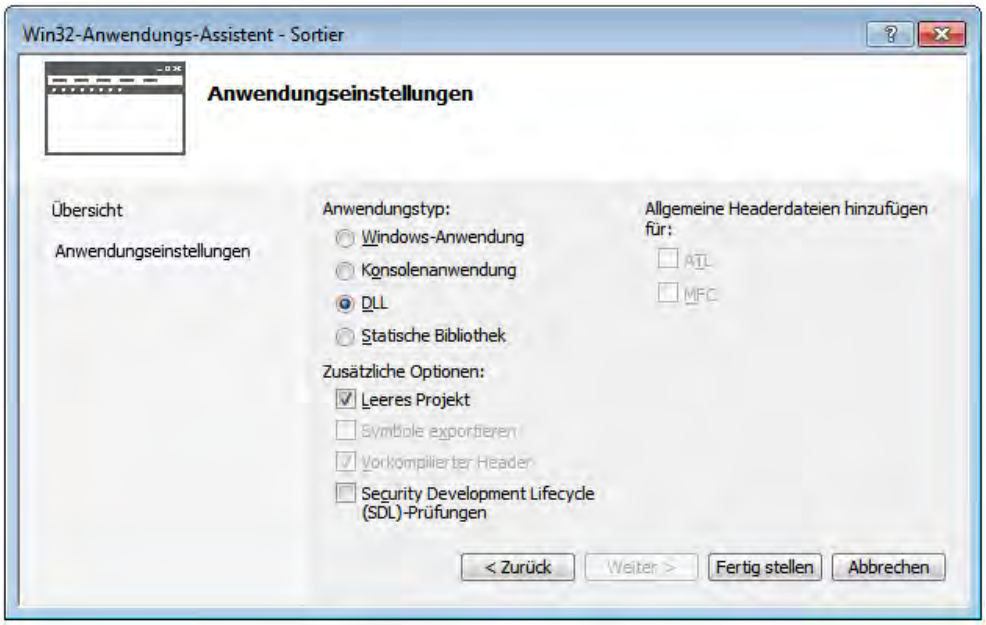


Vergeben Sie den Namen *Sortier* für das Projekt und klicken Sie auf *Ok*.

[1] Das Decompilieren einer .NET-Assembly ist leider sehr leicht möglich, was einen Schutz Ihres geistigen Eigentums schwierig und teuer macht.

HINWEIS: Die Variante *MFC DLL* bringt nur unnötigen Overhead mit sich, den wir nicht benötigen.

In weiteren Verlauf öffnet sich ein neuer Assistent, in welchem wir den Projekttyp spezifizieren. Markieren Sie die Optionen *DLL* und *Leeres Projekt* und klicken Sie auf *Fertig stellen*.



Wählen Sie das Menü *Ansicht/Projektmappen-Explorer*. Dieser präsentiert Ihnen ein noch leeres Projekt, es sind derzeit keine Dateien vorhanden. In ein DLL-Projekt müssen Sie aber zumindest zwei Dateien einbinden: eine *.def*-Datei und eine Datei, in der sich Ihr C-Quellcode (*.cpp)* befindet.

CPP-Datei

Wählen Sie *Projekt/Neues Element hinzufügen...* und dann *C++-Datei(.cpp)* und speichern Sie diese unter ihrem Standardnamen *Quelle.cpp* im Projektverzeichnis ab. Im (noch leeren) Codefenster wartet jetzt einige Arbeit als C++-Programmierer auf uns.

Binden Sie zunächst die Header-Datei *windows.h* zwecks Unterstützung grundlegender Windows-Deklarationen ein:

```
#include <windows.h>
```

Bei der folgenden *DllMain* handelt es sich um die so genannte DLL-Eintrittsprozedur, dem Pendant zum Hauptprogramm einer normalen Anwendung. Mit dieser Prozedur reagieren wir auf das Laden/Entladen unserer DLL durch externe Anwendungen. Dazu ist, wie in den meisten anderen Fällen auch, der Parameter *dwReason* auszuwerten. In unserem Beispiel zeigen wir zwei Meldungsfenster an:

```
BOOL WINAPI DllMain (HINSTANCE hDLL, DWORD dwReason, LPVOID lpReserved)
{
  switch (dwReason) {
   case DLL_PROCESS_ATTACH:
     MessageBoxA( GetFocus(),"DLL wurde geladen","Info", MB_OK | MB_SYSTEMMODAL);
        break;
   case DLL_PROCESS_DETACH:
     MessageBoxA( GetFocus(),"DLL wurde entladen","Info", MB_OK | MB_SYSTEMMODAL);
        break;
  }
  return TRUE;
}
```

Nun endlich kommen wir zu unserer Sortierroutine. Dabei ist zu beachten, dass alle Funktionen, die wir aus der DLL exportieren möchten, eine bestimmte Aufrufkonvention einhalten müssen. Diese beeinflusst die Ausrichtung der übergebenen Argumente auf dem Stack (von rechts nach links), verändert den internen Namen und erweitert die übergebenen Argumente auf ein Vielfaches von 4 Byte. Gleichzeitig ist damit bestimmt, wer für das Aufräumen des Stacks zuständig ist (DLL oder aufrufendes Programm). Für unsere Zwecke ist die *_stdcall*-Syntax geeignet:

```
void _stdcall Sortiere(int a[], int n)
{
   int k, i, j;
   int h;
   k = n/2;
   while (k>0)
   {
     for (i=0; i<n-k; i++)
     {
       j=i;
       while (j>=0 && a[j]>a[j+k])
       {
         h=a[j]; a[j]=a[j+k]; a[j+k]=h;
         j=j-k;
       }
     }
     k=k/2;
   }
}
```

DEF-Datei

Wählen Sie *Projekt/Neues Element hinzufügen...* und dann *DEF-Datei(.def)* und speichern Sie diese unter ihrem Standardnamen *Source.def* im Projektverzeichnis ab.

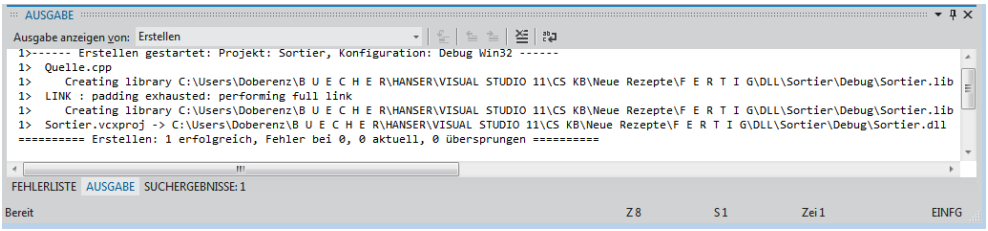
Der Code verweist auf die zu exportierende Routine:

```
LIBRARY   Sortier
EXPORTS
    Sortiere       @1
```

Erstellen der DLL

Öffnen Sie das Ausgabefenster (Menü *Ansicht/Ausgabe*) und klicken Sie anschließend auf den Menüpunkt *Erstellen/Projektmappe erstellen*.

Am unteren Bildrand können Sie jetzt C++-Compiler und -Linker in Aktion erleben:



Die eben erzeugte Dynamic Link Library *Sortier.dll* belassen wir vorläufig im *\Debug*- (bzw. *Release*-) Unterverzeichnis des Projektordners. Später können wir sie in das Windows-System32-Verzeichnis oder, wie in unserem Fall, in dasselbe Verzeichnis wie die noch zu erstellende C#-Anwendung kopieren.

Einbinden der DLL in C#

Schließen Sie Visual Studio und öffnen Sie es erneut. Erstellen Sie eine neue Windows Forms-Anwendung mit dem Namen *Test_DLL* und platzieren Sie auf dem Startformular *Form1* je zwei *ListBox*en und zwei *Button*s (siehe spätere Laufzeitansicht).

```
using System.Windows.Forms;
...
```

Unbedingt erforderlich ist das Einbinden des folgenden Namespace:

```
using System.Runtime.InteropServices;

namespace DLL_Test
{
    public partial class Form1 : Form
    {
```

Die Einbindung der externen DLL und die Deklaration der Routine *Sortiere*:

```
        [DllImport("Sortier.dll")]
        private static extern void Sortiere(int[] a, int n);
```

Für unsere Testzwecke verwenden wir ein Integer-Array mit 100 Feldern:

```
        private const int len = 100;
        private int[] intArray = new int[len];
```

Über die Schaltfläche NEU wird das Array mit Zufallszahlen gefüllt:

```
        private void button1_Click(object sender, EventArgs e)
```

```
        {
            listBox1.Items.Clear();
            Random rnd = new Random();
            for (int i = 0; i < len; i++)
                intArray[i] = rnd.Next(0, len-1);
```

Anzeige des unsortierten Array-Inhalts in der ersten *ListBox*:

```
            for (int i = 0; i < len; i++)
                listBox1.Items.Add(intArray[i].ToString());
        }
```

Die Schaltfläche SORTIER:

```
        private void button2_Click(object sender, EventArgs e)
        {
            listBox2.Items.Clear();
```

Nun der alles entscheidende DLL-Aufruf:

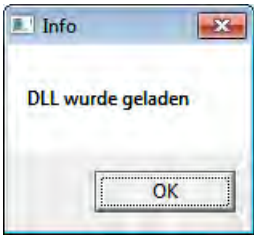
```
            Sortiere(intArray, intArray.Length);
```

Anzeige des sortierten Array-Inhalts in der zweiten *ListBox*:

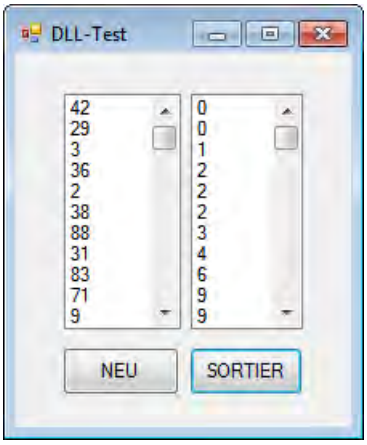
```
            for (int i = 0; i < len; i++)
                listBox2.Items.Add(intArray[i].ToString());
        }
    }
}
```

DLL-Test

- Kompilieren Sie das Projekt *Test_DLL* und kopieren Sie die Datei *Sortier.dll* in das *\Debug*-Unterverzeichnis des Projektordners *Test_DLL*, bzw. in das gleiche Verzeichnis wie auch die Datei *Test_DLL.exe*.
- Durch Klick auf die Schaltfläche NEU füllt sich die linke *ListBox* mit Zufallszahlen. Nachdem Sie aber auf die Schaltfläche SORTIER klicken, wird Ihr Tatendrang höchstwahrscheinlich durch die Meldung *Loader Lock wurde erkannt.* gestoppt. Wie Sie diese Meldung (die in der Regel nicht auf einen echten Fehler hinweist, sondern eine reine Vorsichtsmaßnahme der IDE ist) umgehen können, erläutern wir unter den Bemerkungen am Schluss.
- Für unser einfaches Beispiel kommen Sie aber wahrscheinlich viel eher zum Erfolg, wenn Sie Visual Studio verlassen und die Datei *Test_DLL.exe* direkt durch Doppelklick öffnen. Nach dem Füllen der linken *ListBox* und Klick auf die Schaltfläche SORTIER dürfte zunächst die folgende Meldung erscheinen (siehe folgende Abbildung).



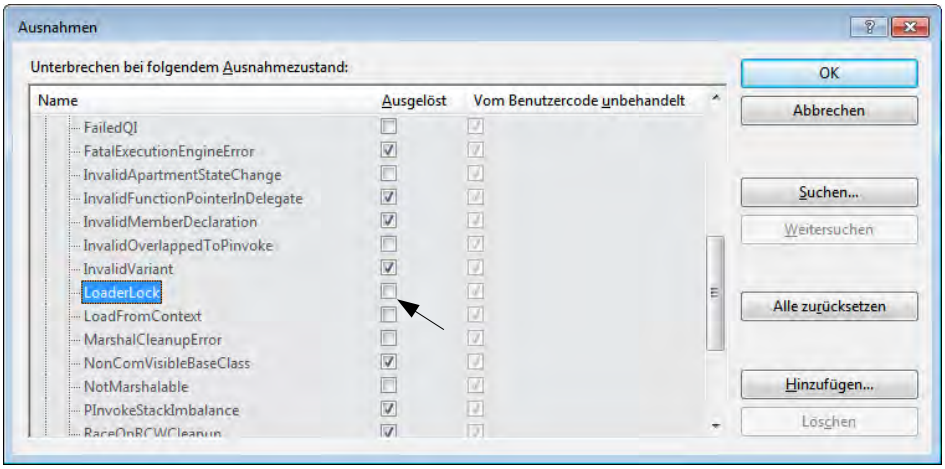
Der Inhalt der rechten *ListBox* überzeugt Sie nun (hoffentlich) vom korrekten Funktionieren Ihrer DLL:



Ein erneuter Klick auf SORTIER führt nicht zu einer erneuten Anzeige obiger Informationsmeldung, da die DLL noch geladen ist. Erst nach dem Beenden der Anwendung taucht unerwartet ein weiterer Dialog auf:

Bemerkungen

- Damit Sie Ihren Programmcode unter Visual Studio debuggen können, gibt es die Möglichkeit, das Anhalten des Debuggers beim Ereignis *LoaderLock* zu deaktivieren. Gehen Sie dazu über das Menü *Debuggen/Ausnahmen...* zum Knoten *Managed Debugging Assistants*. Im Dialog *Ausnahmen* entfernen Sie das Häkchen bei *LoaderLock* in der Spalte *Ausgelöst*.



- Achten Sie bei der DLL-Einbindung immer auf die korrekten Datentypen und sichern Sie Ihr Projekt. Auf mögliche Probleme bei der Übergabe diverser Datentypen können wir hier aus Platzgründen leider nicht näher eingehen.
- Ein DLL-Name sollte eindeutig kennzeichnen, ob es sich um eine 32-Bit- oder eine 64-Bit-Variante handelt. Hängen Sie dazu an den Namen "32" oder "64" an:

```
mstool32.dll
vb40032.dll
```

- Möglichen Konflikten mit verschiedenen DLL-Versionen sollten Sie mit einer einfachen Versionskontrolle vorbeugen. Ihr C#-Programm muss als Erstes prüfen, ob die DLL-Version korrekt ist. Dies könnte zum Beispiel im *Load*-Event des Startformulars geschehen. Damit lässt sich zum einen immer kontrollieren, ob auch die aktuelle Version der DLL verwendet wird, zum anderen wird mit diesem Aufruf die DLL in den Speicher geladen, die Platte "rasselt" also nicht erst später beim eigentlichen Funktionsaufruf los.

```
int _stdcall version (void)
{ return 4711; }
```

- Während der Testphase ist es empfehlenswert, sich die Versionsnummer der DLL anzeigen zu lassen. Erweitern Sie dazu die Eintrittsprozedur wie folgt:

```
BOOL WINAPI DllMain (HINSTANCE hDLL, DWORD dwReason, LPVOID lpReserved)
{
switch (dwReason)
  {
    case DLL_PROCESS_ATTACH:
        MessageBoxA ( GetFocus(),(LPCSTR) "Build 1.241",
        (LPCTSTR) "Info", MB_OK | MB_SYSTEMMODAL);
...
```