Jürgen Kotz

Visual C# 2019 – Grundlagen, Profiwissen und Rezepte

Online-Kapitel:

Teil I: Windows Forms

Teil II: WPF-Bonuskapitel

HANSER

Der Autor:
Jürgen Kotz, München

Bibliografische Information der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Lektorat: Sylvia Hasselbach
Layout: Kösel Media GmbH, Krugzell

„Visual C# 2019 – Grundlagen, Profiwissen und Rezepte“:

Print-ISBN: 978-3-446-45802-4
E-Book-ISBN: 978-3-446-46099-7
E-Pub-ISBN: 978-3-446-46253-3

Inhalt

Teil I: Windows Forms

- Windows Forms-Anwendungen
- Windows-Formulare verwenden
- Komponentenübersicht
- Einführung Grafikausgabe
- Druckausgabe
- Windows Forms-Datenbindung
- Erweiterte Grafikausgabe
- Ressourcen/Lokalisierung
- Komponentenentwicklung

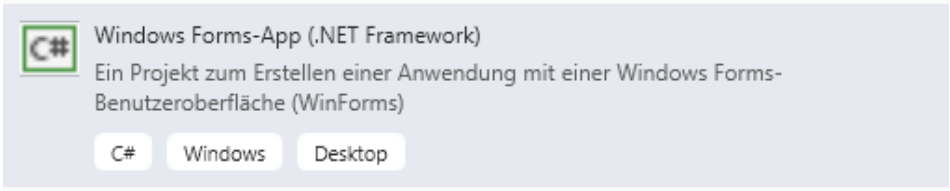
1 Windows Forms-Anwendungen

Nicht mehr ganz frisch, aber dennoch in vielen Projekten noch im Einsatz: die Windows Forms-Anwendungen. Diese entsprechen, im Gegensatz zu den WPF-Anwendungen, in der optischen Aufmachung weitgehend ihren Win32-Pendants. Der große Vorteil dieses Anwendungstyps ist die fast schon unüberschaubare Vielfalt an Komponenten, die es für fast jede Aufgabe gibt.

Ausgerüstet mit grundlegenden C#- und OOP-Kenntnissen werden Sie die folgenden Kapitel in die Lage versetzen, umfangreichere Benutzerschnittstellen eigenständig zu programmieren. Der Schwerpunkt des aktuellen Kapitels liegt zunächst auf dem Anwendungstyp selbst, nachfolgend werden wir uns mit dem Formular als der „Mutter aller Windows Forms-Komponenten“ und seiner grundlegenden Interaktion mit den Steuerelementen beschäftigen. Abschließend werfen wir einen Blick auf die einzelnen visuellen Komponenten bzw. Steuerelemente.

1.1 Grundaufbau/Konzepte

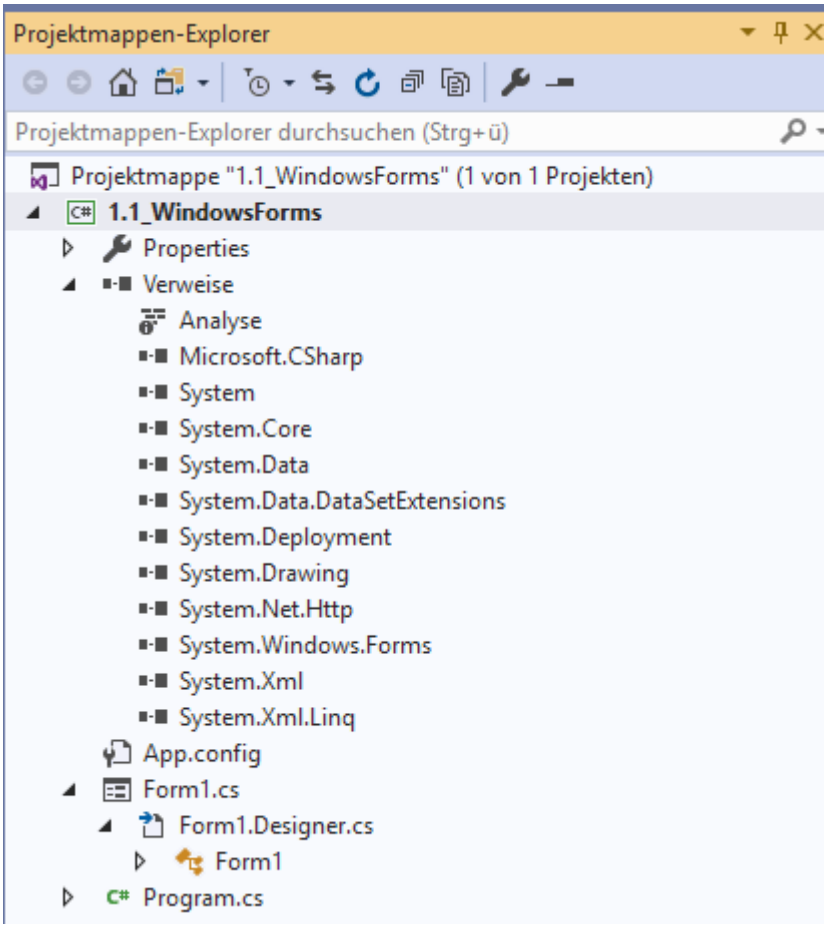
Die „gewöhnliche“ Windows Forms-Anwendung erstellen Sie in Visual Studio über die gleichnamige Vorlage:



Die Abfrage nach dem .NET-Zielframework erfolgt ebenso wie die Frage nach dem Projektnamen und -speicherort im nächsten Dialog.

Bestimmte Anwendungs- bzw. Entwicklungsfeatures, wie z. B. das Entity-Framework, LINQ etc., sind nur verfügbar, wenn Sie eine Framework-Version wählen, die diese Features auch unterstützt. Informieren Sie sich also vorher, welche Features Sie unterstützen müssen.

Nachdem Sie diesen Schritt erfolgreich hinter sich gebracht haben, finden Sie im Designer bereits ein einfaches Windows Form vor. Ein Blick in den Projektmappen-Explorer (siehe folgende Abbildung) zeigt jedoch noch eine Reihe weiterer Dateien, mit denen wir uns im Folgenden intensiver beschäftigen wollen.



Die Anwendung selbst ist zu diesem Zeitpunkt bereits ausführbar, ein Klick auf *F5* und Sie können sich davon überzeugen. Doch nach diesem Blick auf die Bühne wollen wir nun zunächst in den „Technik-Keller“ hinabsteigen, bevor wir uns in den weiteren Kapiteln mit der „Requisite“, das heißt der optischen Gestaltung der Anwendung, abmühen wollen.

1.1.1 Das Hauptprogramm – Program.cs

Der Einstiegspunkt für unsere Anwendung findet sich, wie Sie es sicher schon vermutet haben, in der Datei *Program.cs*. Der bereits vorliegende Code enthält neben diversen Namespace-Importen eine Klasse *Program* mit der statischen Eintrittsmethode *Main*.

Beispiel 1.1: *Program.cs*

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Diagnostics;

namespace _1._1_WindowsForms
{
    static class Program
    {
        /// <summary>
        /// Der Haupteinstiegspunkt für die Anwendung.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

static void Main()

Im einfachsten Fall enthält die *Main*-Methode lediglich drei Methodenaufrufe (auf die Details kommen wir in den folgenden Abschnitten zu sprechen). Alternativ können Sie jedoch auch eine *Main*-Methode mit der Rückgabe eines *int*-Werts und/oder der Übergabe eines *string*-Arrays realisieren Über dieses String-Array können Sie die Kommandozeilenparameter des Programms auswerten.

Beispiel 1.2: Auswerten von Kommandozeilenparametern

C#

```
...
        static void Main(string[] args)
        {
            foreach (string arg in args)
            {
                Debug.WriteLine(arg);
            }
...
```

HINWEIS: Günstiger ist die Auswertung der Kommandozeilenparameter jedoch über die statische *GetCommandLineArgs*-Methode der *Environment*-Klasse. Diese steht jederzeit zur Verfügung und erfordert nicht schon beim Programmstart eine entsprechende Auswertung.

Beispiel 1.3: Verwendung von *GetCommandLineArgs*

C#

```
...
      foreach (string arg in Environment.GetCommandLineArgs())
      {
          Debug.WriteLine(arg);
      }
...
```

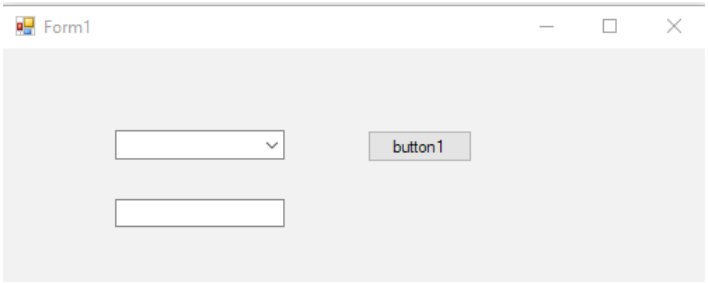
Kommen wir jetzt zum Inhalt der `Main`-Methode.

Application.EnableVisualStyles

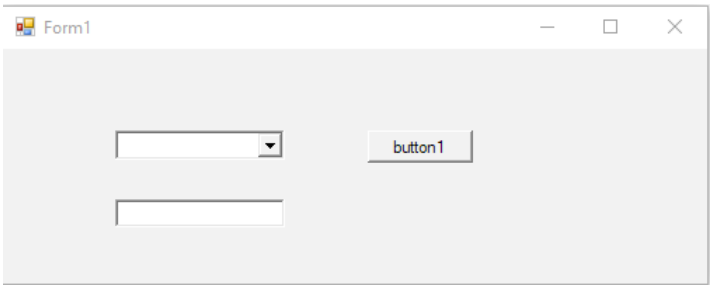
Sicher ist auch Ihnen nicht verborgen geblieben, dass sich seit Windows XP ein neues Layout für die einzelnen Windows Forms-Steuerelemente eingebürgert hat. Voraussetzung für deren Nutzung ist der entsprechende Aufruf der *Application.EnableVisualStyles*-Methode.

Wie sich der Aufruf auf ein fiktives Formular auswirkt, zeigen die beiden folgenden Abbildungen.

Mit *EnableVisualStyles*-Aufruf:



Ohne *EnableVisualStyles*-Aufruf:



Application.SetCompatibleTextRenderingDefault(false)

Mit Einführung von Windows Forms 2.0 wurde die Ausgabe von Text teilweise umgestellt. Da diese Variante nicht hundertprozentig kompatibel mit den bisherigen Ausgabeergebnissen ist, wurde für einzelne Controls eine Eigenschaft *UseCompatibleTextRendering* definiert,

mit der bestimmt wird, ob die alte Variante (1.0 bzw. 1.1) oder die neuere Variante (ab 2.0) verwendet werden soll. Über die Methode *Application.SetCompatibleTextRenderingDefault* wird dieser Wert für alle Controls der Anwendung voreingestellt.

Lange Rede kurzer Sinn: Ist der übergebene Wert *true*, verwenden die Steuerelemente GDI für die Textausgabe, andernfalls GDI+.

Application.Run(new Form1())

Und damit sind wir auch schon beim eigentlichen Mittelpunkt eines jeden Windows Forms-Programms angekommen. *Application.Run* erstellt die für jede Windows-Anwendung lebensnotwendige Nachrichtenschleife, die beispielsweise Maus- und Tastaturereignisse, aber auch Botschaften des Systems oder anderer Anwendungen, entgegennimmt.

Die zweite Aufgabe dieser statischen Methode ist das Instanziieren des Hauptformulars und dessen Anzeige. Wird dieses Fenster geschlossen, ist automatisch auch die Nachrichtenschleife der Anwendung beendet, das Programm terminiert. Dies gilt auch, wenn Sie aus dem Hauptformular heraus (dies ist zunächst *Form1*) weitere Formulare öffnen.

Besteht Ihre Anwendung aus mehr als nur einem Formular, stehen Sie sicher auch vor der Frage, welches Formular denn das Hauptformular bzw. das Startobjekt sein soll. Standardmäßig ist nach dem Erstellen eines Windows Forms-Projekts *Form1* als Startobjekt definiert. Nachträglich können Sie jederzeit auch ein anderes Fenster oder auch eine spezielle Prozedur als Startobjekt festlegen. Durch Editieren des von Visual Studio automatisch generierten Konstruktoraufrufs zum Erzeugen des Startformulars *Form1* kann ein beliebiges anderes Formular der Anwendung zum Startformular gekürt werden.

Nicht in jedem Fall ist es jedoch erwünscht, dass gleich ein Formular beim Start der Anwendung angezeigt wird. Sollen beispielsweise Übergabeparameter ausgewertet werden, ist es häufig günstiger, zunächst diese zu bearbeiten und dann erst das geeignete Formular anzuzeigen. In diesem Fall ändern Sie einfach obigen Abschnitt entsprechend Ihren Wünschen.

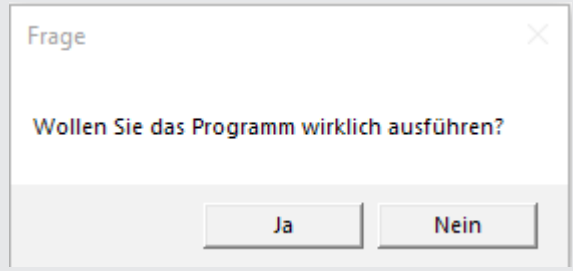
Beispiel 1.4: Eine *Main*-Methode, die zunächst eine *MessageBox*-Abfrage startet

C#

```
...
[STAThread]
static void Main()
{
    if (MessageBox.Show("Wollen Sie das Programm wirklich ausführen?", "Frage",
                        MessageBoxButtons.YesNo) == DialogResult.Yes)
    {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new Form1());
    }
}
```

Ergebnis

Ein Start der Anwendung hat jetzt zur Folge, dass zunächst die *MessageBox*-Abfrage aufgerufen wird. Mit dem Druck auf den *Ja*-Button wird der Anwendung *Form1* (Sie können natürlich auch ein beliebiges anderes Formular auswählen) als Hauptformular zugewiesen und ausgeführt. Andernfalls endet die Programmausführung an dieser Stelle.



1.1.2 Die Oberflächendefinition – Form1.Designer.cs

Eigentlich nicht für Ihre Blicke bestimmt, fristet die Datei *Form1.Designer.cs* ein relativ unbeachtetes Dasein. Doch dieser Eindruck täuscht, enthält doch diese Datei die eigentliche Klassendefinition von *Form1* inklusive Initialisierung der Oberfläche. Gerade der letzte Punkt ist besonders wichtig: Jede Komponente, die Sie im Formulardesigner in das Formular einfügen und konfigurieren, wird in dieser Klassendefinition (Methode *InitializeComponent*) instanziiert und konfiguriert.

Beispiel 1.5: *Form1.Designer.cs*

C#

```
namespace _1._1_WindowsForms
{
    partial class Form1
    {
        private System.ComponentModel.IContainer components = null;

        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Vom Windows Form-Designer generierter Code
```

Die folgende Methode sollten Sie keinesfalls selbst editieren[1]:

```
        private void InitializeComponent()
        {
```

[1] Mit einer Ausnahme: Haben Sie später einmal einen oder mehrere Eventhandler in der Datei *Form1.cs* gelöscht, können Sie deren Zuweisung zu den einzelnen Ereignissen am bequemsten in dieser Datei aufheben. Die betreffenden Zeilen sind durch die „beliebte“ rote Wellenlinie des Editors gekennzeichnet, es genügt, wenn Sie diese Zeilen einfach löschen.

```
            this.comboBox1 = new System.Windows.Forms.ComboBox();
            this.textBox1 = new System.Windows.Forms.TextBox();
            this.button1 = new System.Windows.Forms.Button();
            this.SuspendLayout();
            //
            // comboBox1
            //
            this.comboBox1.FormattingEnabled = true;
            this.comboBox1.Location = new System.Drawing.Point(80, 58);
            this.comboBox1.Name = "comboBox1";
            this.comboBox1.Size = new System.Drawing.Size(121, 21);
            this.comboBox1.TabIndex = 0;
            //
            // textBox1
            //
            this.textBox1.Location = new System.Drawing.Point(80, 107);
            this.textBox1.Name = "textBox1";
            this.textBox1.Size = new System.Drawing.Size(121, 20);
            this.textBox1.TabIndex = 1;
            //
            // button1
            //
            this.button1.Location = new System.Drawing.Point(260, 58);
            this.button1.Name = "button1";
            this.button1.Size = new System.Drawing.Size(75, 23);
            this.button1.TabIndex = 2;
            this.button1.Text = "button1";
            this.button1.UseVisualStyleBackColor = true;
            //
            // Form1
            //
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(502, 167);
            this.Controls.Add(this.button1);
            this.Controls.Add(this.textBox1);
            this.Controls.Add(this.comboBox1);
            this.Name = "Form1";
            this.Text = "Form1";
            this.Load += new System.EventHandler(this.Form1_Load);
            this.ResumeLayout(false);
            this.PerformLayout();
        }
```

Ab hier folgen später die Initialisierungen der einzelnen Formularkomponenten:

```
        #endregion
        private System.Windows.Forms.ComboBox comboBox1;
        private System.Windows.Forms.TextBox textBox1;
        private System.Windows.Forms.Button button1;
    }
}
```

Haben Sie sich obiges Listing genauer angesehen, wird Ihnen aufgefallen sein, dass die Klasse *Form1* als *partial* deklariert ist. Dies ermöglicht es, den Code der Klassendefinition auf mehrere Dateien aufzuteilen, was in unserem Fall bedeutet, dass wir in *Form1.cs* den Rest der Klasse *Form1* vorfinden.

1.1.3 Die Spielwiese des Programmierers – Form1.cs

Jetzt kommen wir zum eigentlichen Tummelplatz für Ihre Aktivitäten als Programmierer. Visual Studio hat bereits ein kleines Codegerüst vorbereitet, das aus dem Import diverser Namespaces sowie der restlichen Klassendefinition von *Form1* inklusive Konstruktor besteht:

Beispiel 1.6: *Form1.cs*

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace _1._1_WindowsForms
{
    public partial class Form1 : Form
    {
```

Der Konstruktor ist dafür verantwortlich, die Methode *InitializeComponent* aufzurufen, diese initialisiert und parametriert die einzelnen Windows Forms-Komponenten:

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

Die Definition des *Button*, den wir schon testweise hinzugefügt haben, wurde in der Datei *Form1.Designer.cs* abgelegt. Den Code für die Ereignisbehandlung definieren Sie in *Form1.cs*.

Beispiel 1.7: Code nach Einfügen eines Buttons

C#

In *Form1.cs*:

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

1.1.4 Die Datei AssemblyInfo.cs

Verschiedene allgemeine Informationen, wie z. B. der Titel der Anwendung oder das Copyright, können als Attribute in die Datei *AssemblyInfo.cs* eingetragen werden. Zugriff auf diese Datei erhalten Sie über den Projektmappen-Explorer (unter *Properties*).

Beispiel 1.8: Auszug aus *AssemblyInfo.cs*

C#

```
...
[assembly: AssemblyTitle("1.1_WindowsForms")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("1.1_WindowsForms")]
[assembly: AssemblyCopyright("Copyright ©  2019")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
...
```

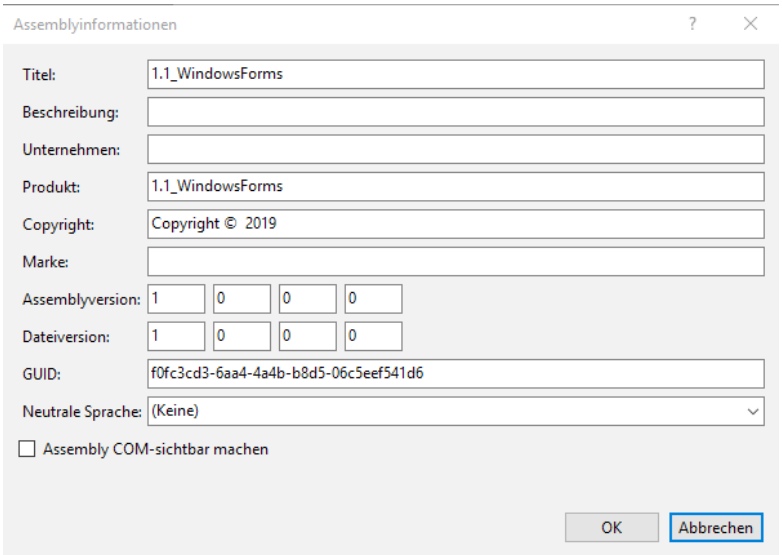
Zur Laufzeit können Sie auf die Einträge über bestimmte Eigenschaften des *Application*-Objekts zugreifen.

Beispiel 1.9: Anzeige des Eintrags unter *AssemblyProduct* in einem Meldungsfenster

C#

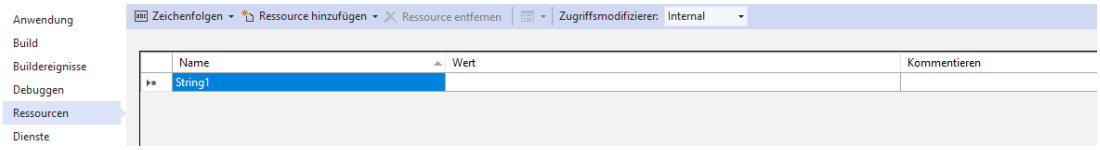
```
MessageBox.Show(Application.ProductName);
```

Editieren lassen sich die Einträge am einfachsten über den folgenden Dialog, den Sie über *Projekt|Eigenschaften|Assemblyinformationen* oder auch durch einen Doppelklick auf *Properties* im Projektmappenexplorer im Projektmappenexplorer erreichen und dann einen Klick aus *Assemblyinformationen...*:



1.1.5 Resources.resx/Resources.Designer.cs

Beide Dateien stellen das Abbild der unter *Projekt|Eigenschaften|Ressourcen* definierten Ressourcen dar. Die XML-Datei *Resources.resx* enthält die eigentlichen Definitionen, die Sie mit dem in der folgenden Abbildung gezeigten Editor erstellen können.



Hier können Sie sehr komfortabel nahezu beliebige Ressourcen (Bilder, Zeichenketten, Audiodateien, …) zu Ihrem Projekt hinzufügen.

Im Gegensatz dazu stellt die Datei *Resources.Designer.cs* für die Ressourcen entsprechende Mapperklassen bereit, die den späteren Zugriff auf die einzelnen Einträge vereinfachen und typisieren. Verantwortlich dafür ist die *Properties.Resources*-Klasse:

Syntax:

```
<Projekt Namespace>.Properties.Resources.<ResourceName>
```

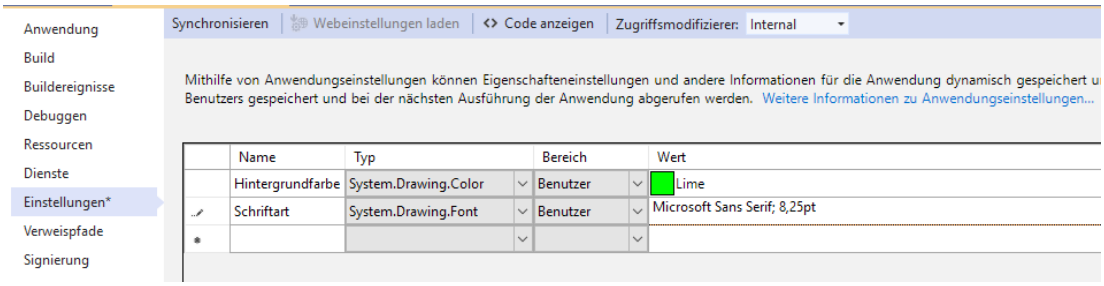
Beispiel 1.10: Die Bildressource *BeimChef.bmp* wird in einer *PictureBox* angezeigt.

C#

```
PictureBox1.Image = Properties.Resources.BeimChef;
```

1.1.6 Settings.settings/Settings.Designer.cs

Diese beiden Dateien sind zur Entwurfszeit für die Verwaltung der Programmeinstellungen verantwortlich. Die XML-Datei *Settings.Settings* enthält die eigentlichen Werte, die Sie jedoch nicht direkt zu bearbeiten brauchen, sondern unter Verwendung eines komfortablen Editors:



Für jede Einstellung können Sie *Name*, *Typ*, *Bereich* und *Wert* festlegen. Wenn Sie den Bereich als *Anwendung* spezifizieren, wird diese Einstellung später (Laufzeit) in der Konfigu-

rationsdatei <*Anwendungsname.exe*>*.config* unter dem <*applicationSettings*>-Knoten gespeichert. Falls Sie *Benutzer* wählen, erfolgt die Ablage unterhalb des <*userSettings*>-Knotens.

Die Datei *Settings.Designer.cs* stellt für die oben definierten Einträge eine Mapperklasse bereit, die den späteren Zugriff auf die einzelnen Einträge vereinfacht und vor allem typisiert.

Beispiel 1.11: *Settings.Designer.cs*

C#

```
namespace _1._1_WindowsForms.Properties {

   [global::System.Runtime.CompilerServices.CompilerGeneratedAttribute()]
   [global::System.CodeDom.Compiler.GeneratedCodeAttribute
            ("Microsoft.VisualStudio.Editors.SettingsDesigner.
              SettingsSingleFileGenerator", "16.1.0.0")]
    internal sealed partial class Settings :
             global::System.Configuration.ApplicationSettingsBase {

        private static Settings defaultInstance =
               ((Settings)(global::System.Configuration.
                 ApplicationSettingsBase.Synchronized(new Settings())));

        public static Settings Default {
            get {
                return defaultInstance;
            }
        }

        [global::System.Configuration.UserScopedSettingAttribute()]
        [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
        [global::System.Configuration.DefaultSettingValueAttribute("Lime")]
        public global::System.Drawing.Color Hintergrundfarbe {
            get {
                return ((global::System.Drawing.Color)
                         (this["Hintergrundfarbe"]));
            }
            set {
                this["Hintergrundfarbe"] = value;
            }
        }

        [global::System.Configuration.UserScopedSettingAttribute()]
        [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
        [global::System.Configuration.DefaultSettingValueAttribute
                 ("Microsoft Sans Serif, 8.25pt")]
        public global::System.Drawing.Font Schriftart {
            get {
                return ((global::System.Drawing.Font)(this["Schriftart"]));
            }
            set {
                this["Schriftart"] = value;
            }
        }
    }
}
```

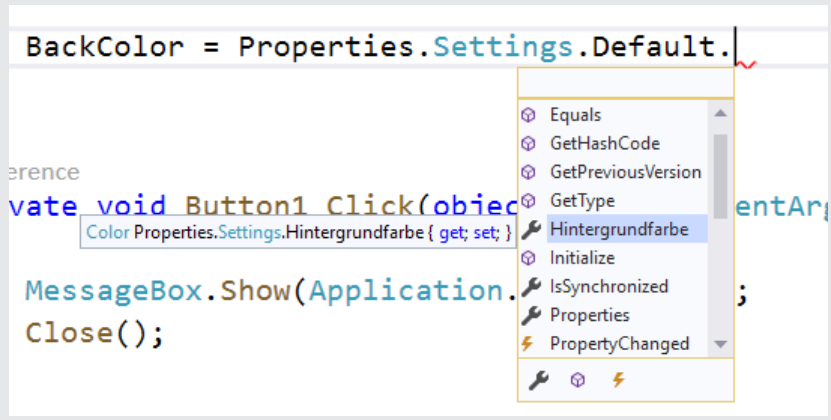
Der Zugriff auf die eingetragenen Einstellungen ist – ähnlich dem Zugriff auf Ressourcen – einfach über die *Properties.Settings*-Klasse möglich.

Syntax:

```
<Projekt Namespace>.Properties.Settings.Default.<SettingsName>
```

Beispiel 1.12: Verwendung der Einstellungen

C#



Doch auch das Zurückschreiben von Nutzereinstellungen in die Datei ist über obige Klasse problemlos möglich.

Beispiel 1.13: Einstellungsänderungen in die Config-Datei zurückschreiben

C#

```
Properties.Settings.Default.Hintergrundfarbe = Color.AliceBlue;
Properties.Settings.Default.Save();
```

HINWEIS: Da auch der spätere Programmbenutzer die zur Assembly mitgegebene XML-Konfigurationsdatei (z. B. *<Anwendungsname>.exe.config*) editieren kann, ergeben sich einfache Möglichkeiten für nachträgliche benutzerspezifische Anpassungen, ohne dazu das Programm erneut kompilieren zu müssen.

1.1.7 Settings.cs

Nanu, hatten wir diese Datei nicht gerade? Nein, hierbei handelt es sich quasi um den zweiten Teil der Klassendefinition von *Settings* aus *Settings.Designer.cs*, was dank partieller Klassendefinition problemlos möglich ist. Im Normalfall dürften Sie diese Datei nicht zu sehen bekommen, aber wenn Sie sich die Mühe machen, über den Designer den Button *Code anzeigen* anzuklicken, findet sich kurz darauf auch diese Datei in Ihrem Projekt.

Beispiel 1.14: *Setting.cs*

C#

```
namespace _1._1_WindowsForms.Properties {

    // Diese Klasse ermöglicht die Behandlung bestimmter
    // Ereignisse der Einstellungsklasse:
    // Das SettingChanging-Ereignis wird ausgelöst, bevor der
    // Wert einer Einstellung geändert wird.
    // Das PropertyChanged-Ereignis wird ausgelöst,
    // nachdem der Wert einer Einstellung geändert wurde.
    // Das SettingsLoaded-Ereignis wird ausgelöst,
    // nachdem die Einstellungswerte geladen wurden.
    //  Das SettingsSaving-Ereignis wird ausgelöst,
    //  bevor die Einstellungswerte gespeichert werden.
    internal sealed partial class Settings {

        public Settings() {
            // Heben Sie die Auskommentierung der unten
            // angezeigten Zeilen auf, um Ereignishandler zum Speichern
            // und Ändern von Einstellungen hinzuzufügen:
            //
            // this.SettingChanging += this.SettingChangingEventHandler;
            //
            // this.SettingsSaving += this.SettingsSavingEventHandler;
            //
        }

        private void SettingChangingEventHandler(object sender,
                  System.Configuration.SettingChangingEventArgs e) {
            // Fügen Sie hier Code zum Behandeln
            // des SettingChangingEvent-Ereignisses hinzu.
        }

        private void SettingsSavingEventHandler(object sender,
                     System.ComponentModel.CancelEventArgs e) {
            // Fügen Sie hier Code zum Behandeln
            // des SettingsSaving-Ereignisses hinzu.
        }
    }
}
```

Mit den in dieser Datei bereitgestellten Eventhandler-Rümpfen bietet sich die Möglichkeit, auf das Laden, Ändern und Sichern von Einstellungen per Ereignis zu reagieren. Es genügt, wenn Sie die entsprechenden Kommentare im Konstruktor entfernen und den gewünschten Code in die Eventhandler eintragen.

1.2 Ein Blick auf die Application-Klasse

Die *Application*-Klasse aus dem Namespace *System.Windows.Forms* ist für den Programmierer von zentraler Bedeutung und unser Kapitel über die Grundlagen von Windows Forms-Anwendungen wäre höchst unvollständig, würden wir auf diese wichtige Klasse nur ganz am Rand eingehen. Den ersten Kontakt mit ihr hatten Sie bereits in der *Main*-Methode.

Die *Application*-Klasse bietet einige interessante statische Eigenschaften und Methoden, von denen wir hier nur die wichtigsten kurz vorstellen wollen:

1.2.1 Eigenschaften

Einige wichtige Eigenschaften von *Application*:

Eigenschaft	Beschreibung
AllowQuit	… darf die Anwendung beendet werden (readonly)?
CommonAppData-Path	… ein gemeinsamer Anwendungspfad
CommonAppData-Registry	… ein gemeinsamer Registry-Eintrag
CompanyName	… der in den Anwendungsressourcen angegebene Firmenname
CurrentCulture	… ein Objekt mit den aktuellen Kulturinformation (Kalender, Währung)
ExecutablePath	… der Name der Anwendung inklusive Pfadangabe
LocalUserAppData-Path	… Anwendungspfad für die Daten des lokalen Benutzers
OpenForms	… Collection der geöffneten Formulare
ProductName	… der in den Ressourcen angegebene Produktname
ProductVersion	… die in den Ressourcen angegebene Produktversion
StartupPath	… der Anwendungspfad
UserAppDataPath	… Pfad für die Anwendungsdaten
UserAppDataRegistry	… Registry-Schlüssel für die Anwendungsdaten

HINWEIS: Einige der obigen Eigenschaften lassen sich den Einträgen der Datei *AssemblyInfo.cs* zuordnen.

Beispiel 1.15: Abrufen aller geöffneten Formulare

C#

```
foreach (Form f in Application.OpenForms)
{
   listBox1.Items.Add(f.Name);
}
```

Beispiel 1.16: Abfrage von *UserAppDataPath* und *UserAppDataRegistry*

C#

```
textBox1.Text = Application.UserAppDataRegistry.ToString();
textBox2.Text = Application.UserAppDataPath.ToString();
```

1.2.2 Methoden

Wichtige Methoden von *Application*:

Methode	Beschreibung
DoEvents	... ermöglicht Verarbeitung ausstehender Windows-Botschaften
EnableVisualStyles	... ermöglicht die Darstellung mit Styles
Exit	... Anwendung beenden
ExitThread	... schließt aktuellen Thread
Run	... startet das Hauptfenster der Anwendung (Beginn einer Nachrichtenschleife für den aktuellen Thread)
Restart	... die Anwendung wird beendet und sofort neu gestartet
SetSuspendState	... versucht das System in den Standbymodus bzw. den Ruhezustand zu versetzen

Den Einsatz der *Run*-Methode haben wir im Unterkapitel 1.1.1 in Abschnitt „Application.Run(new Form1())“ besprochen. Wie Sie *DoEvents* richtig verwenden, zeigen wir Ihnen im folgenden Beispiel.

Beispiel 1.17: Eine zeitaufwendige Berechnungsschleife blockiert die Anzeige der Uhrzeit. Durch Aufrufen von *DoEvents* läuft die Uhr auch während der Berechnung weiter.

C#

Zunächst wird die „Uhr“ programmiert (vorher haben Sie eine *Timer*-Komponente in das Komponentenfach gezogen: `Interval = 1000`, `Enabled = true`):

```
private void Timer1_Tick(object sender, EventArgs e)
{
    label1.Text = DateTime.Now.ToLongTimeString();
}
```

Die Berechnungsschleife wird gestartet:

```
private void Button1_Click(object sender, EventArgs e)
{
    double a = 0;
    for (int i = 0; i < 100000000; i++)
    {
        a = Math.Sin(i) + a;
```

Nach jedem 1000ten Schleifendurchlauf wird CPU-Zeit freigegeben, sodass auf andere Ereignisse reagiert werden kann:

```
        if (i % 1000 ==0)
        {
            Application.DoEvents();
        }
```

Wenn Sie obige Anweisung auskommentieren, „steht“ die Uhr für die Zeit der Berechnung.

```
    }
```

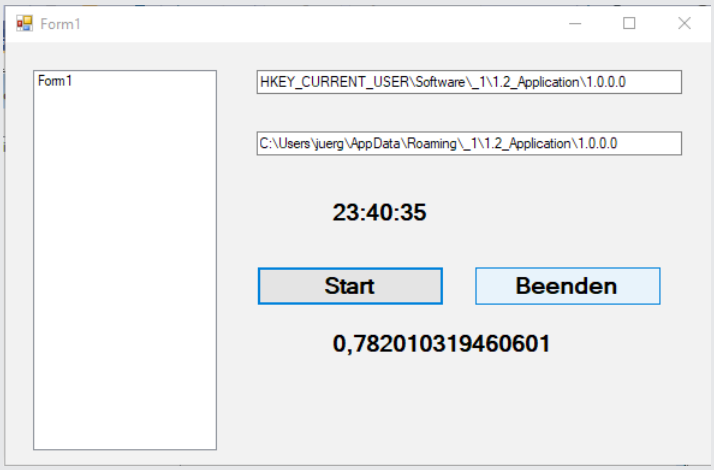
Die Ergebnisanzeige:

```
        label2.Text = a.ToString();
    }
```

Anwendung beenden:

```
    private void button2_Click(object sender, EventArgs e)
    {
        Application.Exit();
    }
}
```

Ergebnis



Eine recht effiziente Art, für Ruhe unter dem Schreibtisch zu sorgen, zeigt das folgende kurze Beispiel.

Beispiel 1.18: Den PC in den Ruhezustand versetzen

C#

```
...
    Application.SetSuspendState(PowerState.Suspend,false,true);
...
```

1.2.3 Ereignisse

Last but not least verfügt die *Application*-Klasse auch über diverse Ereignisse:

Ereignis	Beschreibung
ApplicationExit	... wenn die Anwendung beendet werden soll
EnterThreadModal	... vor dem Eintritt der Anwendung in den modalen Zustand
Idle	... die Applikation hat gerade nichts zu tun

LeaveThreadModal	... der modale Zustand wird verlassen
ThreadException	... eine nicht abgefangene Threadausnahme ist aufgetreten
ThreadExit	... ein Thread wird beendet. Handelt es sich um den Hauptthread der Anwendung, wird zunächst dieses Ereignis und anschließend *ApplicationExit* ausgelöst.

1.3 Allgemeine Eigenschaften von Komponenten

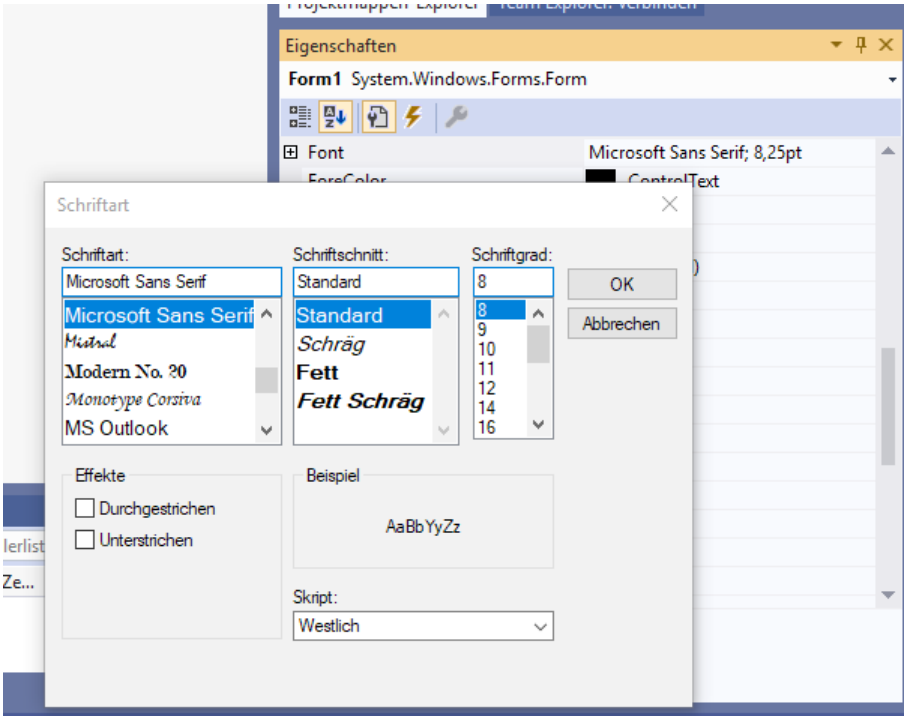
Es gibt eine Vielzahl von Standard-Properties über die (fast) alle Komponenten verfügen. Auf viele Eigenschaften kann nur zur Entwurfszeit, auf andere erst zur Laufzeit zugegriffen werden. Das ist auch der Grund, warum Letztere nicht im Eigenschaften-Fenster der Entwicklungsumgebung zu finden sind. Die Eigenschaften können zur Entwurfszeit (E) und/oder zur Laufzeit (L) verfügbar sein (r = nur lesbar).

Eigenschaft	Erläuterung	E	L
Anchor	Ausrichtung bezüglich des umgebenden Objekts	x	x
BackColor	Hintergrundfarbe	x	x
BorderStyle	Art des Rahmens	x	r
Cursor	Art des Cursors über dem Steuerelement	x	x
CausesValidation	siehe *Validate*-Ereignis		
ContextMenu	Auswahl eines Kontextmenüs	x	x
Dock	Andockstellen bezüglich des umgebenden Objekts	x	x
Enabled	aktiv/nicht aktiv	x	x
Font	Schriftattribute (Namen, Größe, fett, kursiv etc.)	x	x
ForeColor	Vordergrundfarbe, Zeichenfarbe	x	x
Handle	Fensterhandle, an welches das Control gebunden ist		r
Location	Position der linken oberen Ecke	x	x
Locked	Sperren des Steuerelements gegen Veränderungen	x	
Modifiers	Zugriffsmodifizierer (*private*, *public*, ...)	x	x
Name	Bezeichner	x	
TabIndex	Tab-Reihenfolge	x	x
TabStop	Tabulatorstopp Ja/Nein	x	x
Tag	Hilfseigenschaft (speichert Info-Text)	x	x
Text	Beschriftung oder Inhalt	x	x
Visible	Sichtbar Ja/Nein	x	x

Einige dieser Eigenschaften bedürfen einer speziellen Erläuterung.

1.3.1 Font

Mit dieser Eigenschaft (die in Wirklichkeit ein Objekt ist) werden Schriftart, Schriftgröße und Schriftstil der *Text*-Eigenschaft eingestellt (gilt nicht für die Titelleiste eines Formulars). Die Einstellung zur Entwurfszeit ist kein Problem, da ja zusätzlich auch ein komfortabler Fontdialog (der von Windows) zur Verfügung steht.



Der Zugriff auf die *Font*-Eigenschaften per Programmcode ist aber nicht mehr ganz so einfach.

Beispiel 1.19: Zugriff auf die *Font*-Eigenschaften

C#

Sie können zwar die Schriftgröße einer *TextBox* lesen:

```
MessageBox.Show(textBox1.Font.Size.ToString();   // zeigt z.B. 8,25
```

Ein direktes Zuweisen/Ändern der Schriftgröße ist allerdings nicht möglich, da diese Eigenschaft schreibgeschützt ist:

```
textBox1.Font.Size = 12;                  // Fehler (read only)
```

Um die Schrifteigenschaften zu ändern, müssen Sie ein neues *Font*-Objekt erzeugen.

Beispiel 1.20: Ändern der Schriftgröße einer *TextBox*

C#

```
textBox1.Font = new Font(textBox1.Font.Name, 12);
```

Obiges Beispiel zeigt allerdings nur einen der zahlreichen Konstruktoren der *Font*-Klasse. Darunter gibt es natürlich auch welche, die das Zuweisen des Schriftstils (fett, kursiv etc.) ermöglichen.

Die möglichen Schriftstile sind in der *FontStyle*-Enumeration definiert (siehe folgende Tabelle):

Schriftstil	Beschreibung
Regular	normaler Text
Bold	fett formatierter Text
Italic	kursiv formatierter Text
Underline	unterstrichener Text
Strikeout	durchgestrichener Text

Beispiel 1.21: Inhalt einer *TextBox* fett formatieren

C#

```
textBox1.Font = new Font(textBox1.Font, FontStyle.Bold);
```

Was aber ist, wenn ein Text gleichzeitig mehrere Schriftstile haben soll? In diesem Fall müssen die einzelnen Schriftstile mit einer bitweisen ODER-Verknüpfung kombiniert werden.

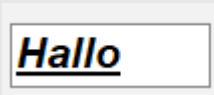
Beispiel 1.22: Mehrere Schriftstile zuweisen

C#

Der Inhalt einer *TextBox* wird mit der Schriftart „Arial“ und einer Schriftgröße von 16pt gleichzeitig fett, kursiv und mit Unterstreichung formatiert.

```
textBox1.Font = new Font(«Arial», 16f, FontStyle.Bold
                  | FontStyle.Underline | FontStyle.Italic);
```

Ergebnis



1.3.2 Tag

Diese Eigenschaft müssen Sie sich wie einen unsichtbaren „Merkzettel“ vorstellen, den man an ein Objekt „anheften“ kann. Im Eigenschaftenfenster erscheint *Tag* zwar als *string*-Datentyp, in Wirklichkeit ist *Tag* aber vom *object*-Datentyp, denn per Programmcode können Sie dieser Eigenschaft einen beliebigen Wert zuweisen.

Beispiel 1.23: Verwendung *Tag*

C#

Sie haben der *Image*-Eigenschaft einer *PictureBox* ein Bild zugewiesen und wollen beim Klick auf die *PictureBox* einen erklärenden Text in einem Meldungsfenster anzeigen lassen:

```
...
pictureBox1.Tag = "Das ist meine Katze!";
...
private void PictureBox1_Click(object sender, EventArgs e)
{
    MessageBox.Show(pictureBox1.Tag.ToString());
    // zeigt "Das ist meine Katze!"
}
```

HINWEIS: Auch Steuerelemente, die im Komponentenfach abgelegt werden, wie *Timer*, *OpenFileDialog* etc. verfügen über eine *Tag*-Eigenschaft.

1.3.3 Modifiers

Beim visuellen Entwurfsprozess werden Steuerelemente vom Designer wie private Felder der *Form*-Klasse angelegt, ein Zugriff von außerhalb ist also zunächst nicht möglich.

Modifiers ist keine Eigenschaft im eigentlichen Sinn, sondern lediglich eine Option der Entwicklungsumgebung, die Sie dann brauchen, wenn Sie von einem Formular aus auf ein Steuerelement zugreifen wollen, das sich auf einem anderen Formular befindet. Angeboten werden die Modifizierer *public*, *protected*, *internal*, *protected internal* und *private*.

Beispiel 1.24: Verwendung von *Modifiers*

C#

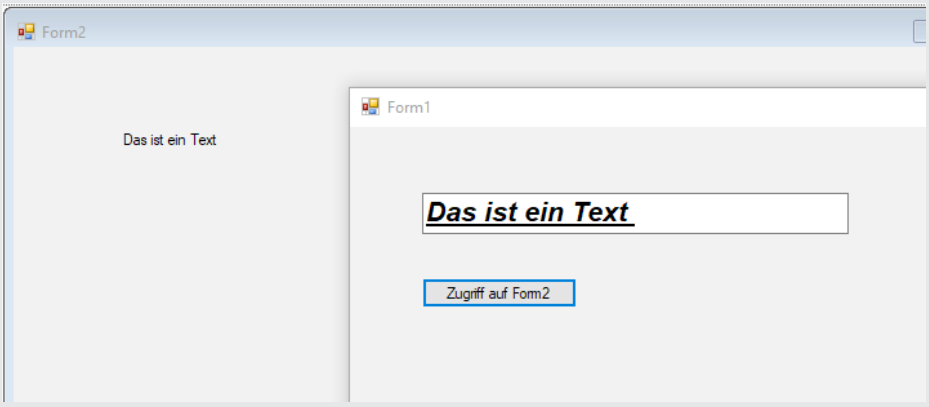
Nach dem Klick auf einen auf *Form1* befindlichen *Button* soll der Inhalt einer auf *Form2* befindlichen *TextBox* in einem *Label* auf *Form1* angezeigt werden. Der folgende Code:

```
public partial class Form1 : Form
{
 ...
    private void Button1_Click(object sender, EventArgs e)
    {
        Form2 f2 = new Form2();
```

```
        // f2.Show();          // nicht notwendig!
        textBox1.Text = f2.label1.Text;
    }
}
```

... funktioniert nur dann, wenn Sie vorher in *Form2* die *Modifiers*-Eigenschaft von *textBox1* auf *public*, *internal* oder *protected internal* gesetzt haben.

Ergebnis



HINWEIS: Der Standardmodifier für Steuerelemente in Visual Basic ist *internal*. Lassen Sie sich dadurch bitte nicht verwirren.

1.4 Allgemeine Ereignisse von Komponenten

Wir wollen uns zunächst nur auf die Ereignisse beschränken, die für die meisten Komponenten gleichermaßen zutreffen, und auf einige grundsätzliche Programmieransätze eingehen.

1.4.1 Die Eventhandler-Argumente

Jedem Ereignis-Handler werden zumindest zwei Parameter übergeben:

- das *sender*-Objekt,
- die eigentlichen Parameter als Objekt mit dem kurzen Namen *e*.

Was können wir mit beiden Werten anfangen?

1.4.2 Sender

Da Ereignis-Handler einen beliebigen Namen erhalten und auch „artfremde“ Komponenten sich ein und denselben Eventhandler teilen können, muss es ein Unterscheidungsmerkmal für den Aufrufer des Ereignisses geben. Über den *sender* ist es möglich, zu entscheiden, welches Objekt das Ereignis ausgelöst hat.

Zunächst werden Sie sicher enttäuscht sein, wenn Sie sich die einzelnen Eigenschaften bzw. Methoden des *sender*-Objekts ansehen. Lediglich eine Methode *GetType* ist zu finden. Doch keine Sorge, typisieren Sie das Objekt, so haben Sie Zugriff auf alle objekttypischen Eigenschaften und Methoden der das Ereignis auslösenden Komponente.

Beispiel 1.25: Ein gemeinsamer Ereignis-Handler (*TextChanged*) für mehrere Textboxen zeigt in der Kopfzeile des Formulars den Inhalt der jeweils bearbeiteten Textbox an.

C#

```
private void TextBox_TextChanged(object sender, EventArgs e)
{
    Text = (sender as TextBox)?.Text;
}
```

Im obigen Beispiel wurde der *as*-Operator zur Typkonvertierung eingesetzt. Aber auch der übliche explizite *()*-Operator ist anwendbar[2]. Falls der Eventhandler durch ein anderes Element wie eine *TextBox* ausgelöst wird, dann schlägt die Konvertierung mit *as* fehl und es wird *null* zurückgegeben. Damit unser Programm nicht abstürzt setzen wir den *Null-Propagator?*.

Beispiel 1.26: Vorgängerbeispiel mit dem *()*-Konvertierungsoperator

C#

```
private void textBox_TextChanged(object sender, EventArgs e)
{
    Text = ((TextBox)sender).Text;
}
```

Im Folgenden werden wir aber den *as*-Operator bevorzugen, da dieser wie bereits erwähnt *null* liefert, falls die Konvertierung misslingt. Beim *()*-Konvertierungsoperator würde hingegen eine Exception ausgelöst.

Das wäre zum Beispiel der Fall, wenn ein *TextChange* von einer *TextBox* und einer *ComboBox* kommt. Spätestens beim Testen der Anwendung gäbe es Ärger, da der Typ nicht immer stimmt. Mit dem *as*-Operator hingegen können Sie jederzeit wie oben gezeigt mit dem Null-Propagator den Fehler vermeiden.

Brauchen Sie unbedingt den speziellen Objekttyp, können Sie zunächst mittels *GetType()* den Typ von *sender* feststellen, um dann, in Abhängigkeit vom Typ, unterschiedliche Aktionen auszuführen.

---

[2] Im Unterschied zum *()*-Konvertierungsoperator ist der *as*-Operator nur für Referenz- bzw. Verweistypen anwendbar, wozu auch alle Controls gehören.

Beispiel 1.27: Verwendung von *sender*

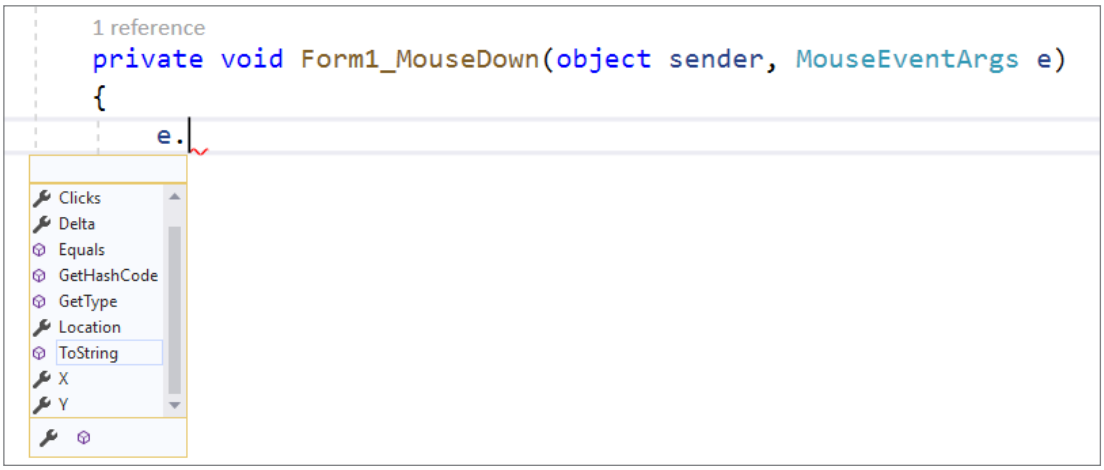
C#

Auch den folgenden Eventhandler teilen sich mehrere *Text*- und *ComboBox*en gemeinsam. Ändert sich der Inhalt einer *TextBox*, so wird dieser in der Titelleiste des Formulars angezeigt. Ändert sich aber die *Text*-Eigenschaft einer *ComboBox*, so wird nur deren Name angezeigt.

```
private void Control_TextChanged(object sender, EventArgs e)
{
    switch (sender.GetType().Name)
    {
        case "TextBox":
            Text = (sender as TextBox).Text;
            break;
        case "ComboBox":
             Text = (sender as ComboBox).Name;
             break;
    }
}
```

1.4.3 Der Parameter e

Was sich hinter dem Parameter *e* versteckt, ist vom jeweiligen Ereignis abhängig. So werden bei einem *MouseDown*-Ereignis unter anderem die gedrückten Tasten und die Koordinaten des Mausklicks im Parameter *e* übergeben:



Teilweise werden über diesen Parameter auch Werte an das aufrufende Programm zurückgegeben.

Beispiel 1.28: Das Schließen des Formulars wird verhindert.

C#

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    e.Cancel = true;
}
```

1.4.4 Mausereignisse

Wenn sich der Mauscursor über einem Objekt befindet, können die folgenden Mausaktivitäten (teilweise mit Übergabeparametern) ausgewertet werden:

Ereignis	... tritt ein, wenn
Click	... auf das Objekt geklickt wird.
DoubleClick	... auf das Objekt doppelt geklickt wird.
MouseDown	... eine Maustaste niedergedrückt wird.
MouseUp	... eine Maustaste losgelassen wird.
MouseMove	... die Maus bewegt wird.
MouseEnter	... wenn die Maus in das Control hineinbewegt wird.
MouseLeave	... wenn die Maus aus dem Control hinausbewegt wird.

Bei einem *MouseDown* können Sie über *e.Button* unterscheiden, welcher Button gerade gedrückt wurde (*Left, Middle, Right*). Gleichzeitig können Sie über *e.x* bzw. *e.y* die Koordinaten bezüglich des jeweiligen Objekts ermitteln.

HINWEIS: Beachten Sie, dass jeder Doppelklick auch ein „normales" Klickereignis auslöst. Man sollte deshalb überlegt zu Werke gehen, wenn für ein Control beide Events gleichzeitig besetzt werden sollen.

Beispiel 1.29: *MouseDown*

C#

Beim Niederdrücken der rechten Maustaste über dem Formular wird eine *MessageBox* erzeugt, wenn sich die Maus in dem durch die Koordinaten 10,10 (linke obere Ecke) und 110,110 (rechte untere Ecke) bezeichneten Rechteck befindet:

```
private void Form1_MouseDown(object sender, MouseEventArgs e)
{
    if ((e.Button == MouseButtons.Right) &&
            new Rectangle(10, 10, 100, 100).Contains(e.X, e.Y))
    {
        MessageBox.Show("Erfolg");
    }
}
```

Beispiel 1.30: Ändern der *TextBox*-Hintergrundfarbe, wenn die Maus darüber bewegt wird.

C#

```
private void TextBox1_MouseEnter(object sender, EventArgs e)
{
    (sender as TextBox).BackColor = Color.Blue;
}

private void TextBox1_MouseLeave(object sender, EventArgs e)
{
    (sender as TextBox).BackColor = Color.White;
}
```

Tastaturereignisse

Wenn ein Steuerelement den Fokus hat, können für dieses Objekt die in der folgenden Tabelle aufgelisteten Keyboard-Events ausgewertet werden.

Ereignis	... tritt ein, wenn
KeyPress	... eine Taste gedrückt wird.
KeyDown	... die Taste nach unten bewegt wird (mit Intervall).
KeyUp	... eine Taste losgelassen wird.

KeyPress registriert das Zeichen der gedrückten Taste, während *KeyDown* und *KeyUp* auf alle Tasten der Tastatur (einschließlich Funktionstasten und Tastenkombinationen mit den Tasten *Umschalt*, *Alt* und *Strg*) reagieren können.

Beispiel 1.31: Verwendung *KeyUp*

C#

Beim Loslassen einer Zifferntaste innerhalb einer *TextBox* wird die Ziffer in eine *ListBox* übernommen (48 ... 57 sind die ANSI-Codes der Ziffern 0 ... 9).

```
private void TextBox2_KeyDown(object sender, KeyEventArgs e)
{
    if ((e.KeyValue > 47) & (e.KeyValue < 58))
    {
        listBox1.Items.Add((e.KeyValue-48).ToString());
    }
}
```

1.4.5 KeyPreview

KeyPreview ist kein Ereignis, sondern eine Formulareigenschaft! *KeyPreview* steht aber im engen Zusammenhang mit den Tastaturereignissen und soll deshalb (ausnahmsweise) bereits an dieser Stelle erwähnt werden. Wenn *KeyPreview* den Wert *false* hat (Voreinstellung), werden die Ereignisse sofort zur Komponente weitergegeben. Hat *KeyPreview* aber den Wert *true*, so gehen, unabhängig von der aktiven Komponente, die Tastaturereignisse

KeyDown, *KeyUp* und *KeyPress* zuerst an das Formular. Erst danach wird das Tastaturereignis an das Steuerelement weitergereicht. Damit kann an zentraler Stelle auf Tastaturereignisse reagiert werden (Vielleicht erinnern Sie sich an die Ereignisabfolge in WPF).

Beispiel 1.32: *KeyPreview*

C#

Entsprechend dem vorhergehenden Beispiel soll beim Drücken einer Zifferntaste diese in eine *ListBox* übernommen werden, in diesem Fall jedoch bei **allen** Controls des aktuellen Fensters.

Setzen Sie im Eigenschaftenfenster (*F4*) die Eigenschaft *KeyPreview* des Formulars auf *true* und erzeugen Sie folgenden Event-Handler:

```
private void Form1_KeyUp(object sender, KeyEventArgs e)
{
    if ((e.KeyValue > 47) & (e.KeyValue < 58))
    {
        listBox1.Items.Add((e.KeyValue - 48).ToString());
    }
    e.Handled = true;
}
```

HINWEIS: Da in diesem Beispiel bei *KeyPreview=true* das Ereignis nur im Event-Handler des Formulars und nicht auch im aktuellen Steuerelement verarbeitet werden soll, haben wir *e.Handled* auf *true* gesetzt. In unserem Beispiel reagieren wir aber im Formular auf das *KeyUp*-Ereignis und in der *TextBox* auf das *KeyDown*, somit werden beide Ereignisse ausgelöst. Das *KeyUp*-Ereignis der *TextBox* wäre jedoch nicht gefeuert worden. Probieren Sie gerne etwas in dem Beispiel herum.

1.4.6 Weitere Ereignisse

Die folgenden Events finden Sie ebenfalls bei einer Vielzahl von Objekten:

Ereignis	… tritt ein, wenn
Change	… der Inhalt der Komponente geändert wird.
Enter	… die Komponente den Fokus erhält.
DragDrop	… das Objekt über der Komponente abgelegt wird.
DragOver	… das Objekt über die Komponente gezogen wird.
HelpRequested	… die Hilfe angefordert wird (*F1*).
Leave	… die Komponente den Fokus verliert.
Paint	… das Steuerelement gezeichnet wird.
Resize	… die Komponente in der Größe verändert wird.
Validate	… der Inhalt von Steuerelementen überprüft wird.

Beispiel 1.33: Erhält die *TextBox* den Eingabefokus, soll sich die Hintergrundfarbe ändern.

C#

```
private void TextBox2_Enter(object sender, EventArgs e)
{
    textBox2.BackColor = Color.Yellow;
}
```

Beim Verlassen stellen wir die normale Farbe ein:

```
private void TextBox2_Leave(object sender, EventArgs e)
{
    textBox2.BackColor = Color.White;
}
```

Im Zusammenhang mit dem Auftreten der Fokus-Ereignisse spielt häufig auch die Reihenfolge eine Rolle:

Enter → *GotFocus* → *Leave* → *Validating* → *Validated* → *LostFocus*

1.4.7 Validätsprüfungen

Das Validate-Ereignis ermöglicht es, zusammen mit der *CausesValidation*-Eigenschaft den Inhalt von Steuerelementen zu prüfen, **bevor** der Fokus das Steuerelement verlässt. Das *Validate*-Ereignis eignet sich besser zum Überprüfen der Dateneingabe als das *LostFocus*-Ereignis, da *LostFocus* erst **nach** dem Verschieben des Fokus eintritt.

Validate wird nur dann ausgelöst, wenn der Fokus in ein Steuerelement wechselt, bei dem die *CausesValidation*-Eigenschaft *true* ist (Standardeinstellung). *CausesValidation* sollte nur bei den Controls auf *false* gesetzt werden, deren Aktivierung keine Validätskontrolle auslösen soll, wie z. B. eine *Abbrechen*- oder eine *Hilfe*-Schaltfläche.

HINWEIS: Ist die Prüfung innerhalb des *Validating*-Events nicht erfolgreich, können Sie mit *e.Cancel* die weitere Ereigniskette (siehe oben) abbrechen.

Beispiel 1.34: *Validating*

C#

Der Fokus wandert nur dann zum nächsten Steuerelement, wenn in die *TextBox* mehr als fünf Zeichen eingegeben werden.

```
private void TextBox2_Validating(object sender, CancelEventArgs e)
{
    if (textBox2.Text.Length < 5)
    {
        MessageBox.Show("Bitte mehr als 5 Zeichen eingeben!");
        e.Cancel = true;
    }
}
```

1.4.8 SendKeys

Mit diesem Objekt werden Tastatureingaben durch den Bediener simuliert, daher ist es zweckmäßig, es bereits an dieser Stelle im Zusammenhang mit Tastaturereignissen zu erwähnen. Zwei Methoden stehen zur Auswahl:

- *Send*
- *SendWait*

Während Erstere sich damit begnügt, die Tastatureingaben einfach an die aktive Anwendung zu senden, wartet *SendWait* auch darauf, dass die Daten verarbeitet werden. Insbesondere bei etwas langsameren Operationen kann es sonst schnell zu einem Fehlverhalten kommen.

Das Argument der beiden Methoden ist eine Zeichenkette. Jede Taste wird dabei durch mindestens ein Zeichen repräsentiert.

HINWEIS: Das Pluszeichen (+), Caret-Zeichen (^) und Prozentzeichen (%) sind für die UMSCHALT-, STRG- und ALT-Taste vorgesehen. Sondertasten sind in geschweifte Klammern einzuschließen.

Beispiel 1.35: *SendKeys*

C#

Die folgende Anweisung sendet die Tastenfolge *Alt+F4* an das aktive Fenster und bewirkt damit ein Schließen der Applikation.

```
private void Button1_Click(object sender, EventArgs e)
{
    SendKeys.Send("%{F4}");
}
```

Häufig soll sich die „Tastatureingabe“ nicht auf das aktuelle Formular, sondern auf das aktive Steuerelement beziehen. Dann muss dieses Steuerelement vorher den Fokus erhalten.

Beispiel 1.36: *SendWait* mit *SetFocus*

C#

Die folgende Sequenz füllt das Textfeld *textBox1* mit den Ziffern 12345678 und setzt danach die Ausführung fort.

```
private void Button1_Click(object sender, EventArgs e)
{
    textBox1.Focus();
    SendKeys.SendWait("12345678");
}
```

HINWEIS: *SendKeys* macht es auch möglich, quasi „wie von Geisterhand“ andere Windows-Programme (z. B. den integrierten Taschenrechner) aufzurufen.

1.5 Allgemeine Methoden von Komponenten

Auch hier nur ein Auszug aus dem reichhaltigen Sortiment:

Methode	Erläuterung
Contains	… kontrolliert, ob ein angegebenes Steuerelement dem aktuellen untergeordnet ist
CreateGraphics	… erzeugt ein *Graphics*-Objekt zum Zeichnen im Steuerelement
DoDragDrop	… beginnt eine Drag&Drop-Operation
FindForm	… ruft das übergeordnete Formular ab
Focus	… setzt den Fokus auf das Objekt
GetContainerControl	… ruft das übergeordnete Steuerelement ab
GetType	… ruft den Typ des Steuerelements ab
Hide	… verbirgt das Steuerelement
Invalidate	… veranlasst das Neuzeichnen des Controls
Refresh	… erneuert den Aufbau des Steuerelements
Scale	… skaliert das Steuerelement
SelectNextControl	… aktiviert das folgende Steuerelement
SetBounds	… setzt die Größe des Steuerelements
Show	… zeigt das Steuerelement nach dem Verbergen wieder an

HINWEIS: Über die an die Methoden zu übergebenden Parameter informieren Sie sich am besten per IntelliSense bzw. in der Dokumentation.

2 Windows Forms-Formulare

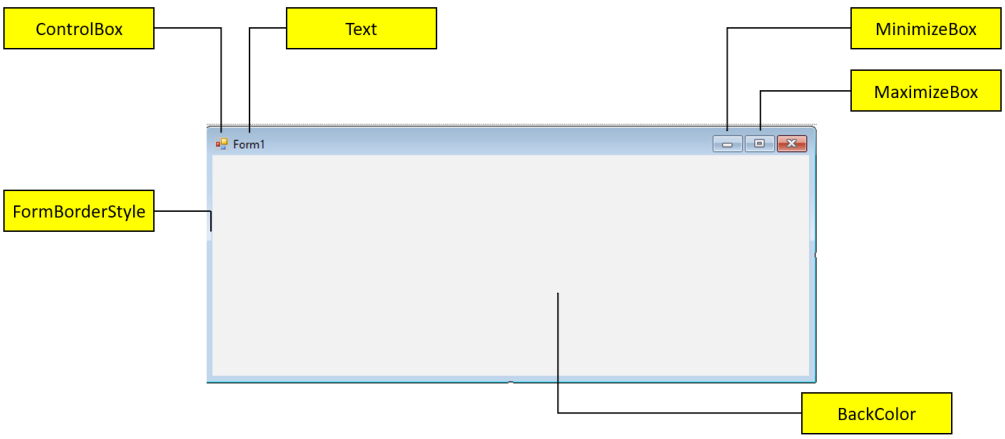
Fast jede Windows Forms-Applikation läuft in einem oder mehreren Fenstern ab, die sozusagen als Container für die Anwendung fungieren. Schon deshalb ist das Formular (*Form*) das wichtigste Objekt, wir werden uns also damit etwas ausführlicher auseinandersetzen müssen.

HINWEIS: Die Ausführungen zu den allgemeinen Eigenschaften/Methoden und Ereignissen aus dem vorhergehenden Kapitel treffen auch auf die Windows-Formulare zu. Sie sollten also das vorhergehende Kapitel bereits durchgearbeitet haben.

2.1 Übersicht

Haben Sie eine neue Windows Forms-App geöffnet, werden Sie bereits von einem ersten Formular im Designer begrüßt.

Wie ein Formular aussieht, brauchen wir sicher nicht weiter zu erklären. Welche Eigenschaften jedoch für das Aussehen verantwortlich sind, soll die folgende Abbildung verdeutlichen:



Für die weitere Orientierung sollen die folgenden Tabellen sorgen, die in Kürze die wichtigsten Eigenschaften, Methoden und Ereignisse des Formulars erläutern. In den weiteren Abschnitten wenden wir uns dann spezifischen Themen rund um das Formular zu.

2.1.1 Wichtige Eigenschaften des Form-Objekts

Eigenschaft	Beschreibung
AcceptButton *CancelButton*	Diese Eigenschaften legen fest, welche Buttons mit den Standardereignissen *Enter*-Taste bzw. *Esc*-Taste verknüpft werden. Diese Funktionalität wird häufig in Dialogboxen verwendet (siehe auch *DialogResult*). Löst der Nutzer eine der beiden Aktionen aus, wird der Ereigniscode der jeweiligen Taste verarbeitet.
ActiveControl	… bestimmt das gerade aktive Control, z. B. `if (sender == this.ActiveControl) berechne();`
ActiveForm	… bestimmt das aktive Formular der Anwendung
ActiveMdiChild	… ermittelt das gerade aktive MDI-Child-Fenster
AutoScroll	… bestimmt, ob das Formular automatisch Scrollbars einfügen soll, wenn der Clientbereich nicht komplett darstellbar ist
BackgroundImage	… eine Grafik für den Hintergrund
ClientSize	… ermittelt die Größe des Formular-Clientbereichs
ContextMenu	… das Kontextmenü des Formulars
ControlBox	… Anzeige des Systemmenüs (*true/false*)
Controls	… eine Collection aller enthaltenen Controls
Cursor	… legt die Cursorform für das aktuelle Formular fest
DesktopBounds	… legt Position und Größe des Formulars auf dem Desktop fest, z. B. `DesktopBounds = new Rectangle.Create(10, 10, 100, 100);`
DesktopLocation	… legt die Position des Formulars fest
DialogResult	… über diesen Wert können Dialog-Statusinformationen an ein aufrufendes Programm zurückgegeben werden, z. B. Form2 f2 = new Form2(); `if (f2.ShowDialog() == DialoResult.Abort) {...}`
DoubleBuffered	… ermöglicht flackerfreie Darstellung von Grafiken
Dock	… setzt die Ausrichtung gegenüber einem übergeordneten Fenster
DockPadding	… setzt den Zwischenraum beim Docking von Controls
FormBorderStyle	… setzt den Formularrahmen. Dieser hat auch Einfluss auf das Verhalten (Tool-Window, Dialog etc.).
HelpButton	… soll der Hilfe-Button angezeigt werden?
Icon	… das Formular-Icon
IsMdiChild	… handelt es sich um ein MDI-Child-Fenster?
IsMdiContainer	… handelt es sich um ein MDI-Container-Fenster?

Location	… die linke obere Ecke des Formulars.
MaximizeBox *MinimizeBox*	… Anzeige der beiden Formular-Buttons (*true/false*)
MaximumSize *MinimumSize*	… setzt maximale bzw. minimale Maße für das Fenster
MdiChildren	… eine Collection der untergeordneten MDI-Child-Fenster
MdiParent	… ermittelt den MDI-Container
Menu	… das Hauptmenü des Formulars
Modal	… wird das Formular (Dialog) modal angezeigt?
Opacity	… Transparenz des Formulars in Prozent
OwnedForms	… eine Collection der untergeordneten Formulare
Owner	… das übergeordnete Formular
ShowInTaskbar	… soll das Formular in der Taskbar angezeigt werden?
Size	… die Formulargröße
StartPosition	… wo wird das Fenster beim ersten Aufruf angezeigt (zentriert etc.)?
Text	… der Text in der Titelleiste
TopMost	… soll das Formular an oberster Position angezeigt werden?
TransparencyKey	… welche Formularfarbe soll transparent dargestellt werden?
Visible	… ist das Formular sichtbar?
WindowState	… ist das Fenster maximiert, minimiert oder normal dargestellt?

Controls-Auflistung

Mit dieser Eigenschaft ist der Zugriff auf alle im Formular vorhandenen Steuerelemente möglich.

Beispiel 2.1: Alle Controls im aktuellen Formular um zehn Pixel nach links verschieben

C#

```
private void Button1_Click(object sender, EventArgs e)
{
    foreach (Control c in Controls)
    {
        c.Left -= 10;
    }
}
```

Der Zugriff auf Elemente der *Controls*-Auflistung eines Formulars ist nicht nur über den Index, sondern auch über den Namen des Elements möglich.

Beispiel 2.2: Zugriff auf ein spezielles Control über dessen Namen

C#

```
TextBox tb = (TextBox)Controls["textBox2"];
MessageBox.Show(tb.Text);
```

2.1.2 Wichtige Ereignisse des Form-Objekts

Neben den bereits im vorhergehenden Kapitel aufgelisteten Events sind für ein Formular die folgenden Events von Bedeutung:

Ereignis	... tritt ein, wenn
Activated	... das Formular aktiviert wird.
FormClosing	... das Formular geschlossen werden soll. Sie können den Vorgang über den Parameter *e.Cancel* abbrechen und so ein Schließen des Formulars verhindern.
FormClosed	... das Formular geschlossen ist.
Deactivate	... ein anderes Formular aktiviert wird.
Load	... das Formular geladen wird.
Paint	... das Formular neu gezeichnet werden muss.
Resize	... die Größe eines Formulars verändert wird.
HelpRequested	... Hilfe vom Nutzer angefordert wird (*F1*).
Layout	... die untergeordneten Controls neu positioniert werden müssen (Größenänderung des Formulars oder Hinzufügen von Steuerelementen).
LocationChanged	... sich die Position des Formulars ändert.
MdiChildActivated	... wenn ein MDI-Child-Fenster aktiviert wird.

Beispiel 2.3: Verwenden des *Paint*-Events zum Zeichnen eines Kreises

C#

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    e.Graphics.DrawEllipse(new Pen(Color.Black), 0, 0, 100, 100);
}
```

Ergebnis

Mit jedem Anzeigen oder jeder Größenänderung wird *Paint* ausgelöst. Das Ergebnis:

HINWEIS: Mehr zur Grafikprogrammierung finden Sie in Kapitel 4 des Onlineteils.

Beim Laden eines Formulars treten – nach Aufruf des Konstruktors – die Ereignisse in folgender Reihenfolge auf:

Move → *Load* → *Layout* → *Activated* → *Paint*

Beim Schließen hingegen haben wir es mit folgender Ereigniskette zu tun:

FormClosing → *FormClosed* → *Deactivate*

HINWEIS: Das Praxisbeispiel in Abschnitt 2.4.2 zeigt, wie Sie obige Ereignisketten selbst experimentell ermitteln können!

2.1.3 Wichtige Methoden des Form-Objekts

Die wichtigsten Methoden für Formulare sind im Folgenden zusammengestellt.

Methode	Beschreibung
Activate	… aktiviert das Formular
BringToFront	… verschiebt das Formular an die oberste Position (innerhalb der Anwendung)
Close	… schließt das Formular
CreateControl	… erzeugt ein neues Steuerelement
CreateGraphics	… erstellt ein *Graphics*-Objekt für die grafische Ausgabe
DoDragDrop	… startet eine Drag & Drop-Operation
Focus	… setzt den Fokus auf das Formular

GetNextControl	… liefert das folgende Control in der Tab-Reihenfolge
Hide	… verbirgt das Formular
Invalidate	… erzwingt ein Neuzeichnen des Formularinhalts
PointToClient *RectangleToClient*	… rechnet Screen-Koordinaten in Fensterkoordinaten um (je nach Fensterposition)
PointToScreen *RectangleToScreen*	… rechnet Fensterkoordinaten in Screen-Koordinaten um
Refresh	… erzwingt ein Neuzeichnen des Fensters und der untergeordneten Controls
SelectNextControl	… verschiebt den Eingabefokus bei den untergeordneten Controls
Show	… zeigt das Fenster an
ShowDialog	… zeigt das Fenster als Dialogbox (modal) an

Beispiel 2.4: Umrechnen in Screen-Koordinaten

C#

```
private void Form1_MouseDown(object sender, MouseEventArgs e)
{
    Text = $"Screen-Koordinaten: {PointToScreen(new Point(e.X, e.Y))}";
}
```

Ergebnis

Die Anzeige hängt jetzt nicht nur von der relativen Mausposition, sondern auch von der absoluten Position des Formulars ab:

2.2 Praktische Aufgabenstellungen

Im Folgenden wollen wir die oben genannten Eigenschaften, Methoden und Ereignisse nutzen, um einige recht praktische Aufgabenstellungen im Umgang mit Windows-Formularen zu lösen.

2.2.1 Fenster anzeigen

Wie Sie aus der Startprozedur *Main()* heraus ein Fenster aufrufen, wurde im vorhergehenden Kapitel beschrieben. Wie Sie aus dem Hauptfenster heraus weitere Formulare aufrufen, soll Mittelpunkt dieses Abschnitts sein.

Zwei grundsätzliche Typen von Formularen müssen Sie unterscheiden:

- modale Fenster (Dialoge),
- nichtmodale Fenster.

Die Unterscheidung zwischen beiden Varianten findet erst beim Aufruf bzw. bei der Anzeige eines *Form*-Objekts statt. Zur Entwurfszeit wird diese Unterscheidung nicht getroffen, sieht man einmal von unterschiedlichen Rahmentypen (*FormBorderStyle*-Eigenschaft) ab.

Nichtmodale Fenster

Hierbei handelt es sich um ein Fenster, das den Fokus auch an andere Fenster abgeben bzw. das auch verdeckt werden kann.

Die Anzeige erfolgt mithilfe der Methode *Show*, die asynchron ausgeführt wird, das heißt, es wird mit der Programmverarbeitung nicht auf das Schließen des Formulars gewartet.

HINWEIS: Vor der Anzeige des Formulars muss dieses mit *new* instanziiert werden!

Beispiel 2.5: Instanziieren und Anzeigen eines weiteren Formulars

C#

```
private void ButtonShowForm2_Click(object sender, EventArgs e)
{
    Form2 f2 = new Form2();
    f2.Text = "Mein zweites Formular";
    f2.Show();
}
```

Der „Nachteil“ dieser Fenster: Sie können zum einen verdeckt werden, zum anderen wissen Sie als Programmierer nie, wann der Anwender das Fenster schließt. Für die Eingabe von Werten und deren spätere Verarbeitung sind sie also ungeeignet.

Modale Fenster (Dialoge)

Abhilfe schaffen die Dialogfenster, auch modale Fenster genannt. Diese werden statt mit *Show* mit der Methode *ShowDialog* angezeigt. Die Programmausführung wird mit dem Aufruf der Methode an dieser Stelle so lange gestoppt, bis der Nutzer das Formular wieder geschlossen hat.

Beispiel 2.6: Anzeige eines Dialogfensters

C#

```
private void ButtonShowForm2Modal_Click(object sender, EventArgs e)
{
    Form2 f2 = new Form2();
    f2.Text = "Bitte tragen Sie Ihren Namen ein ...";
    f2.ShowDialog();
    MessageBox.Show(f2.textBox1.Text);  // Anzeige des Eingabewerts
}
```

HINWEIS: Bevor Sie auf Controls in *Form2* zugreifen können, müssen Sie deren *Modifiers*-Eigenschaft auf *public* oder internal festgelegt haben. Andernfalls können Sie nicht mit den Controls arbeiten.

Zu einer ordentlichen Dialogbox gehören im Allgemeinen auch ein OK- und ein Abbruch-Button. Auf diese Weise kann im aufrufenden Programm schnell entschieden werden, welcher Meinung der Anwender beim Schließen der Dialogbox war. Von zentraler Bedeutung ist in diesem Fall der Rückgabewert der Methode *ShowDialog*.

Beispiel 2.7: Auswerten des Rückgabewerts

C#

```
Form2 f2 = new Form2();
if (f2.ShowDialog() == DialogResult.OK)
{...}
```

Die möglichen Rückgabewerte:

DialogResult
Abort
Cancel
Ignore
No
None
OK
Retry
Yes

In der Dialogbox selbst stellen Sie den Rückgabewert entweder durch das direkte Setzen der Eigenschaft *DialogResult* per Code ein oder Sie weisen den beiden Buttons (OK, Abbruch) die gewünschte *DialogResult*-Eigenschaft zu.

Beispiel 2.8: Verwendung von *DialogResult*

C#

Auf *Form1* befinden sich ein *Label* und ein *Button*. Über Letzteren wird ein Dialogfenster *Form2* aufgerufen, das die Schaltflächen „OK“ und „Abbruch“ besitzt. Außerdem hat *Form2* eine *TextBox*, deren *Modifiers*-Eigenschaft Sie auf *public* setzen.

```
public partial class Form1 : Form
{
    private void ButtonShowForm2Modal_Click(object sender, EventArgs e)
    {
        Form2 f2 = new Form2();
        f2.Text = "Bitte tragen Sie Ihren Namen ein ... ";
        if (f2.ShowDialog() == DialogResult.OK)
        {
            label1.Text = f2.textBox1.Text;
        }
    }
}
```

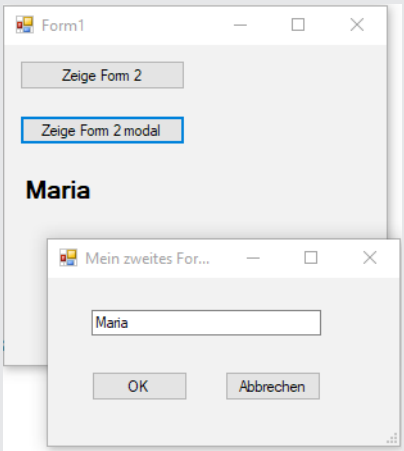
Der Code von *Form2*:

```
public partial class Form2 : Form
{
    public Form2()
    {
        InitializeComponent();

        btnOK.DialogResult = DialogResult.OK;
        btnCancel.DialogResult = DialogResult.Abort;
    }
}
```

Die folgenden beiden Zuweisungen können Sie auch direkt im Eigenschaftenfenster von *button1* bzw. *button2* vornehmen:

Ergebnis

Nach dem Klick auf „OK“ wird der in *Form1* eingetragene Wert in *Form2* angezeigt.



2.2.2 Splash Screens beim Anwendungsstart anzeigen

Von vielen kommerziellen Anwendungen ist Ihnen sicher die Funktion eines Splash Screens bekannt. Der Hintergrund ist in vielen Fällen, dass die Zeit für das Laden von Programmmodulen, Datenbanken etc. für den Endanwender irgendwie sinnvoll überbrückt werden soll, ohne dass der Verdacht aufkommt, die Anwendung „hängt". An einem recht einfachen Vertreter dieser Gattung wollen wir Ihnen die prinzipielle Vorgehensweise demonstrieren.

Beispiel 2.9: Einsatz eines Splash Screens

C#

Erstellen Sie zunächst ein neues Projekt und fügen Sie diesem neben dem Standardformular *Form1* ein weiteres Formular *SplashScreen* hinzu. Diesem gilt auch zunächst unsere Aufmerksamkeit.

Fügen Sie *SplashScreen* einen *Label* für die Begrüßungsmeldung und einen *Label* für mögliche Statusmeldungen hinzu. Setzen Sie weiterhin die *FormBorderStyle*-Eigenschaft auf den von Ihnen favorisierten Wert (z. B. *None*) sowie die Eigenschaft *StartPosition* auf *ScreenCenter*. Nicht vergessen dürfen wir auch die Eigenschaft *TopMost*, die wir tunlichst auf *true* setzen sollten, damit unser Formular auch im Vordergrund steht und nicht hinter dem Hauptformular der Anwendung verloren geht.

Die Klassendefinition erweitern Sie bitte um die im Folgenden fett hervorgehobenen Einträge:

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

Da wir nur ein Formular benötigen, definieren wir dieses gleich intern und statisch:

```
        static SplashScreen frmSplashScreen = null;
```

Mit der folgenden *Start*-Methode wird das Formular initialisiert und angezeigt. Da die Methode statisch ist, können wir diese direkt mit *SplashScreen.Start()* aufrufen.

```
        static public void Start()
        {
            if (frmSplashScreen != null)
            {
                return;
            }
            frmSplashScreen = new SplashScreen();
            frmSplashScreen.Show();
        }
```

Auch das Schließen des Formulars übernimmt eine statische Methode der *SplashScreen*-Klasse:

```
        static public void Stop()
        {
            frmSplashScreen.Close();
            frmSplashScreen = null;
        }
```

Möchten Sie Statusmeldungen anzeigen, ist dies über die statische Methode *SetMessage* möglich. Die Meldungen werden im *label2* angezeigt:

```
        static public void SetMessage(string msg)
        {
            frmSplashScreen.label2.Text = msg;
            Application.DoEvents();
        }
    }
...
```

Selbstverständlich können Sie auch noch Fortschrittsanzeigen etc. in diesem Formular unterbringen, das Grundprinzip dürfte jedoch weitgehend gleich bleiben.

Was fehlt, ist die Integration des neuen Splash Screens in Ihre Anwendung. Öffnen Sie dafür die Datei *Program.cs* und nehmen Sie folgende Erweiterung vor:

```
    static class Program
    {
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            SplashScreen.Start();
            Application.Run(new Form1());
        }
    }
```

Last but not least ist auch unser eigentliches Programm von Interesse. Wir wollen hektische und langwierige Datenbankoperationen beim Öffnen von *Form1* mit einer einfachen Schleife und der *Thread.Sleep*-Methode simulieren:

```
...
    public partial class Form1 : Form
    {
        private void Form1_Load(object sender, EventArgs e)
        {
            for (int i = 0; i < 100; i++)
            {
                Application.DoEvents();
                System.Threading.Thread.Sleep(50);
```

Hier besteht die Möglichkeit, den aktuellen Fortschritt im Splash Screen anzuzeigen:

```
                SplashScreen.SetMessage("Schritt " + i.ToString());
            }
```

Sind alle Ladeaktivitäten absolviert, sollten wir auch unseren Splash Screen wieder ausblenden:

```
            SplashScreen.Stop();
        }
```

Beim Start der Anwendung sollte jetzt zunächst unser Splash Screen erscheinen und anschließend *Form1*.

HINWEIS: Man kann sicher auch die Anzeige des Splash Screens in einen extra Thread auslagern, um die Aktualisierungen dieser Forms unabhängig von den anderen Aktivitäten der Anwendung zu realisieren. Die Alternative ist das regelmäßige Aufrufen der *DoEvents*-Methode, wie in obigem Beispiel demonstriert.

2.2.3 Eine Sicherheitsabfrage vor dem Schließen anzeigen

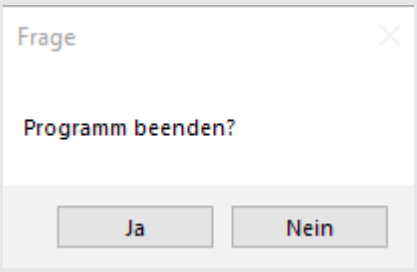
Für das Schließen der Anwendung bieten sich viele Möglichkeiten und so ist es sicher ratsam, die entsprechende Routine zentral zu organisieren. Dazu bietet sich das *FormClosing*-Ereignis an, wie es das folgende Beispiel zeigt.

Beispiel 2.10: Sicherheitsabfrage vor dem Schließen des Formulars

C#

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    e.Cancel = MessageBox.Show("Programm beenden?", "Frage",
          MessageBoxButtons.YesNo) == DialogResult.No;
}
```

Ergebnis



Die Neugier des Programmierers ist in vielen Fällen angebracht und so bietet der Parameter *e* ganz nebenbei auch Informationen darüber, warum das Ereignis ausgelöst wurde. Verantwortlich dafür ist der Member *CloseReason*, dessen einzelne Konstanten Sie der folgenden Tabelle entnehmen können:

Member	Beschreibung
`ApplicationExitCall`	Die Methode *Application.Exit* wurde im Programm aufgerufen.
`FormOwnerClosing`	Das Hauptformular wurde geschlossen.
`MdiFormClosing`	Das zentrale MDI-Form wurde geschlossen.
`None`	Hier weiß auch die API nicht weiter.
`TaskManagerClosing`	Der Taskmanager will die Anwendung schließen.
`UserClosing`	Eine Nutzeraktion (Formularschaltflächen) führt zum Schließen.
`WindowsShutDown`	Das Betriebssystem wird heruntergefahren.

2.2.4 Ein Formular durchsichtig machen

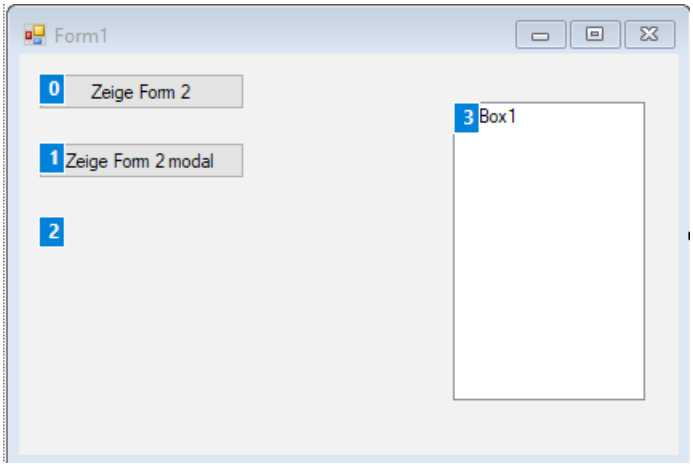
Sind Ihnen die bisherigen Formulare zu schlicht, können Sie Ihre Anwendung auch mit einem Transparenzeffekt aufpeppen. Setzen Sie dazu einfach die *Opacity*-Eigenschaft auf einen Wert zwischen 0% (vollständige Transparenz) und 100%. Das Ergebnis bei 50% zeigt folgender Bildschirmausschnitt mit Blick auf den Desktop:

2.2.5 Die Tabulatorreihenfolge festlegen

Gerade bei Dialogboxen ist die Eingabereihenfolge von übergeordnetem Interesse. Was nützt dem Anwender eine Dialogbox, in der der Eingabefokus willkürlich zwischen den Text- und ComboBoxen hin und her springt?

Da der Wechsel von einem Eingabe-Control zum nächsten mit der Tabulatortaste erfolgt, spricht man auch von Tabulatorreihenfolge. Jedes sichtbare Steuerelement verfügt zu diesem Zweck über die Eigenschaften *TabIndex* und *TabStop*. Während mit *TabStop* lediglich festgelegt wird, ob das Control überhaupt den Fokus erhalten kann (mittels Tab-Taste), können Sie mit *TabIndex* Einfluss auf die Reihenfolge nehmen.

Visual Studio unterstützt Sie bei dieser Arbeit recht gut. Um den Überblick zu verbessern, können Sie die Tabulatorreihenfolge im Entwurfsmodus sichtbar machen. Aktivieren Sie diese über den Kontextmenüpunkt **Ansicht | Aktivierreihenfolge**. Ihr Formular dürfte danach zum Beispiel folgenden Anblick bieten:



Klicken Sie jetzt einfach in der gewünschten Reihenfolge in die kleinen Fähnchen, um die Tabulatorreihenfolge zu ändern.

2.2.6 Ausrichten von Komponenten im Formular

War es in der ersten Version (zu Beginn des Jahrtausends) von Visual Studio noch eine Qual bzw. ein riesiger Aufwand, Komponenten in einem Formular sauber auszurichten, stellt dies heutzutage kaum noch ein Problem dar. Zwei wesentliche Verfahren bieten sich an:

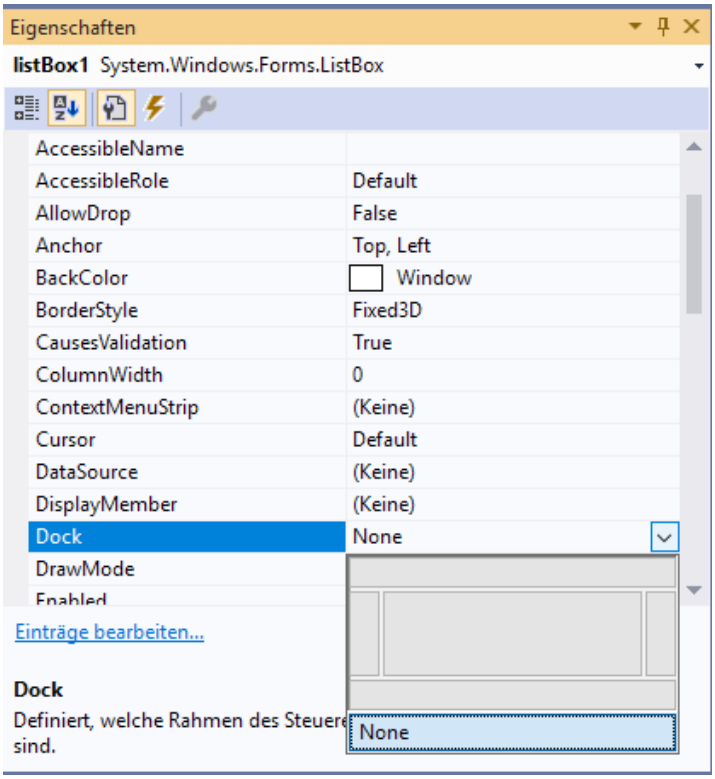
- **Docking** (Andocken an die Außenkanten einer übergeordneten Komponente)
- **Anchoring** (Ausrichten relativ zu den Außenkanten einer übergeordneten Komponente)

Verantwortlich für das Ausrichten sind die beiden naheliegenden Eigenschaften *Dock* und *Anchor*.

HINWEIS: Natürlich wollen wir der Vollständigkeit halber nicht vergessen, hier auch auf die Eigenschaften *Location* (linke obere Ecke) und *Size* (Breite, Höhe) hinzuweisen.

Dock

Öffnen Sie das Eigenschaftenfenster (F4) und wählen Sie die Eigenschaft *Dock*, steht Ihnen der Eigenschafteneditor zur Verfügung:



Eine Komponente lässt sich mit dieser Eigenschaft fest an den vier Außenkanten oder in der verbleibenden Clientfläche ausrichten. Dabei verändert sich die Größe der Komponente nur so, dass die Ausrichtung an den Außenkanten erhalten bleibt.

HINWEIS: Möchten Sie die Ausrichtung aufheben, wählen Sie im Eigenschaften-Editor die unterste Schaltfläche (*None*).

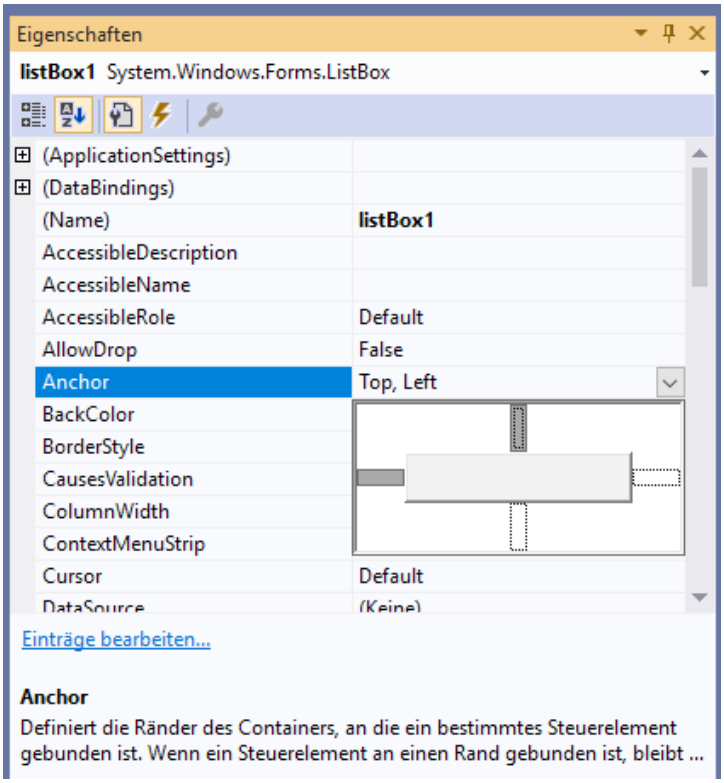
Damit stellt jetzt auch das Positionieren von Bildlaufleisten kein Problem mehr dar, einfach die Komponenten am rechten bzw. am unteren Rand ausrichten.

HINWEIS: Mit der Formulareigenschaft *Padding* können Sie einen Mindestabstand beim Docking vorgeben. Auf diese Weise lassen sich Ränder zu den Formularkanten definieren.

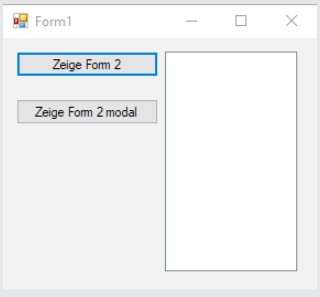
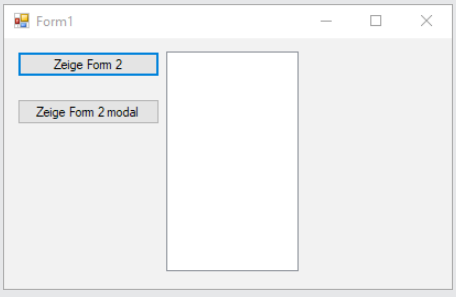
Anchor

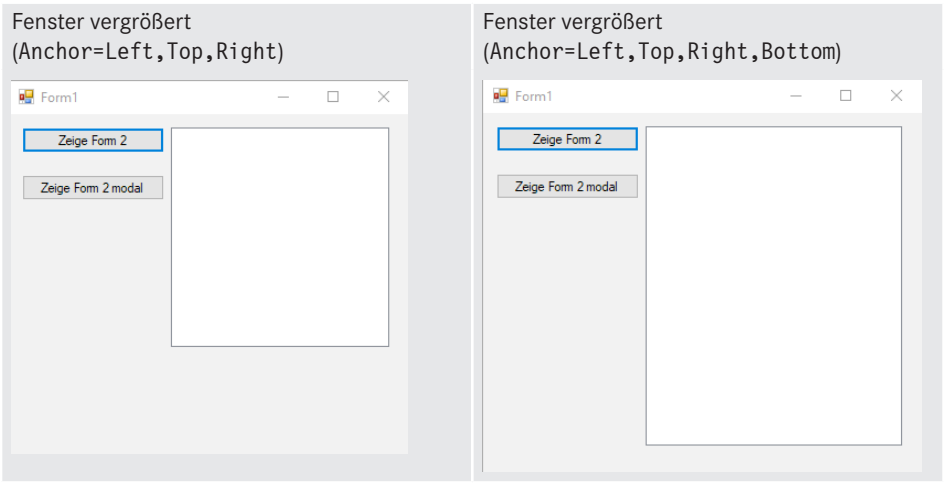
Etwas anders als die *Dock*-Eigenschaft verhält sich die *Anchor*-Eigenschaft. Standardmäßig ist *Anchor* immer mit *Left*, *Top* aktiv, bei Größenänderungen des Formulars bleibt also der

Abstand der Komponente zum linken und oberen Rand der umgebenden Komponente immer gleich. Auch hier steht ein eigener Eigenschafteneditor zur Verfügung:



Die folgenden Abbildungen zeigen Ihnen die Auswirkungen verschiedener *Anchor*-Einstellungen auf ein Formular:

Original-Fenstergröße	Fenster vergrößert (`Anchor=Left,Top`)
	



Zur Laufzeit können Sie die *Anchor*-Eigenschaft mit den Werten der Enumeration *AnchorStyles* festlegen (siehe folgende Tabelle).

Konstante	Das Steuerelement ist …
None	… nicht verankert.
Top	… am oberen Rand verankert.
Bottom	… am unteren Rand verankert.
Left	… am linken Rand verankert.
Right	… am rechten Rand verankert.

Da die *AnchorStyles*-Enumeration über ein *[Flags]*-Attribut verfügt, können die einzelnen Konstanten bitweise verknüpft werden.

Beispiel 2.11: Herstellen der Standardverankerung eines Buttons

C#

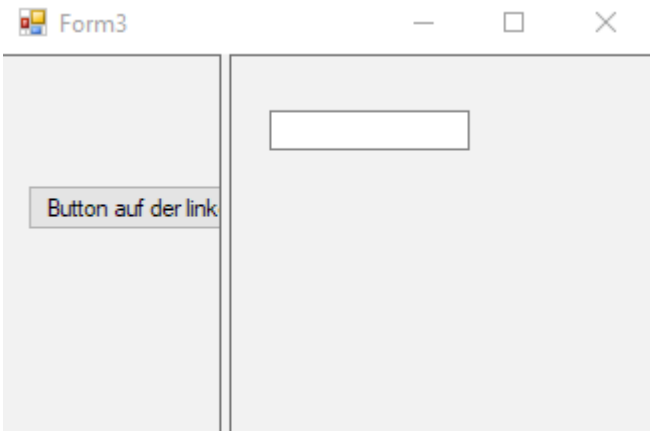
```
button1.Anchor = AnchorStyles.Left | AnchorStyles.Top;
```

2.2.7 Spezielle Panels für flexible Layouts

Auch hier geht es um die Bereitstellung eines flexiblen Formularlayouts. Allerdings haben wir es, im Unterschied zum vorhergehenden Abschnitt, mit keinen Formulareigenschaften mehr zu tun, sondern mit speziellen Panels, die Sie im „Container“-Segment der Toolbox vorfinden.

SplitContainer

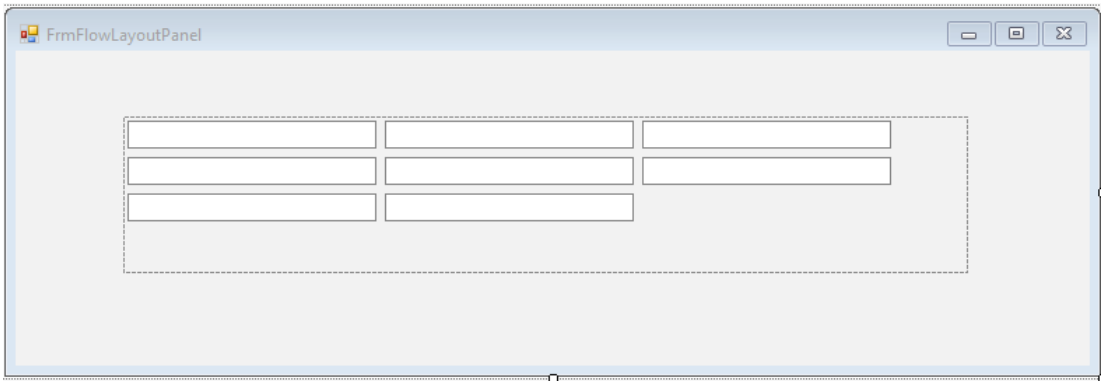
Diese Komponente gilt als Nachfolger für den *Splitter* und besteht aus zwei Panels, die durch einen zur Laufzeit veränderlichen Balken getrennt sind.



Wichtige Eigenschaften sind *Orientation* (horizontaler oder vertikaler Trennbalken), *IsSplitterFixed* (fester oder beweglicher Balken) und *BorderStyle* (in der Standardeinstellung *None* bleibt der Balken unsichtbar).

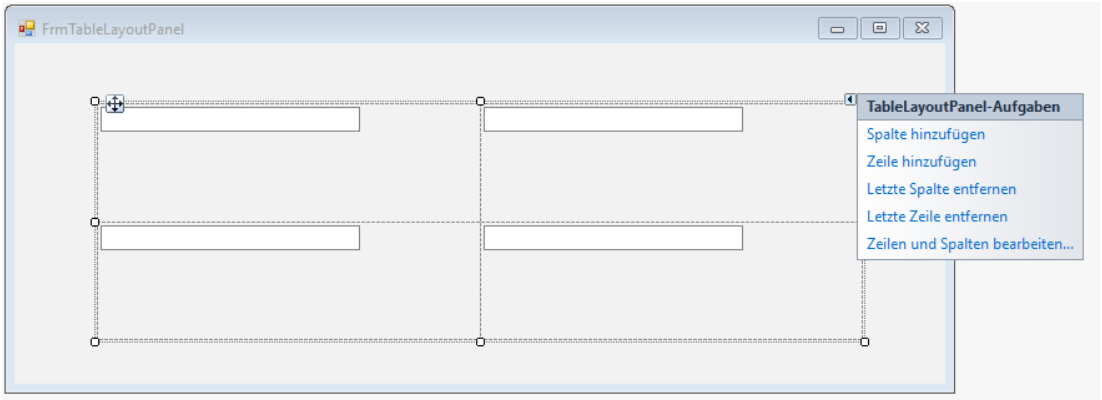
FlowLayoutPanel

Diese Komponente layoutet die auf ihr platzierten Steuerelemente dynamisch in horizontaler oder vertikaler Richtung. Das heißt, bei einer Größenänderung des Formulars werden die Steuerelemente automatisch „umgebrochen“ (`WrapContents = True`). Wichtig ist die *Anchor*-Eigenschaft, die die Ränder des Containers (`Top`, `Bottom`, `Left`, `Right`) bestimmt, an welche die Steuerelemente gebunden werden sollen.



TableLayoutPanel

Diese Komponente besorgt ein dynamisches Layout, das sich an einer Gitterstruktur orientiert. Wichtig sind die *Rows*- und die *Columns*-Eigenschaft, die über spezielle Dialoge (aufrufbar über den Smarttag rechts oben) den individuellen Bedürfnissen angepasst werden können.



2.2.8 Menüs erzeugen

An dieser Stelle wollen wir etwas vorgreifen, da *MenuStrip*- und *ContextMenuStrip*-Komponente eigentlich erst in das folgende Kapitel gehören, wo es um die Beschreibung der wichtigsten Steuerelemente geht. Doch die optische Verzahnung zwischen Menü und Formular ist so eng, dass wir bereits an dieser Stelle auf dieses Thema eingehen wollen.

Jedem Menüelement kann über seine *Image*-Eigenschaft auf einfache Weise eine Grafik zugewiesen werden. Menüeinträge können auch als *ComboBox*, *TextBox* oder *Separator* in Erscheinung treten.

MenuStrip

Möchten Sie ein „normales“ Menü erzeugen, platzieren Sie einfach eine *MenuStrip*-Komponente auf das Formular. Es handelt sich zunächst um eine nicht sichtbare Komponente, die lediglich im Komponentenfach zu sehen ist. Doch halt, auch in der Kopfzeile des Formulars tut sich etwas, sobald die Komponente markiert wird: Das Menü wird genau dort bearbeitet, wo es sich zur Laufzeit auch befindet. Klicken Sie also in den Text „Hier eingeben“ und tragen Sie beispielsweise „Datei“ ein. Automatisch werden bereits zwei weitere potenzielle Menüpunkte erzeugt (ein Menüpunkt auf der gleichen Ebene, ein untergeordneter Menüpunkt), die Sie auf die gleiche Weise bearbeiten können.

HINWEIS: Möchten Sie einen Trennstrich einfügen, genügt die Eingabe eines einzelnen Minuszeichens (-) als Beschriftung!

Beispiel 2.12: MenuStrip

C#

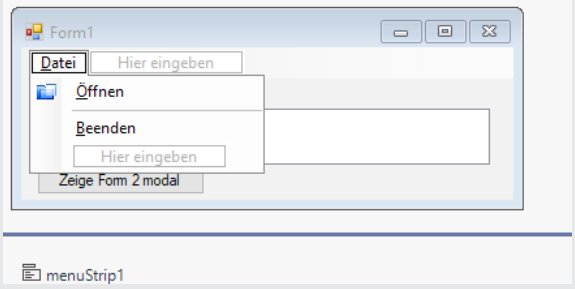
Die Abbildung am Ende des Beispiels zeigt ein Datei-Menü mit zwei Einträgen (*MenuItems*). Dazwischen befindet sich ein Trennstrich. Die kleine Grafik bei *Datei/Öffnen* wurde diesem Menüpunkt über dessen *Image*-Eigenschaft zugewiesen.

Um für einen Menüpunkt die Ereignisprozedur zu erzeugen, genügt ein Doppelklick und schon befinden Sie sich wieder im Code-Editor und programmieren die Funktionalität.

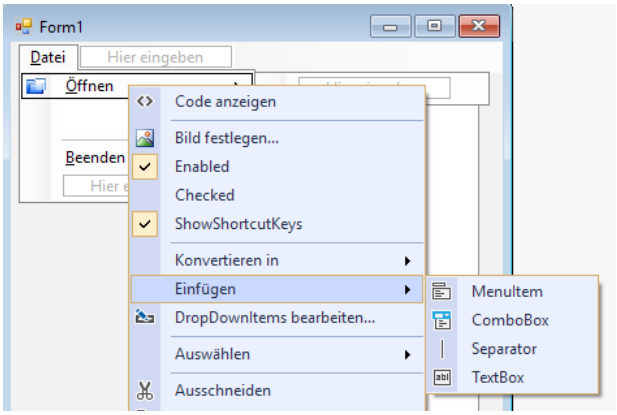
Die beiden Menüeinträge werden mit Code hinterlegt.

```
public partial class Form1 : Form
{
    ...
    private void ÖffnenToolStripMenuItem_Click(object sender, EventArgs e)
    {
        MessageBox.Show("Das ist nur ein Test!");
    }

    private void BeendenToolStripMenuItem_Click(object sender, EventArgs e)
    {
        Close();
    }
}
```

Ergebnis



Anstatt eines normalen Menüeintrags können auch eine *TextBox* oder eine *ComboBox* verwendet werden.



Beispiel 2.13: *ComboBox* in *MenuStrip*

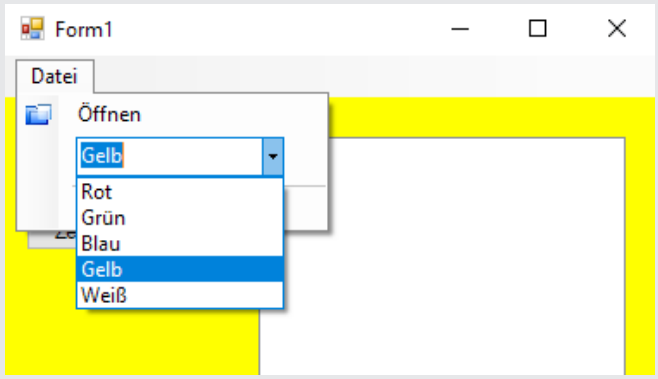
C#

Die oberste Menüebene des Vorgängerbeispiels wird um eine *ComboBox* erweitert, mit der man die Hintergrundfarbe des Formulars einstellen kann.

Die *Items*-Eigenschaft der *ComboBox* wird mit dem über das Eigenschaftenfenster erreichbaren Editor zeilenweise mit den Einträgen für die einzelnen Farben gefüllt. Zur Programmierung kann man das *SelectedIndexChanged*-Ereignis der *ComboBox* ausnutzen:

```
private void toolStripComboBox1_SelectedIndexChanged(object sender, EventArgs e)
{
    Color c = Color.White;
    switch (toolStripComboBox1.SelectedIndex)
    {
        case 0:
            c = Color.Red;
            break;
        case 1:
            c = Color.Green;
            break;
        case 2:
            c = Color.Blue;
            break;
        case 3:
            c = Color.Yellow;
            break;
        case 4:
            c = Color.White;
            break;
    }
    BackColor = c;
}
```

Ergebnis



ContextMenuStrip

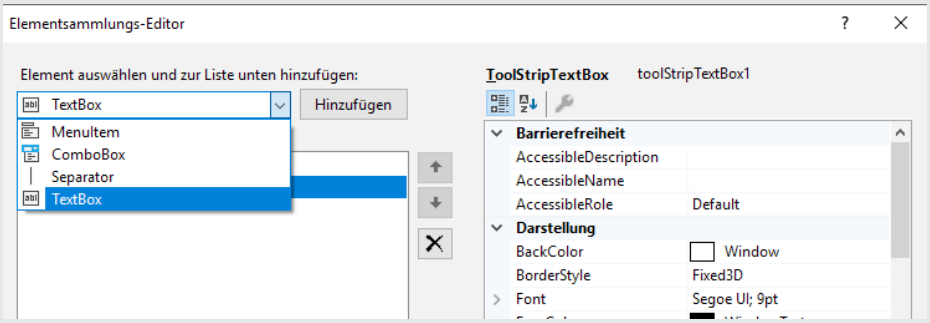
Die Programmierung eines Kontextmenüs unterscheidet sich nur unwesentlich von der eines normalen Menüs. Der Entwurfsprozess ist nahezu identisch.

HINWEIS: Da sich ein Kontextmenü in der Regel auf ein bestimmtes visuelles Steuerelement bezieht, verfügen Letztere über eine *ContextMenuStrip*-Eigenschaft.

Beispiel 2.14: *ContextMenuStrip*

C#

Eine *ContextMenuStrip*-Komponente wird mit einer *TextBox* ausgestattet, um zur Laufzeit eine *ListBox* mit Einträgen zu füllen.

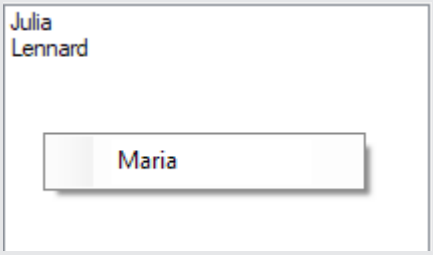


Die *ContextMenuStrip*-Eigenschaft der *ListBox* ist mit dem Kontextmenü zu verbinden!

Nach Klick mit der rechten Maustaste auf die *ListBox* erscheint das Kontextmenü. Jeder neue Eintrag ist mittels *Enter*-Taste abzuschließen:

```
private void ToolStripTextBox1_KeyUp(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.Enter)
    {
        listBox1.Items.Add(toolStripTextBox1.Text);
        toolStripTextBox1.Text = String.Empty;
    }
}
```

Ergebnis



Weitere Eigenschaften von Menüeinträgen

Jeder Menüeintrag wird durch ein *ToolStripMenuItem*-Objekt dargestellt, hat also eigene Eigenschaften, von denen die *Text*-Eigenschaft die offensichtlichste ist. Auch viele der übrigen Eigenschaften erklären sich von selbst.

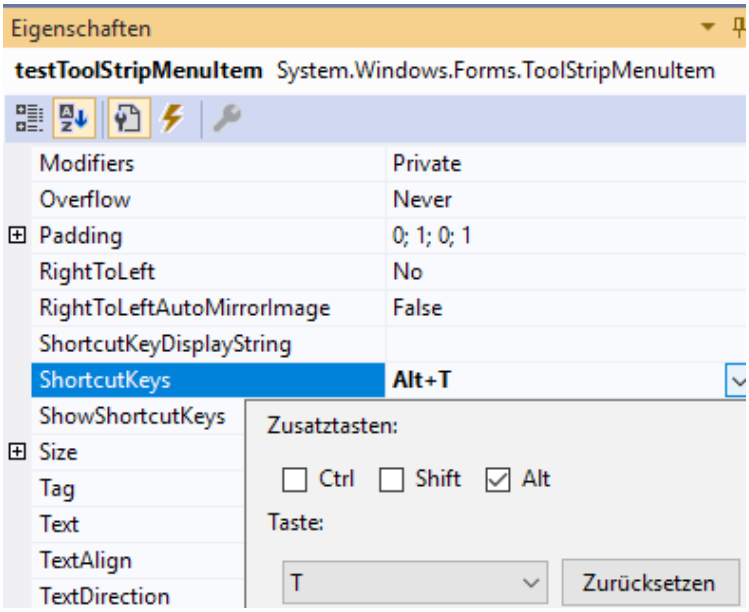
Beispiel 2.15: Mit der Eigenschaft *Checked* bestimmen Sie, ob vor dem Menüpunkt ein Häkchen angezeigt werden soll oder nicht.

C#

In der Ereignisprozedur können Sie den Wert entsprechend setzen oder auslesen:

```
private void TestToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (testToolStripMenuItem.Checked)
    {
        MessageBox.Show("Das ist nur ein Test!");
    }
    testToolStripMenuItem.Checked = !testToolStripMenuItem.Checked;
}
```

Über die Eigenschaft *Visible* steuern Sie die Sichtbarkeit des Menüeintrags. Besser als das Ausblenden ist meist das Deaktivieren mit der *Enabled*-Eigenschaft. Über *ShortcutKeys* bzw. *ShowShortcutKeys* können Sie den Menüpunkt mit einer Tastenkombination verbinden (z. B. *Alt+G* oder *F7*).



2.3 MDI-Anwendungen

Ein Windows-Programm läuft oft als MDI-Applikation ab. Das heißt, innerhalb eines Rahmen- bzw. Mutterfensters können sich mehrere sogenannte Kindfenster tummeln[1]. Diese werden vom Mutterfenster verwaltet und so „an der Leine gehalten", dass sie z. B. auch dessen Bereich nicht verlassen können.

2.3.1 „Falsche" MDI-Fenster bzw. Verwenden von Parent

Formulare verfügen, wie auch die einfachen Controls, über die Eigenschaft *Parent*. Mithilfe dieser Eigenschaft können Sie ein Formular wie ein Control behandeln und in ein übergeordnetes Formular einfügen.

Das Resultat dieses Vorgehens: Das Formular kann den Clientbereich seines Parent nicht mehr verlassen, Sie können es aber wie gewohnt verschieben, skalieren oder auch schließen. Von besonderem Interesse dürfte die Möglichkeit sein, die untergeordneten Formulare wie Controls anzudocken. Damit steht der Programmierung von Toolbars etc. kaum noch etwas im Wege.

Beispiel 2.16: Einfügen des Formulars *Form2* in den Clientbereich von *Form1*

C#

```
public partial class Form1 : Form
{ ...
    private void Button1_Click(object sender, EventArgs e)
    {
        Form2 f2 = new Form2();
        f2.TopLevel = false;
        f2.Parent = this;
        f2.Show();
    }
}
```

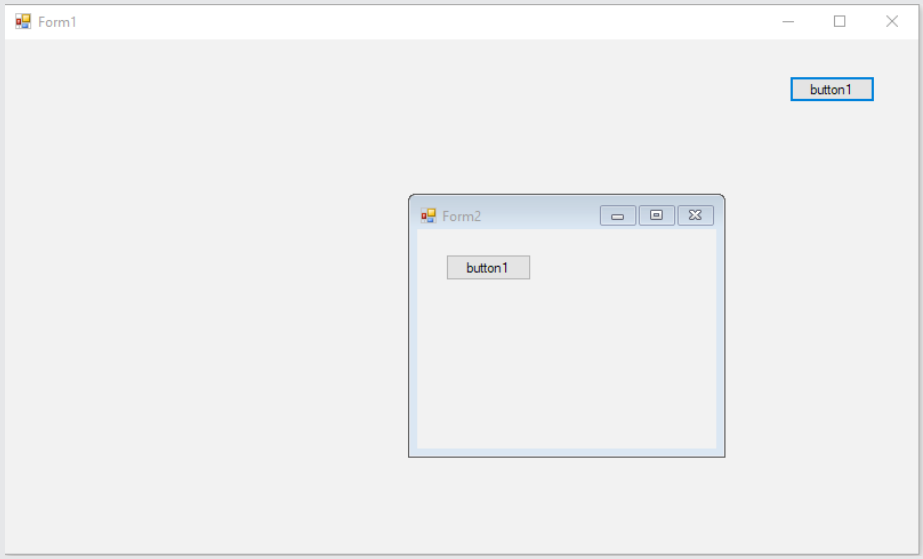
In *Form2* realisieren wir das Docking:

```
public partial class Form2 : Form
{ ...
    private void Button1_Click(object sender, EventArgs e)
    {
       Dock = DockStyle.Left;
    }
}
```

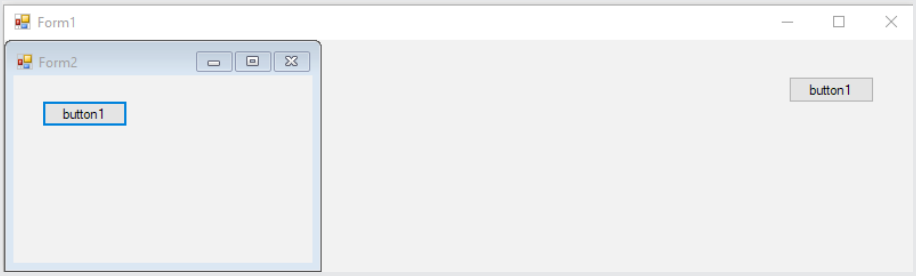
[1] Textverarbeitungsprogramme wie Word waren lange Zeit meist als Multiple Document Interface-Applikation aufgebaut. Die einzelnen Dokumente sind die Kindfenster.

Ergebnis

Das Endergebnis unserer Bemühungen:



bzw. angedockt:



2.3.2 Die echten MDI-Fenster

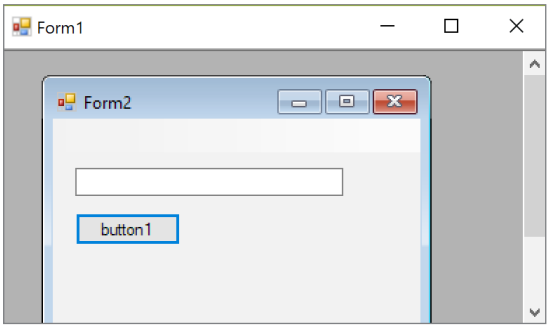
Möchten Sie „echte“ MDI-Anwendungen programmieren, brauchen Sie wie im vorhergehenden Beispiel ebenfalls mindestens zwei Formulare, von denen jedoch eines als MDIContainer definiert ist (Eigenschaft *IsMdiContainer=True*). Nachfolgend ändert sich schon zur Entwurfszeit das Aussehen des Formulars (der Hintergurnd wird dunkler).

HINWEIS: Sie sollten keine weiteren Controls im MDIContainer platzieren, es sei denn, Sie richten diese mithilfe der *Dock*-Eigenschaft an den Außenkanten des Formulars aus (z. B. ein *Panel*).

2.3.3 Die Kindfenster

Die Kind- oder auch Child-Fenster werden über die Eigenschaft *MdiParent* kenntlich gemacht. Weisen Sie dieser Eigenschaft ein MDIContainer-Fenster zu, werden die Kindfenster automatisch in den Clientbereich des MDIContainers verschoben.

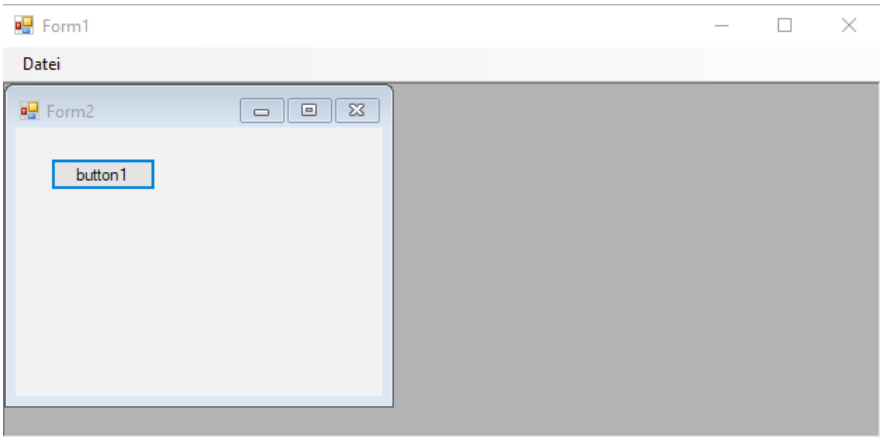
HINWEIS: Die *MDIParent*-Eigenschaft lässt sich nur zur Laufzeit zuweisen (also nicht über das Eigenschaftenfenster)!



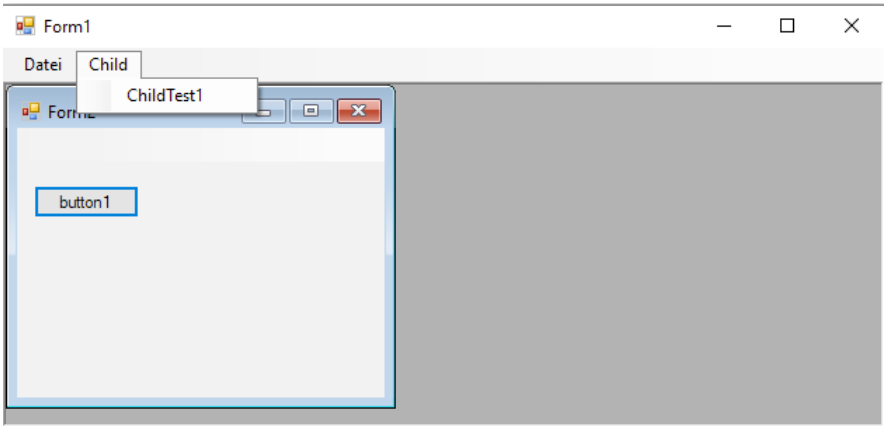
Die Umsetzung:

```
public partial class Form1 : Form
{ ...
    private void NeuesChildToolStripMenuItem_Click(object sender, EventArgs e)
    {
        Form2 f2 = new Form2();
        f2.MdiParent = this;
        f2.Show();
    }
}
```

Vergrößern Sie ein MDI-Kindfenster auf Vollbild, so erscheint dessen Titel eingefasst in eckigen Klammern neben dem Titel des Hauptfensters:



Verfügen beide Fenster über eigene Menüs, so werden diese standardmäßig im Menü des MDIContainers kombiniert:

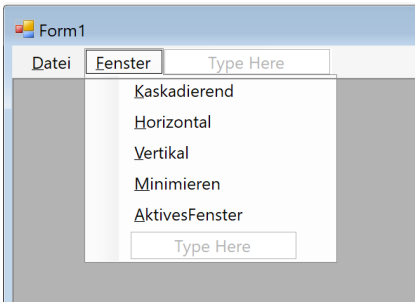


Gegenüber ihren konventionellen Kollegen besitzen die MDI-Kindfenster einige Einschränkungen, die durch das MDI-Konzept bedingt sind:

- MDI-Kindfenster werden nicht in der Windows-Taskleiste angezeigt.
- MDI-Kindfenster können den Clientbereich des MDIContainers nicht verlassen. Verschieben Sie die Fenster aus dem sichtbaren Bereich des MDIContainers, werden automatisch die nötigen Bildlaufleisten im Container angezeigt.
- Maximieren Sie ein MDI-Kindfenster, erreicht dieses maximal die Größe des verbleibenden Clientbereichs des MDIContainers (abzüglich Statuszeile, Menüleiste und Toolbar). Die Schaltflächen für Maximieren, Minimieren und Schließen des Kindfensters werden in diesem Fall in die Menüleiste des MDIContainers eingefügt.

2.3.4 Automatisches Anordnen der Kindfenster

Ein MDIContainer verfügt über die Methode *LayoutMdi*, mit der die vorhandenen Kindfenster nebeneinander, überlappend oder als Symbole angeordnet werden können. Üblicherweise entspricht dies dem in vielen MDI-Anwendungen vorhandenen *Fenster*-Menü:

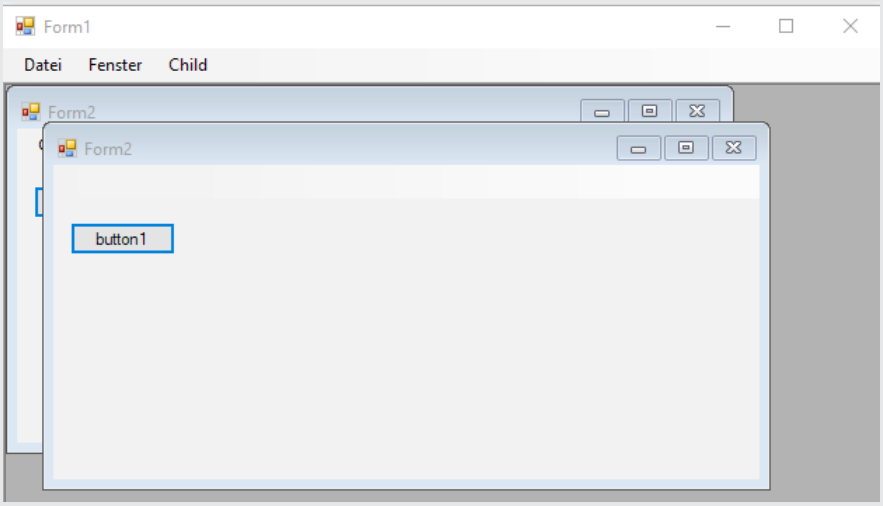


Beispiel 2.17: Hintereinander anordnen

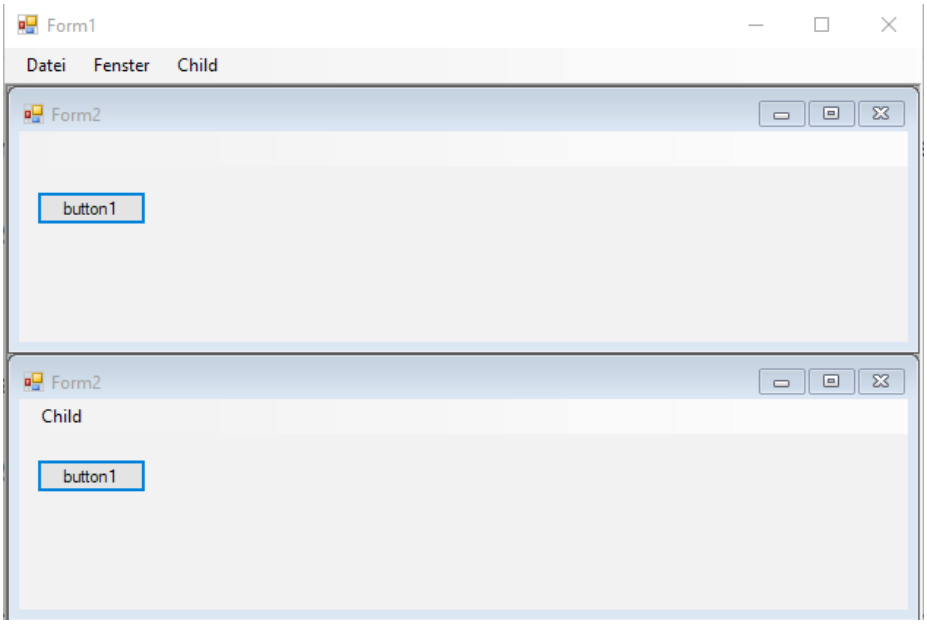
C#

```
private void KaskadierendToolStripMenuItem_Click(object sender,
                                                 EventArgs e)
{
    this.LayoutMdi(MdiLayout.Cascade);
}
```

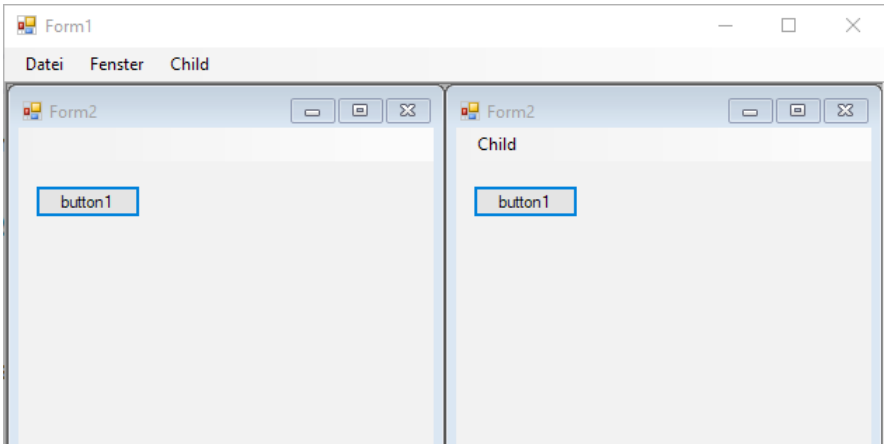
Ergebnis



Horizontal anordnen:



Vertikal anordnen:



2.3.5 Zugriff auf die geöffneten MDI-Kindfenster

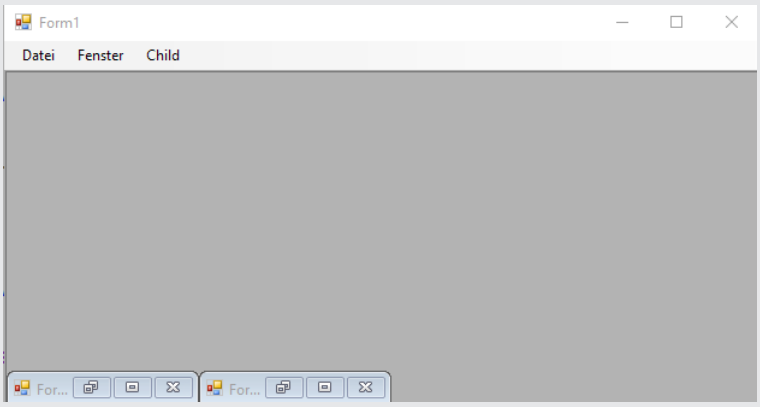
Über die Collection *MdiChildren* bietet sich die Möglichkeit, alle vorhandenen Kindfenster des MDIContainers zu verwalten.

Beispiel 2.18: Minimieren aller MDI-Kindfenster

C#

```
private void MinimierenToolStripMenuItem_Click(object sender, EventArgs e)
{
    foreach (Form f in this.MdiChildren)
    {
        f.WindowState = FormWindowState.Minimized;
    }
}
```

Ergebnis



2.3.6 Zugriff auf das aktive MDI-Kindfenster

Neben der Liste aller Kindfenster ist meist auch das gerade aktuelle von besonderem Interesse für den Programmierer, muss er doch häufig auf Inhalte des aktuellen Fensters (z. B. Textfelder, Markierungen) zugreifen. Die per MDIContainer verfügbare Eigenschaft *ActiveMdiChild* lässt die aufkommende Freude recht schnell vergessen, liefert diese doch zunächst nur ein Objekt der Klasse *Form* zurück. Es ist also Ihre Aufgabe, das zurückgegebene Objekt entsprechend zu typisieren.

Beispiel 2.19: Abfrage und Typisieren des aktuellen MDI-Kindfensters per MDIContainer

C#

```
private void AktivesFensterToolStripMenuItem_Click(object sender,
                                                   EventArgs e)
{
    if(ActiveMdiChild != null)
    {
        Form2 activeChild = (Form2)ActiveMdiChild;
        activeChild.label1.Text = "Ich bin aktiv";
    }
}
```

HINWEIS: Setzen Sie dazu den *Modifier* von *label1* auf *internal*. Ansonsten klappt der Zugriff nicht.

2.3.7 Mischen von Kindfenstermenü/MDIContainer-Menü

Das Vermischen der Menüs von MDIContainer und MDI-Kindfenstern kann über spezielle Eigenschaften (*AllowMerge*, *MergeAction*, *MergeIndex*) gesteuert werden und ist (fast) eine „Wissenschaft“ für sich.

AllowMerge

Die Einstellungen der Eigenschaften *MergeAction* und *MergeIndex* wirken sich nur dann aus, wenn für den *MenuStrip* der Wert von *AllowMerge* auf *true* gesetzt ist, was standardmäßig der Fall ist. Setzen Sie den Wert auf *false*, erfolgt keine Mischung der Menüs.

MergeAction und MergeIndex

Mit der Eigenschaft *MergeAction* geben Sie vor, wie die Menüleisten von Container und Kind kombiniert werden, *MergeIndex* spezifiziert in einigen Fällen die Position des einzufügenden Menüelements.

2.4 Praxisbeispiele

2.4.1 Informationsaustausch zwischen Formularen

In diesem Beispiel wird das auch von fortgeschrittenen Programmierern häufig nachgefragte Thema „Wie greife ich von FormA auf FormB zu“ abgehandelt.

Überblick

Unter Visual Studio wird der Code zum Erzeugen des Haupt- bzw. Startformulars (standardmäßig *Form1*) von der IDE automatisch generiert. Um das Erzeugen weiterer Formulare nach dem Muster

```
Form2 f2 = new Form2();
```

... muss sich der Programmierer selbst kümmern. Existiert die Formularinstanz, so können Sie gemäß den Regeln der OOP auf deren öffentliche Mitglieder auch von außerhalb zugreifen. Wenn beispielsweise im Code von *Form1* ein weiteres Formular (*Form2*) instanziiert wird, so ist es kein Problem, die öffentlichen Member von *Form2* zu verwenden. Etwas komplizierter wird es in umgekehrter Richtung, also wenn Sie von *Form2* (oder weiteren Kindformularen *Form3*, *Form4*, …) auf das Hauptformular (oder andere Kindformulare) zugreifen wollen. Wir wollen folgende Varianten betrachten, die eine Beziehung zwischen *Form2* (Childform) und *Form1* (Parentform) ermöglichen:

- Erzeugen einer Referenz auf *Form1* im Konstruktor von *Form2*,
- Setzen einer Eigenschaft in *Form2*, die eine Referenz auf *Form1* einrichtet,
- Verwenden des Owner-Mechanismus der Formulare.

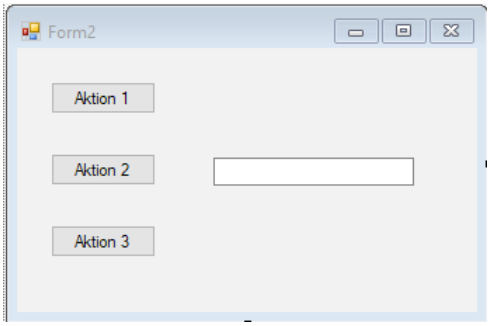
In unserem Beispiel, für das wir vier Formulare benötigen, werden wir diese drei Varianten vergleichen. Das erste Formular (*Form1*) wird als Hauptformular dienen und die anderen drei (*Form2*, *Form3*, *Form4*) sollen die untergeordneten Formulare sein. Letztere sind – der besseren Vergleichsmöglichkeiten wegen – mit der gleichen Bedienoberfläche ausgestattet und arbeiten auf identische Weise mit dem Hauptformular zusammen.

Bedienoberfläche Form1 (Hauptformular)

Öffnen Sie ein neues C#-Projekt als Windows Forms-Anwendung und gestalten Sie die abgebildete Oberfläche. Die in der *GroupBox* angeordneten Schaltflächen (*button2*, …, *button4*) rufen je ein untergeordnetes Formular (*Form2*, …, *Form4*) auf, das die gewünschte Variante demonstriert, wobei auf die beiden oben angeordneten Controls (*label1*, *button1*) des Hauptformulars zugegriffen wird.

Bedienoberfläche Form2 ... Form4 (untergeordnete Formulare)

Über das Menü **Projekt | Windows-Form hinzufügen...** ergänzen Sie das Projekt um drei weitere Formulare mit identischer Oberfläche (drei *Button*s und eine *TextBox*):



Die drei *Button*s sollen diverse Aktionen auf dem Hauptformular *Form1* auslösen:

- *button1* führt eine Instanzenmethode aus (Anzeige von Datum/Uhrzeit),
- *button2* soll *label1* mit dem Inhalt von *textBox1* füllen und
- *button3* löst das *Click*-Event für *button1* aus, wodurch der *BorderStyle* von *Form1* geändert wird.

Allgemeiner Code für Form1

Zunächst treffen wir im Code von *Form1* einige Vorbereitungen für den späteren Zugriff:

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

Den Lesezugriff auf die von außerhalb zu manipulierenden Controls als Eigenschaften offenlegen:

```
        public Label Label1
        {
            get
            { return label1;  }
        }

        public Button Button1
        {
            get
            { return button1; }
        }
```

Irgendeine Methode:

```
        public void MachWas()
        {
            Text = DateTime.Now.ToString();
        }
```

Click-Aktion auf *button1*:

```
        private void Button1_Click(object sender, EventArgs e)
        {
            if (FormBorderStyle.Equals(FormBorderStyle.None))
            {
                FormBorderStyle = FormBorderStyle.Sizable;
            }
            else
            {
                FormBorderStyle = FormBorderStyle.None;
            }
        }
        ...
}
```

Den Ereigniscode für *button2* … *button4*, in dem die untergeordneten Formulare aufgerufen werden, ergänzen wir später.

Variante 1: Übergabe der Formular-Referenz im Konstruktor

Bei dieser Variante wird der Konstruktor von *Form2* so modifiziert, dass er eine Referenz auf *Form1* entgegennehmen und damit die private Zustandsvariable *m_Form* setzen kann.

```
public partial class Form2 : Form
{
        private Form1 mainForm;

        public Form2(Form1 frm)
        {
            InitializeComponent();
            mainForm = frm;   // Referenz auf Hauptformular setzen
        }
```

Eine Instanzenmethode in *Form1* ausführen:

```
        private void Button1_Click(object sender, EventArgs e)
        {
            if (mainForm != null)
            {
                mainForm.MachWas();
            }
            else
            {
                MessageBox.Show("Form1 Instanz nicht gesetzt!");
            }
        }
```

Ein Steuerelement in *Form1* setzen:

```
        private void Button2_Click(object sender, EventArgs e)
        {
            if (mainForm != null)
            {
                mainForm.Label1.Text = textBox1.Text;
            }
            else
            {
                MessageBox.Show("Form1 Instanz nicht gesetzt!");
            }
        }
```

Ein Ereignis in *Form1* auslösen:

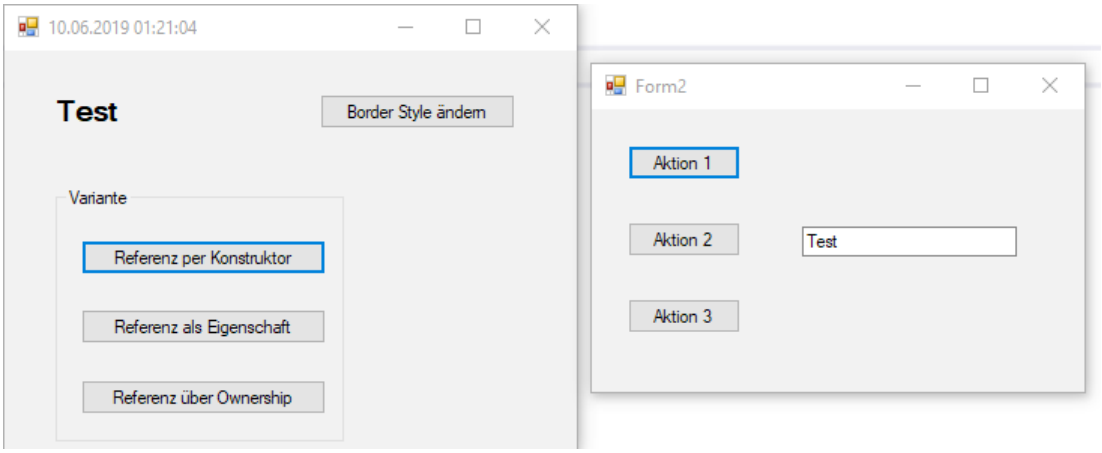
```
        private void Button3_Click(object sender, EventArgs e)
        {
            if (mainForm != null)
            {
                mainForm.Button1.PerformClick();
            }
            else
            {
                MessageBox.Show("Form1 Instanz nicht gesetzt!");
            }
        }
    }
```

Sie haben jetzt sicherlich bemerkt, dass alle *Click*-Events die Referenz für *m_Form* auf *null* testen und andernfalls einen Fehler auswerfen. Dies ist wichtig, weil der Benutzer von *Form2* sichergehen muss, dass eine gültige Referenz auf *Form1* vorliegt.

Nun müssen wir nur noch den Code von *Form1* ergänzen:

```
    private void Button2_Click(object sender, EventArgs e)
    {
            Form2 frm = new Form2(this);  // Form1-Referenz übergeben!
            AddOwnedForm(frm);       // frm zum Besitz von Form1 hinzufügen
            frm.Show();
    }
```

Starten Sie die Anwendung, so wird zunächst nur das Hauptformular angezeigt. Ein Klick auf *button2* lässt *Form2* erscheinen. Klicken Sie hier auf „Aktion 1“, so werden in der Titelleiste des Hauptformulars Datum und Uhrzeit angezeigt. Geben Sie irgendetwas in die *TextBox* ein, klicken Sie auf „Aktion 2“ und das *Label* des Hauptformulars wird den Inhalt übernehmen. Ein Klick auf „Aktion 3“ lässt die Umrandung des Hauptformulars verschwinden und beim nächsten Klick wieder erscheinen.



An dieser Stelle scheint eine Bemerkung zur *AddOwnedForm()*-Methode angebracht: Sie können deren Aufruf auch weglassen, müssen dann aber zum Beispiel in Kauf nehmen, dass nach dem Minimieren des Hauptformulars das untergeordnete Formular an seinem Platz verbleibt und nicht – wie in unserem Fall – komplett verschwindet, um nach Vergrößern des Hauptformulars wieder aufzutauchen. Auch liegt *Form2* immer auf *Form1*, kann also von diesem nicht verdeckt werden.

Variante 2: Übergabe der Formular-Referenz als Eigenschaft

Die zweite Variante weist sehr starke Ähnlichkeiten zur ersten Variante auf, weshalb wir uns auf eine verkürzte Darstellung beschränken können. *Form3* hält die *Form1*-Referenz ebenfalls in der privaten Zustandsvariablen *m_Form*, lediglich an die Stelle der Übergabe im Konstruktor tritt jetzt die Übergabe in einer WriteOnly-Eigenschaft *Form* vom Typ *Form1*:

```
public partial class Form3 : Form
{
        private Form1 mainForm;
        ...
        public Form1 Form
        {
            set
            {  mainForm = value; }
        }
        ...
}
```

Der restliche Code entspricht dem von *Form2*.

Nun müssen wir nur noch das *Click*-Event für *button3* des Hauptformulars hinzufügen, damit *Form3* angezeigt und getestet werden kann. Im Vergleich zur Vorgängervariante ist eine zusätzliche Codezeile erforderlich, in der die *Form1*-Referenz als *Form*-Eigenschaft dem untergeordneten Formular zugewiesen wird:

```
        private void Button3_Click(object sender, EventArgs e)
        {
            Form3 frm = new Form3();
            AddOwnedForm(frm);
            frm.Form = this;         // Eigenschaft setzen!
            frm.Show();
        }
```

Beim Testen gibt es erwartungsgemäß keinerlei Unterschiede zur ersten Variante.

Variante 3: Übergabe der Formular-Referenz als Ownership

Diese dritte und letzte Version ist leider weitaus weniger bekannt als die bis jetzt besprochenen Versionen. Benutzt wird das in Windows Forms (*System.Windows.Forms.Control*-Klasse) eingebaute Ownership-Verhalten, nach dem ein Control eines oder mehrere andere „besitzen“ kann. In unserem Fall müssen wir *Form4* zum „Besitz“ (Ownership) der übergeordneten *Form1* hinzufügen (auf die Merkmale von Ownership-Beziehungen wurde bereits im Zusammenhang mit der Version 1 bzw. der *AddOwnedForm()*-Methode eingegangen).

Der folgende Code für *Form5* funktioniert nur, wenn eine *Owner*-Eigenschaft vorliegt und in den korrekten Typ (*Form1*) gecastet werden kann. Anderenfalls wird eine Meldung ausgegeben.

```
public partial class Form4 : Form
{
    ...
    private void Button1_Click(object sender, EventArgs e)
    {
        if (Owner != null && Owner.GetType().Name == "Form1")
        {
            (Owner as Form1).MachWas();
        }
        else
        {
            MessageBox.Show("Kein Owner vorhanden!");
        }
    }

    private void Button2_Click(object sender, EventArgs e)
    {
        if (Owner != null && Owner.GetType().Name == "Form1")
        {
            (Owner as Form1).Label1.Text = textBox1.Text;
        }
        else
        {
            MessageBox.Show("Kein Owner vorhanden!");
        }
    }
```

```
    private void Button3_Click(object sender, EventArgs e)
    {
        if (Owner != null && Owner.GetType().Name == "Form1")
        {
            (Owner as Form1).Button1.PerformClick();
        }
        else
        {
            MessageBox.Show("Kein Owner vorhanden!");
        }
    }
}
```

Nun zum *Form1*-Code. Instanziierung und Aufruf von *Form4* sind ähnlich einfach wie bei den drei Vorgängerversionen:

```
    private void Button4_Click(object sender, EventArgs e)
    {
         Form4 frm = new Form4();
         AddOwnedForm(frm);   // darf nicht weggelassen werden!
         frm.Show();
    }
```

Auch der Test dieser Version führt zu exakt den gleichen Ergebnissen wie bei den Vorgängern.

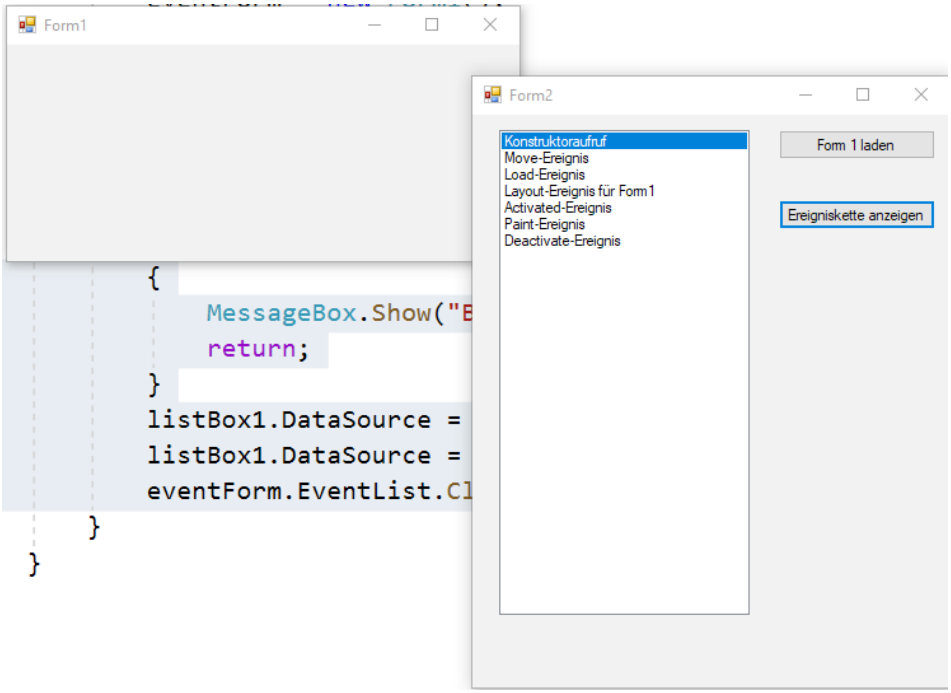
2.4.2 Ereigniskette beim Laden/Entladen eines Formulars

In welcher Reihenfolge werden die Ereignisse beim Laden bzw. Entladen eines Formulars ausgelöst? Das durch ein eigenes Experiment zu erkunden, wirkt nachhaltiger als das pure Auswendiglernen. Ganz nebenbei werden im vorliegenden Beispiel auch solche Fragen beantwortet: Wie ändere ich das Startformular? Wie rufe ich von *Form2* aus *Form1* auf?

Bedienoberfläche (Form1 und Form2)

Da das Startformular (*Form1*) hier als „Messobjekt" dient, brauchen wir noch ein zweites Formular (*Form2*), das unser „Messgerät" aufnimmt, im konkreten Fall eine *ListBox* als Ereignislogger und zwei Buttons, mit denen das Laden von *Form1* und die Anzeige der Ereigniskette gestartet werden. *Form1* hingegen bleibt „nackt", das heißt, seine Oberfläche enthält keinerlei Steuerelemente (siehe Laufzeitansicht).

Warum genügt uns nicht ein einziges Formular? Die Antwort ist einfach: Durch die auf dem Formular befindlichen Steuerelemente und die in die *ListBox* vorzunehmenden Einträge vergrößert sich die Anzahl der ausgelösten Ereignisse (*Layout*, *Paint*, …), die Sache wird unübersichtlich und eine klare Reihenfolge ist nur noch schwer zu erkennen.



Änderung des Startformulars

Nachdem Sie über das Menü **Projekt | Windows Form hinzufügen...** ein zweites Formular erzeugt haben, wollen Sie, dass dieses nach Start der Anwendung erscheint. Am einfachsten realisieren Sie das, indem Sie in der *Main*-Methode der Anwendung nicht mehr den Konstruktor von *Form1* aufrufen, sondern den von *Form2*. Um den Code der *Main*-Methode editieren zu können, müssen Sie im Projektmappen-Explorer auf die Datei *Program.cs* doppelklicken und dann die im Folgenden fettgedruckte Änderung vornehmen:

```
namespace _2._4._2_Ereigniskette
{
    static class Program
    {
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form2());
        }
    }
}
```

Quellcode von Form1

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

Eine generische Liste übernimmt das Zwischenspeichern des Ereignislogs. Die Liste ist *public*, damit von *Form2* aus der Inhalt auf möglichst einfache Weise gelesen werden kann:

```
    public List<string> EventList { get; } = new List<string>();
```

Der erste Zugriff auf die Liste erfolgt im Konstruktor des Formulars (vor Aufruf von *InitializeComponent()*!).

```
    public Form1()
    {
        EventList.Add("Konstruktoraufruf");
        InitializeComponent();
    }
```

Was jetzt kommt, können Sie sich leicht selbst zusammenreimen: Für jedes der interessierenden Formularereignisse schreiben Sie einen Eventhandler, dessen Rahmencode Sie wie üblich über die Ereignisse-Seite des Eigenschaftenfensters (F4) automatisch generieren lassen. Die Reihenfolge der folgenden Eventhandler ist unwichtig.

```
        private void Form1_Move(object sender, EventArgs e)
        {
            EventList.Add("Move-Ereignis");
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            EventList.Add("Load-Ereignis");
        }

        private void Form1_Layout(object sender, LayoutEventArgs e)
        {
           Control c = e.AffectedControl;
           EventList.Add ($"Layout-Ereignis für {c.Name}");
        }

        private void Form1_Activated(object sender, EventArgs e)
        {
            EventList.Add("Activated-Ereignis");
        }

        private void Form1_Paint(object sender, PaintEventArgs e)
        {
            EventList.Add( "Paint-Ereignis");
        }
        private void Form1_Resize(object sender, EventArgs e)
        {
            EventList.Add("Resize-Ereignis");
        }

        private void Form1_FormClosing(object sender, FormClosingEventArgs e)
        {
```

```
            EventList.Add("FormClosing-Ereignis");
        }

        private void Form1_FormClosed(object sender, FormClosedEventArgs e)
        {
            EventList.Add("FormClosed-Ereignis");
        }

        private void Form1_Deactivate(object sender, EventArgs e)
        {
            EventList.Add("Deactivate-Ereignis");
        }
    }
}
```

Quellcode von Form2

```
public partial class Form2 : Form
{ ...
```

Zunächst brauchen wir eine Referenz auf *Form1*:

```
    Form1 eventForm = null;
```

Nach Klick auf die erste Schaltfläche wird *Form1* erzeugt und geladen:

```
    private void Button1_Click(object sender, EventArgs e)
    {
        eventForm = new Form1();
        eventForm.Show();
    }
```

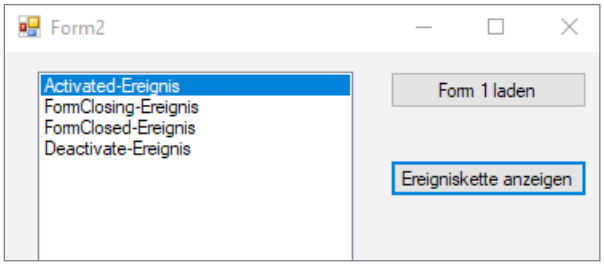
Ein Klick auf die zweite Schaltfläche bringt den Inhalt der Liste *EventList* aus *Form1* in der *ListBox* zur Anzeige:

```
    private void Button2_Click(object sender, EventArgs e)
    {
        if (eventForm==null)
        {
            MessageBox.Show("Bitte zuerst Form1 laden");
            return;
        }
        listBox1.DataSource = null;
        listBox1.DataSource = eventForm.EventList;
        eventForm.EventList.Clear();
    }
}
```

Test

Vorhang auf für die verschiedensten Experimente! Untersuchen Sie zunächst die Ereigniskette beim Laden des Formulars (siehe obige Abbildung). Dazu klicken Sie zunächst auf den linken und dann auf den rechten Button.

Nach dem Schließen von *Form1* klicken Sie erneut auf den rechten Button, um sich die Ereigniskette beim Entladen anzuschauen (siehe folgende Abbildung).



Verschieben Sie *Form1*, verändern Sie die Abmessungen, wechseln Sie den Fokus mit anderen Formularen – all diese Manipulationen finden ihren Niederschlag in einer typischen Ereigniskette.

3 Windows Forms-Komponenten

Nachdem Sie in den beiden vorhergehenden Kapiteln bereits die Grundlagen der Windows-Formulare kennengelernt haben, soll es Ziel dieses Kapitels sein, Ihnen einen Überblick über die wichtigsten visuellen Steuerelemente von Windows Forms-Anwendungen zu verschaffen. Da dieses Kapitel keine vollständige Referenz bereitstellt – diese Rolle kann die Ihnen zur Verfügung stehende Dokumentation viel effektiver übernehmen –, werden nur die aus der Sicht des Praktikers wichtigsten Eigenschaften, Ereignisse und Methoden in Gestalt von Übersichten und knappen Beispielen vorgestellt.

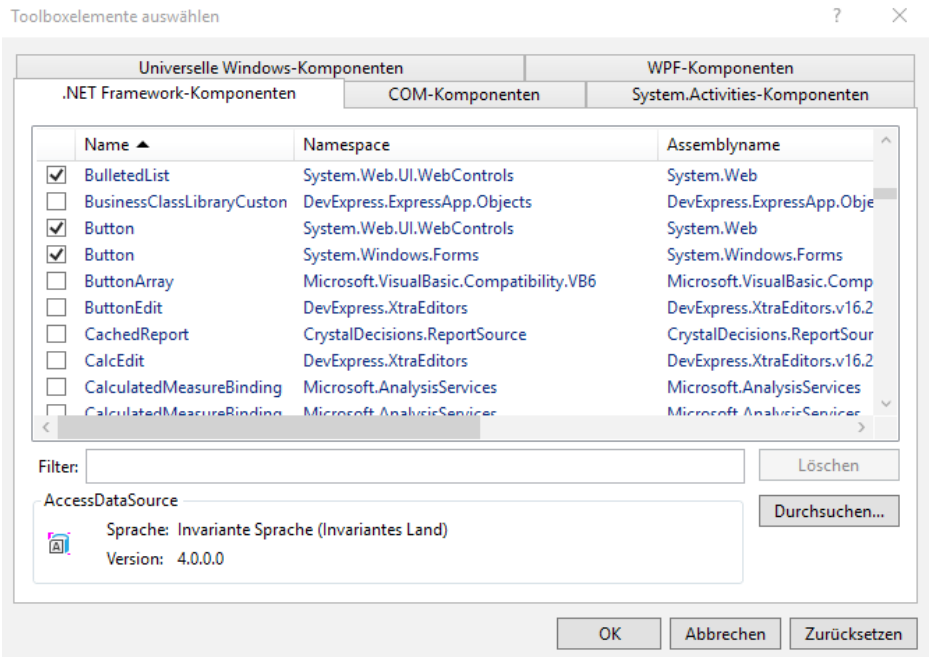
3.1 Allgemeine Hinweise

Visual Studio bietet Ihnen eine hervorragende Unterstützung bei der Entwicklung grafischer Benutzeroberflächen. Das grundlegende Handwerkszeug (Toolbox, Eigenschaftenfenster, …) wurde Ihnen bereits in Online-Kapitel 1 im ausreichenden Umfang vermittelt. Die Vorgehensweise beim Oberflächenentwurf ist so intuitiv, dass sich weitere Erklärungen erübrigen. Stattdessen folgen einige grundsätzliche Hinweise.

3.1.1 Hinzufügen von Komponenten

Die Anzahl der Komponenten hat mit jeder neuen Version des .NET Frameworks zugenommen, sodass es aus Gründen der Übersichtlichkeit nicht mehr zu empfehlen ist, alle zusammen im Werkzeugkasten anzubieten. Außerdem wurden im Laufe der Zeit einige Steuerelemente durch neuere ersetzt (z. B. *MainMenu/ContextMenu* durch *MenuStrip/ContextMenuStrip*, *DataGrid* durch *DataGridView*, …). Die älteren Versionen müssen aber aus Gründen der Abwärtskompatibilität auch weiterhin zur Verfügung stehen, sollten aber zunächst nicht mehr in der Toolbox auftauchen.

Wenn Sie das standardmäßige Angebot ändern wollen, können Sie dem Werkzeugkasten weitere Steuerelemente hinzufügen oder welche davon entfernen. Über der entsprechenden Kategorie des Werkzeugkastens klicken Sie im Kontextmenü *Elemente auswählen …* Im Dialog „Werkzeugkastenelemente auswählen“ markieren Sie die gewünschten Steuerelemente bzw. entfernen bestimmte Häkchen.



3.1.2 Komponenten zur Laufzeit per Code erzeugen

Grundsätzlich haben Sie jederzeit die Möglichkeit, auf die Dienste des visuellen Designers zu verzichten und stattdessen den Code zur Erzeugung des Steuerelements eigenhändig zu schreiben. Obwohl das scheinbar mit einiger Mehrarbeit verbunden ist, kann es in einigen Fällen durchaus sinnvoll sein (z. B. beim Erzeugen von Steuerelemente-Arrays).

Beispiel 3.1: Ein Array mit drei Schaltflächen per Code erzeugen

C#

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

Die Deklaration der Buttonliste:

```
    private List<Button> buttons = new List<Button>();
```

Steuerelemente werden im Konstruktorcode erzeugt:

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

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

Die Buttonliste befüllen:

```
        for (int i = 0; i < 3; i++)
        {
```

Einen Button „nackt“ erzeugen:

```
            Button button = new Button();
            buttons.Add(button);
```

Eigenschaften zuweisen:

```
            button.Bounds = new Rectangle(new Point(10 + i*100, 10),
                                          new Size(100, 50));
            button.Text = $"Button{i}";
```

Gemeinsame Ereignisbehandlung anmelden:

```
            button.Click += new EventHandler(button_Click);
        }
```

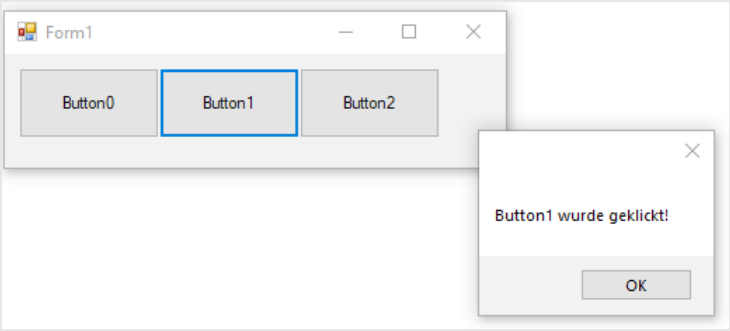
Alle Buttons zum Formular hinzufügen:

```
        Controls.AddRange(buttons.ToArray());
    }
```

Gemeinsamer Eventhandler für *Click*-Ereignis:

```
    private void button_Click(object sender, EventArgs e)
    {
        Button btn = (Button)sender;
        MessageBox.Show($"{btn.Text} wurde geklickt!");
    }
}
```

Ergebnis



Alle Buttons sind exakt nebeneinander ausgerichtet. Das Vergrößern der Liste ist mit keinerlei Mehraufwand an Code verbunden.

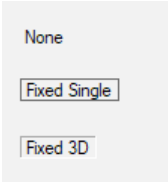
HINWEIS: Die sinnvolle Anwendung eines *CheckBox*-Arrays wird im Praxisbeispiel in Abschnitt 3.9.2 gezeigt!

3.2 Allgemeine Steuerelemente

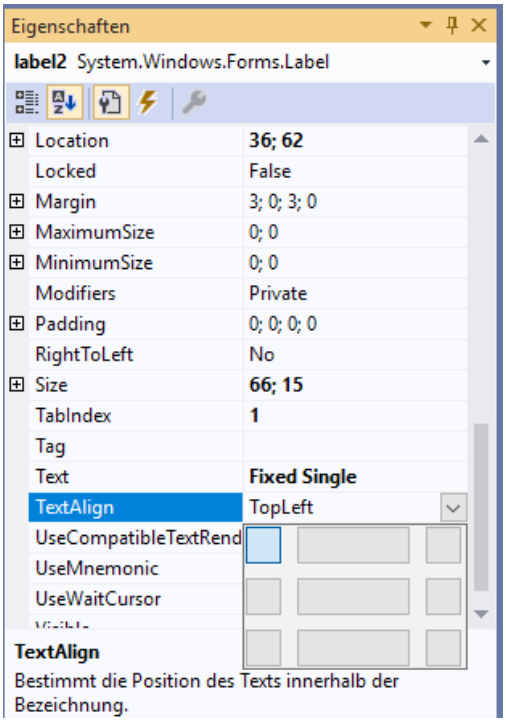
Diese Toolbox-Kategorie enthält standardmäßig die normalerweise am häufigsten benötigten Steuerelemente.

3.2.1 Label

Das harmlose, aber unverzichtbare *Label* dient, im Gegensatz zur *TextBox*, nur zur Anzeige von statischem (unveränderbarem) Text (*Text*-Property). Mit *BorderStyle* haben Sie die Wahl zwischen drei Erscheinungsbildern:

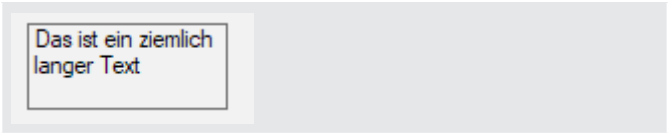


Hervorzuheben ist die *TextAlign*-Eigenschaft, welche die Ausrichtung des Textes auf insgesamt neun verschiedene Arten ermittelt bzw. setzt. Auch hier steht Ihnen ein komfortabler Property-Editor zur Verfügung:



Wenn die *AutoSize*-Eigenschaft auf *True* gesetzt wird, passt sich die Breite des Labels dem aufzunehmenden Text an. Bei *False* ist auch die Anzeige von mehrzeiligem Text möglich.

Beispiel 3.2: Der Zeilenumbruch bei *AutoSize* = *False* erfolgt „intelligent“, also beim nächsten passenden Leerzeichen.



HINWEIS: Möchten Sie ein &-Zeichen im *Label* verwenden, können Sie entweder die Eigenschaft *UseMnemonic* auf *False* setzen oder Sie müssen zwei &-Zeichen einfügen.

3.2.2 LinkLabel

Im Grunde handelt es sich bei dieser Komponente um ein *Label*, das mit etwas Funktionalität ergänzt wurde, um einen Hyperlink nachzubilden.

Konzeptionell kann dieses Control leider nur als missglückt angesehen werden. Mit viel Aufwand und Verspieltheit (Sie können mehrere Hyperlinks innerhalb des Controls definieren) wurde am Problem vorbei programmiert. Statt einer simplen *Link*-Eigenschaft, die den Hyperlink enthält und die sich auch zur Entwurfszeit zuweisen lässt, wurde noch eine Klasse integriert, die sich nur zur Laufzeit sinnvoll ansprechen lässt. Zu allem Überfluss muss auch noch die eigentliche Funktionalität, das heißt der Aufruf des Hyperlinks, selbst programmiert werden. Da ist man mit einem einfachen Label fast schneller am Ziel.

Wichtige Eigenschaften

Zur Gestaltung des optischen Erscheinungsbilds können Sie folgende Eigenschaften verwenden:

- *ActiveLinkColor* (der Hyperlink ist aktiv)
- *DisabledLinkColor* (der Hyperlink ist gesperrt)
- *LinkColor* (die Standardfarbe)
- *VisitedLinkColor* (der Hyperlink wurde bereits angeklickt)
- *LinkBehavior* (wie bzw. wann wird der Hyperlink unterstrichen)

Hyperlink einfügen

Verwenden Sie die *Links.Add*-Methode, um zur Laufzeit einen Hyperlink in das Control einzufügen. Übergabewerte sind der Bereich des Hyperlinks bezüglich der *Text*-Eigenschaft und der URL.

Beispiel 3.3: Aufruf der MS-Homepage

C#

Platzieren Sie einen Hyperlink mit dem Text „Microsoft im Internet“ auf dem Formular.

Weisen Sie im Form-Load-Ereignis den Hyperlink zu:

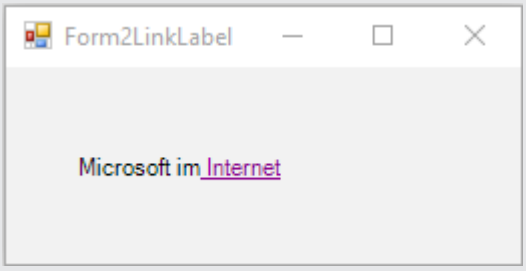
```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    private void Form2LinkLabel_Load(object sender, EventArgs e)
    {
        linkLabel1.Links.Add(12, 9, "www.microsoft.com");
    }
```

Die Zahlenangaben bewirken, dass lediglich das Wort „Internet“ als Hyperlink verwendet wird.

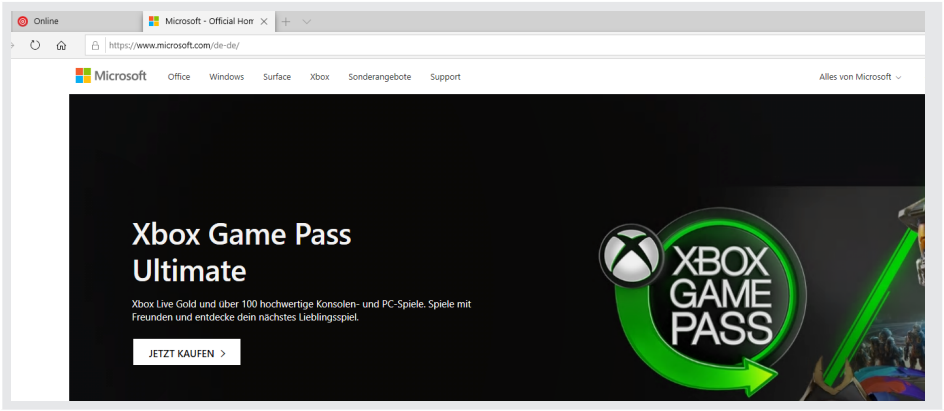
Mit dem Klick auf den *LinkLabel* wird das *LinkClicked*-Ereignis ausgelöst. Hier können Sie der Komponente zum einen mitteilen, dass der Hyperlink besucht wurde, zum anderen sind Sie dafür verantwortlich, den URL auszuwerten (wird im Parameter *e* übergeben) und aufzurufen:

```
    private void LinkLabel1_LinkClicked(object sender,
                 LinkLabelLinkClickedEventArgs e)
    {
        linkLabel1.Links[linkLabel1.Links.IndexOf(e.Link)].Visited = true;
        System.Diagnostics.Process.Start(e.Link.LinkData.ToString());
    }
}
```

Ergebnis



Wie Sie sehen, erhält der eigentliche Hyperlink auch den Fokus, Sie können also auch mit der *Tabulator*- und der *Enter*-Taste arbeiten.



3.2.3 Button

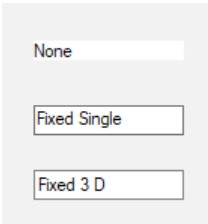
Dieses Steuerelement ist wohl aus (fast) keiner Applikation wegzudenken, über die Funktionalität brauchen wir deshalb kaum Worte zu verlieren. Die folgende Abbildung zeigt vier Vertreter dieser Gattung mit unterschiedlich gesetzter *FlatStyle*-Eigenschaft:

HINWEIS: Entgegen der üblichen Vorgehensweise legen Sie die *Default*-Taste in Dialogboxen nicht mehr über eine Eigenschaft des Buttons, sondern über die *AcceptButton*-Eigenschaft des Formulars fest. Das Gleiche gilt für *CancelButton*.

Mit der *Image*-Eigenschaft bietet sich ein weiteres gestalterisches Mittel, um die trostlosen Buttons etwas aufzupeppen. Alternativ können Sie als Quelle der Grafik auch eine *ImageList* verwenden, in diesem Fall wählen Sie die Grafik mit der *ImageIndex*-Eigenschaft aus.

3.2.4 TextBox

Im Unterschied zum *Label* besteht hier die Möglichkeit, den Text zur Laufzeit zu editieren oder zu markieren. All dies geschieht durch Zugriff auf die *Text*-Eigenschaft. Bei mehrzeiligem Text können Sie über die *Lines*-Eigenschaft auf die einzelnen Textzeilen zugreifen. Das äußere Erscheinungsbild wird, wie beim Label, im Wesentlichen durch die *BorderStyle*-Eigenschaft bestimmt:



Hervorzuheben sind weiterhin folgende Properties:

Eigenschaft	Beschreibung
AutoCompleteSource *AutoCompleteMode* *AutoComplete-CustomSource*	… ermöglichen automatisches Vervollständigen des einzugebenden Textes
TextAlign	… bestimmt die Ausrichtung des Textes (*Left*, *Right*, *Center*)
MaxLength	… legt die maximale Anzahl einzugebender Zeichen fest (Standardeinstellung 32.767 Zeichen)
ReadOnly	… das Control ist schreibgeschützt.

Mehrzeilige Textboxen

Wichtige Eigenschaften in diesem Zusammenhang:

Eigenschaft	Beschreibung
MultiLine	… erlaubt die Eingabe mehrzeiliger Texte (*True*). Für diesen Fall ist auch eine vertikale Scrollbar sinnvoll.
ScrollBars	… bestimmt, ob Bildlaufleisten enthalten sind. Die Eigenschaft zeigt nur bei *MultiLine=True* Wirkung.
Accepts-Return	Ist diese Eigenschaft *True*, so können Sie mittels Enter-Taste einen Zeilenumbruch einfügen. Ein eventuell vorhandener *AcceptButton* wird damit außer Kraft gesetzt! Bleibt *WantReturns* auf *False,* müssten Sie nach wie vor *Strg+Enter* für einen Zeilenumbruch verwenden.
WordWrap	Damit bestimmen Sie, ob der Text im Eingabefeld am rechten Rand umgebrochen wird (*True*). Der Umbruch wird lediglich auf dem Bildschirm angezeigt, der Text selbst enthält keinerlei Zeilenumbrüche, die nicht eingegeben wurden. Wenn *WordWrap False* ist, entsteht eine neue Zeile nur dort, wo auch ein Zeilenumbruch in den Text eingefügt wurde.
Lines	Zwar gibt es auch eine *Text*-Eigenschaft, doch ist diese für die praktische Arbeit weniger gut geeignet. Sie arbeiten besser mit der *Lines*-Eigenschaft, die einen gezielten Zugriff auf einzelne Zeilen gestattet und die Sie im Stringlisten-Editor oder auch per Quellcode zuweisen können, z. B. Auslesen einer Zeile: `textBox2.Text = textBox1.Lines[3];`

Markieren von Text

SelectionLength, *SelectionStart*, *SelectedText* gelten für markierte Textausschnitte. *SelectionLength* bestimmt bzw. liefert die Zeichenzahl, *SelectionStart* ermittelt die Anfangsposition und *SelectedText* setzt bzw. ermittelt den Inhalt.

HINWEIS: Auf diese Properties kann nur zur Laufzeit zugegriffen werden, sie befinden sich **nicht** im Eigenschaftenfenster!

Alternativ können Sie auch die Methoden *SelectAll* bzw. *Select* verwenden.

Beispiel 3.4: Wenn man mit der Tab-Taste zur *TextBox* wechselt, wird das erste Zeichen markiert.

C#

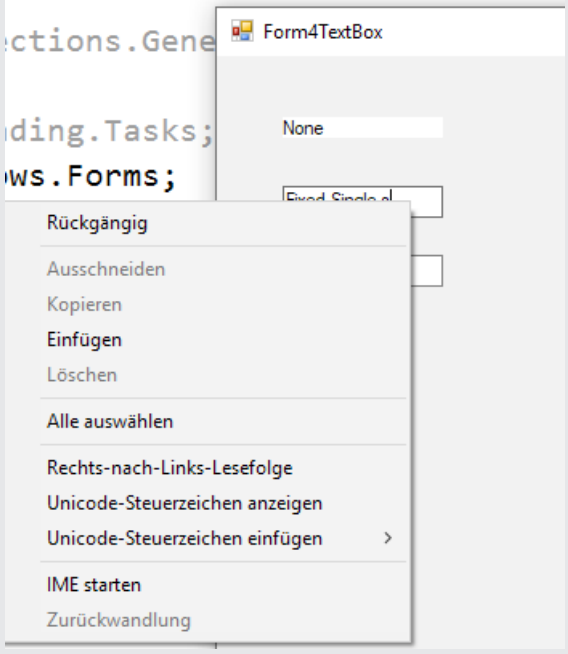
```
private void TextBox1_Enter(object sender, EventArgs e);
{
   textBox1.Select(0, 1);
}
```

oder

```
   textBox1.SelectionStart  = 0;
   textBox1.SelectionLength = 1;
```

Ergebnis

Übrigens können Sie zur Laufzeit für jedes Editierfeld ein umfangreiches Kontextmenü aufrufen, über das die wichtigsten Operationen direkt ausführbar sind:



PasswordChar

Diese Eigenschaft erlaubt das verdeckte Eingeben eines Passworts. Sie können das gewünschte Zeichen im Eigenschaften-Fenster oder per Quellcode zuweisen.

Beispiel 3.5: Passworteingabe

C#

```
textBox3.PasswordChar = '*';
```

Ergebnis

Automatisches Vervollständigen

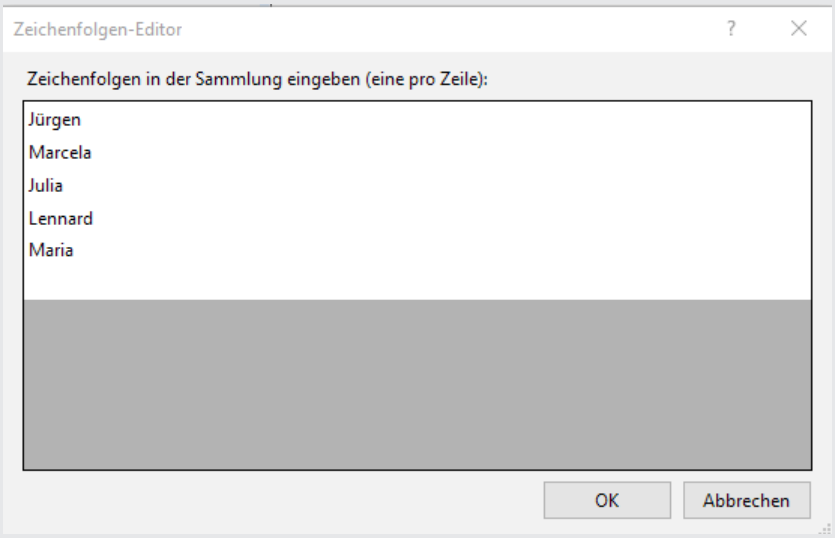
Die Eigenschaften *AutoCompleteCustomSource*, *AutoCompleteSource* und *AutoCompleteMode* einer *TextBox* bewirken das automatische Vervollständigen des eingegebenen Textes durch Vergleich mit einer vorhandenen Liste. Das ist besonders vorteilhaft bei sich häufig wiederholenden Eingabewerten, wie z. B. URLs, Adressen oder Dateinamen.

Voraussetzung für die Verwendung der *AutoCompleteCustomSource*-Eigenschaft ist, dass Sie der *AutoCompleteSource*-Eigenschaft den Wert *CustomSource* zuweisen. Das „Wie" der Textvervollständigung legen Sie mit der *AutoCompleteMode*-Eigenschaft (*None*, *Suggest*, *Append*, *SuggestAppend*) fest.

Beispiel 3.6: Automatisches Vervollständigen

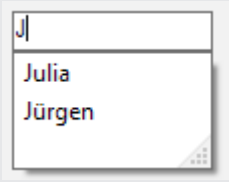
Entwurf

Einer *TextBox* (*AutoCompleteSource* = *CustomSource*, *AutoCompleteMode* = *Suggest*) wird die folgende *AutoCompleteCustomSource*-Eigenschaft zugewiesen:



Ergebnis

Nach Eingabe eines Buchstabens öffnet sich eine Liste mit passenden Einträgen (Auswahl über Maus oder Cursortasten):



HINWEIS: *AutoComplete* gibt es auch für die *ComboBox*!

3.2.5 MaskedTextBox

Dieses von der abstrakten *TextBoxBase* abgeleitete Steuerelement benutzt eine Maske, die es erlaubt, Benutzereingaben zu filtern.

Von zentraler Bedeutung ist, wen wundert es, die *Mask*-Eigenschaft. Dies ist ein String, der sich aus bestimmten Zeichen zusammensetzt, von denen die wichtigsten in der folgenden Tabelle erklärt werden. Einige Zeichen sind optional, sie können durch Eingabe eines Leerzeichens übergangen werden.

Zeichen	Beschreibung	Zeichen	Beschreibung
0	Ziffer zwischen 0 und 9	.	Dezimal-Trennzeichen
9	Ziffer oder Leerzeichen, optional	,	Tausender-Trennzeichen
#	Ziffer oder Leerzeichen mit Plus (+) oder Minus (-), optional	:	Zeit-Trennzeichen
L	Buchstabe (a-z, A-Z)	/	Datums-Trennzeichen
?	Buchstabe (a-z, A-Z), optional	<	Konvertiert alle folgenden Zeichen in Kleinbuchstaben
$	Währungssymbol	>	Konvertiert alle folgenden Zeichen in Großbuchstaben
&	Alphanumerisches Zeichen		

Dezimal- und Tausender-Trennzeichen sowie die Symbole für Datum, Zeit und Währung, entsprechen der standardmäßig eingestellten Kultur.

Beispiel 3.7: *MaskedTextBox*

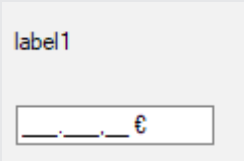
C#

Ein Währungswert von 0 bis 999999 wird in eine *MaskedTextBox* eingegeben. Ist die Eingabe komplett, so wird der Inhalt in ein *Label* übernommen. Währungssymbol, Dezimal- und Tausender-Trennzeichen werden zur Laufzeit durch die kulturspezifischen Einstellungen ersetzt.

```
maskedTextBox1.Mask = "999,999.00 $";
...
private void MaskedTextBox1_TextChanged(object sender, EventArgs e)
   if (maskedTextBox1.MaskCompleted)
   {
      label1.Text = maskedTextBox1.Text;
   }
```

Ergebnis

Vor Beginn der Eingabe:



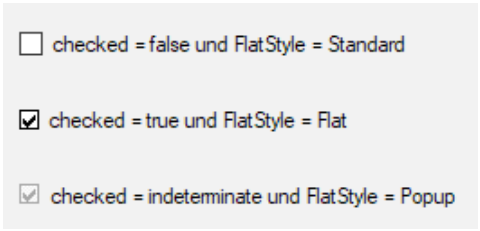
Nach der Eingabe:

HINWEIS: Eine Leerzeichenkette als Maske wird den vorhandenen Inhalt unverändert belassen, die *MaskedTextBox* benimmt sich dann wie eine einzeilige „normale“ *TextBox*.

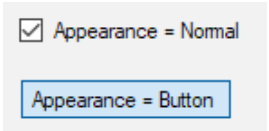
3.2.6 CheckBox

Bei der *CheckBox* entscheidet die *Checked*-Eigenschaft (*True/False*) darüber, ob das Häkchen gesetzt wurde oder nicht. Das äußere Erscheinungsbild kann über die *FlatStyle*-Eigenschaft geringfügig modifiziert werden. Für den Programmierer bietet sich zusätzlich die Möglichkeit, über *CheckState* alle drei Zustände zu bestimmen:

- *Checked*
- *Indeterminate*
- *Unchecked*



Die *Appearance*-Eigenschaft erlaubt es Ihnen, die *CheckBox* optisch in einen Button zu verwandeln, das Verhalten bleibt jedoch gleich:



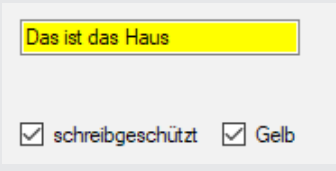
Eine Änderung der *Checked*-Eigenschaft löst das Ereignis *CheckedChanged* aus.

Beispiel 3.8: Mit zwei CheckBoxen werden Schreibschutz und Hintergrundfarbe einer *TextBox* eingestellt.

C#

```
private void CheckBox_CheckedChanged(object sender, EventArgs e)
{
     if (chkReadOnly.Checked)
     {
         textBox1.ReadOnly = true;
     }
     else
     {
         textBox1.ReadOnly = false;
     }

     if (chkYellow.Checked)
     {
         textBox1.BackColor = Color.Yellow;
     }
     else
     {
         textBox1.BackColor = Color.White;
     }
}
```

Ergebnis



Eine weitere interessante Eigenschaft ist *AutoCheck*. Ändert man ihren Wert von *True* nach *False*, so wird der Aktivierungszustand der *CheckBox* bei einem Klick nicht mehr automatisch umgeschaltet und der Programmierer muss sich selbst darum kümmern, z. B. durch Auswerten des *Click*-Ereignisses.

Beispiel 3.9: *Click*-Eventhandler einer *CheckBox* bei *AutoCheck* = *False*

C#

```
private void CheckBox1_Click(object sender, EventArgs e)
{
    checkBox1.Checked = !checkBox1.Checked;
```

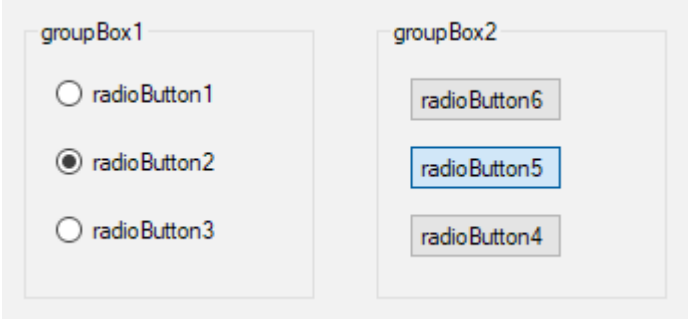
oder allgemeiner:

```
     CheckBox cb = (CheckBox)sender;
     cb.Checked = !cb.Checked;
}
```

3.2.7 RadioButton

Dieses Steuerelement dient zur Auswahl von Optionen innerhalb einer Anwendung. Im Unterschied zur *CheckBox* kann aber innerhalb einer Gruppe immer nur ein einziger *RadioButton* aktiv sein.

Meist fasst man mehrere *RadioButtons* mittels *GroupBox* (oder *Panel*) zu einer Optionsgruppe zusammen. Auch bei dieser Komponente können Sie die *Appearance*-Eigenschaft dazu nutzen, das Erscheinungsbild so zu ändern, dass für jeden *RadioButton* ein *Button* dargestellt wird:



Ausgewertet wird der Status über die *Checked*-Eigenschaft.

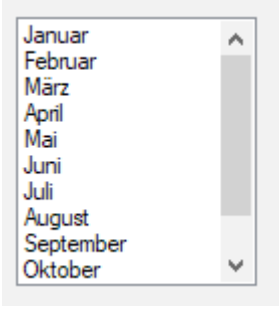
Beispiel 3.10: Ändern der Hintergrundfarbe des Formulars

C#

```
private void RadioButton_CheckedChanged(object sender, EventArgs e)
{
    if (radioButton1.Checked)
    {
        BackColor = Color.White;
    }
    else if (radioButton2.Checked)
    {
        BackColor = Color.Yellow;
    }
    else if (radioButton3.Checked)
    {
         BackColor = Color.Red;
    }
}
```

3.2.8 ListBox

In einer *ListBox* kann eine Auflistung von Einträgen angezeigt werden, von denen der Benutzer mittels Maus oder Tastatur einen oder auch mehrere auswählen kann.



Die wichtigsten Eigenschaften zeigt die folgende Tabelle, auf datenbezogene Eigenschaften wird später näher eingegangen.

Eigenschaft	Beschreibung
Items *Items.Count*	… ist die Liste der enthaltenen Einträge, die Sie zur Entwurfszeit auch über den Zeichenfolgen-Editor eingeben können. Über *Items.Count* können Sie die Anzahl bestimmen.
Sorted	… legt fest, ob die Einträge alphabetisch geordnet erscheinen sollen (*True*/*False*)
SelectedIndex	… setzt bzw. ermittelt die Position (Index) des aktuellen Eintrags (-1, wenn nichts ausgewählt wurde)
SelectionMode	… entscheidet, ob Einzel- oder Mehrfachauswahl zulässig ist
SelectedItem	… der Text des ausgewählten Eintrags

Die Methoden *Items.Add* und *Items.Remove/ItemsRemoveAt* fügen Einträge hinzu bzw. entfernen sie, *Items.Clear* löscht den gesamten Inhalt.

Beispiel 3.11: In die *ListBox* wird zehnmal „Hallo“ eingetragen, anschließend wird der zweite Eintrag über seinen Index gelöscht.

C#

```
private void Form8ListBox_Load(object sender, EventArgs e)
{
    for (int i = 1; i < 11; i++)
    {
        listBox2.Items.Add($"{i} Hallo");
    }
    listBox2.Items.RemoveAt(1);
```

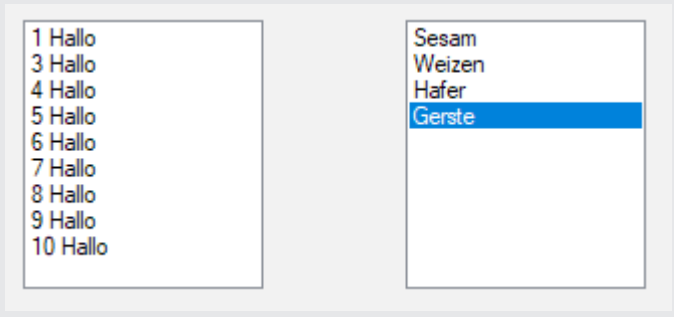
Häufig will man auch den Inhalt eines (eindimensionalen) Arrays anzeigen.

Beispiel 3.12: Der Inhalt eines *String*-Arrays wird in eine *ListBox* übertragen.

C#

```
    string[] a = {"Sesam", "Weizen", "Hafer", "Gerste"};
    for (int i = 0; i < getreide.Length; i++)
    {
         listBox3.Items.Add(getreide[i]);
    }
    listBox3.SelectedIndex = 3;
}
```

Ergebnis



Für den Programmierer ist das *SelectedIndexChanged*-Ereignis von besonderem Interesse, da es bei jedem Wechsel zwischen den Einträgen aufgerufen wird.

Beispiel 3.13: Nach Auswahl eines Eintrags aus einem Listenfeld wird dieser in eine *Textbox* übernommen.

C#

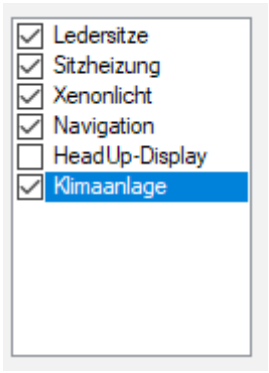
```
private void ListBox1_SelectedIndexChanged(object sender, EventArgs e);
{
   textBox1.Text = listBox1.SelectedItem.ToString();
}
```

Ob der Wechsel mittels Maus oder Tastatur erfolgt, ist in diesem Fall egal.

HINWEIS: Ist die Eigenschaft *SelectionMode* auf *MultiSimple* oder *MultiExtended* festgelegt, können Sie auch mehrere Einträge gleichzeitig auswählen. Über *SelectedItems* greifen Sie auf diese Einträge zu.

3.2.9 CheckedListBox

Hierbei handelt es sich um einen nahen Verwandten der „gemeinen" Listbox mit dem Unterschied, dass die einzelnen Einträge gleichzeitig einen *True/False*-Wert (*Checked*) verwalten können.



Für das Setzen der Häkchen per Programm verwenden Sie die *SetItemChecked*-Methode.

Beispiel 3.14: Häkchen beim vierten Eintrag setzen

C#

```
checkedListBox1.SetItemChecked(3, true);
```

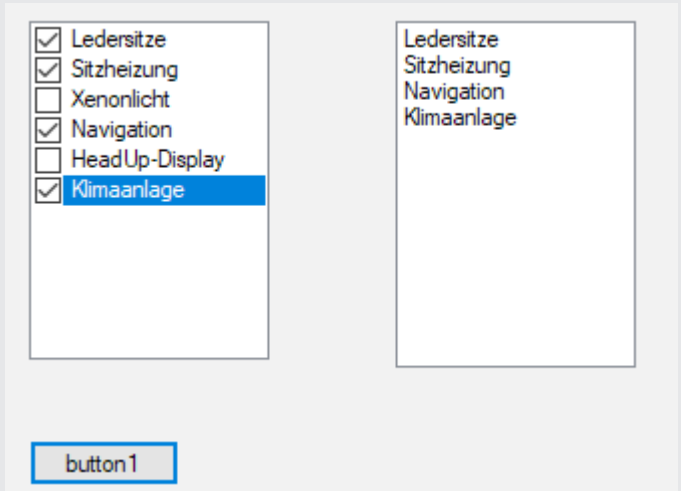
Über die *CheckedItems*-Collection haben Sie Zugriff auf alle markierten Einträge.

Beispiel 3.15: Anzeige aller markierten Einträge in einer weiteren *ListBox*

C#

```
private void Button1_Click(object sender, EventArgs e)
{
    for (int i = 0; i < checkedListBox1.CheckedItems.Count; i++)
    {
        listBox1.Items.Add(checkedListBox1.CheckedItems[i]);
    }
}
```

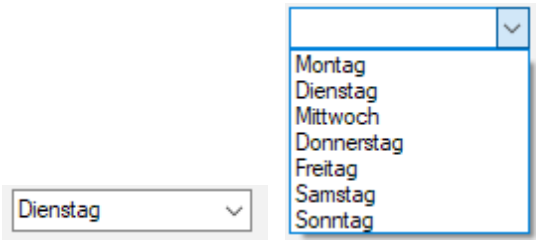
Ergebnis



HINWEIS: Da das Standardverhalten der *CheckedListBox* recht gewöhnungsbedürftig sein dürfte, sollten Sie besser die Eigenschaft *CheckOnClick* auf *True* setzen, damit nicht zweimal geklickt werden muss.

3.2.10 ComboBox

Eine *ComboBox* ist eine Mischung aus Text- und Listenfeld. Sie erlaubt also Eingaben und kann, im Unterschied zum Listenfeld, auch „aufgeklappt“ werden:



Hervorzuheben sind folgende Eigenschaften:

Eigenschaft	Beschreibung
DropDownStyle	… *Simple* (nur Texteingabe), *DropDown* (Texteingabe und Listenauswahl), *DropDownList* (nur Listenauswahl)
Items *Items.Count*	… ist die Liste der angezeigten Werte. Über *Items.Count* können Sie die Anzahl bestimmen.

MaxDropDownItems	… die maximale Höhe der aufgeklappten Listbox in Einträgen
Sorted	... legt fest, ob die Einträge alphabetisch geordnet erscheinen sollen (*True*/*False*)
SelectedIndex	… setzt bzw. ermittelt die Position (Index) des aktuellen Eintrags (-1, wenn nichts ausgewählt wurde)
SelectionMode	… entscheidet, ob Einzel- oder Mehrfachauswahl zulässig ist
Text	… der Text des ausgewählten Eintrags

Die Methoden *Items.Add* und *Items.Remove/ItemsRemoveAt* fügen Einträge hinzu bzw. entfernen sie, *Items.Clear* löscht den gesamten Inhalt.

Beispiel 3.16: *ComboBox* mit den Namen aller Monate füllen und den ersten Eintrag anzeigen

C#

```
string[] monate = { "Januar", "Februar", "März", "April", "Mai", "Juni",
     "Juli", "August", "September", "Oktober", "November", "Dezember" };
comboBox2.Items.AddRange(monate);
comboBox2.SelectedIndex = 0;          // zeigt "Januar"
```

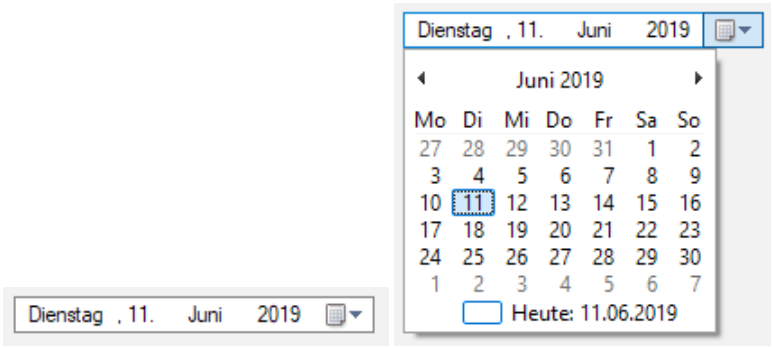
3.2.11 PictureBox

Dieses Control dient der Darstellung von fertigen Grafiken diverser Formate (BMP, GIF, TIFF, PNG …). Außerdem kann mit Grafikmethoden in die *PictureBox* gezeichnet werden, weshalb man sie oft auch als „kleine Schwester“ des Formulars bezeichnet. Mittels *Image*-Eigenschaft können Sie diesem Control direkt eine Bildressource zuweisen. Über die *SizeMode*-Eigenschaft steuern Sie, wie die Grafik in den Clientbereich des Controls eingepasst wird bzw. ob sich das Control an die Grafik anpasst.

HINWEIS: Mehr zu dieser ebenso interessanten wie leistungsfähigen Komponente erfahren Sie im Online-Kapitel „Grafikprogrammierung“ (Abschnitt 4.2.1).

3.2.12 DateTimePicker

Geht es um die platzsparende Auswahl eines Datums- oder eines Zeitwerts, sollten Sie sich mit der *DateTimePicker*-Komponente anfreunden. Diese funktioniert wie eine *ComboBox*, nach dem Aufklappen steht Ihnen ein recht komfortabler Kalender zur Verfügung:



Alternativ kann auch nur eine Uhrzeit bearbeitet werden (Eigenschaft *Format* = *Time*):

Auf alle Möglichkeiten der Konfiguration einzugehen, dürfte den Rahmen dieses Kapitels sprengen. Die wohl wichtigsten Eigenschaften dürften *MinDate*, *MaxDate* für die Beschränkung der Auswahl bzw. *Value* für den Inhalt des Controls sein.

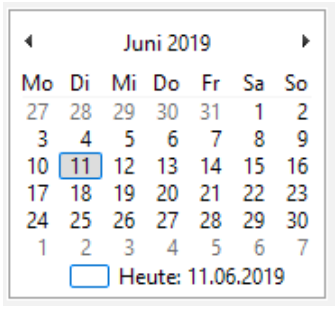
Beispiel 3.17: Anzeige des Auswahlwerts

C#

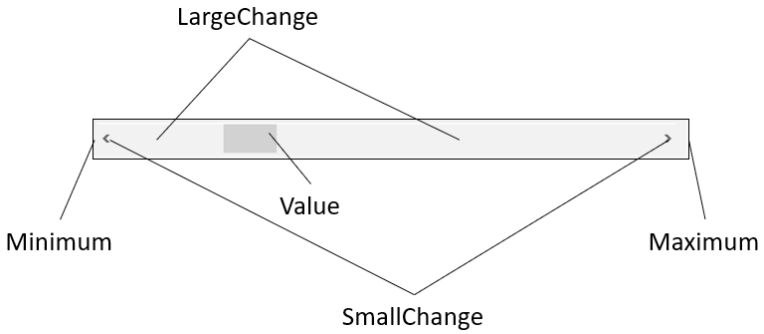
```
private void DateTimePicker1_ValueChanged(object sender, EventArgs e)
{
   MessageBox.Show(dateTimePicker1.Value.ToLongDateString());
}
```

3.2.13 MonthCalendar

Funktionell ist diese Komponente dem *DateTimePicker* sehr ähnlich, der wesentliche Unterschied besteht darin, dass diese Komponente nicht auf- und zugeklappt werden kann und damit natürlich auch mehr Platz auf dem Formular verbraucht:



3.2.14 HScrollBar, VScrollBar



Diese Komponenten werden häufig zum Scrollen von Bild- und Fensterinhalten verwendet.

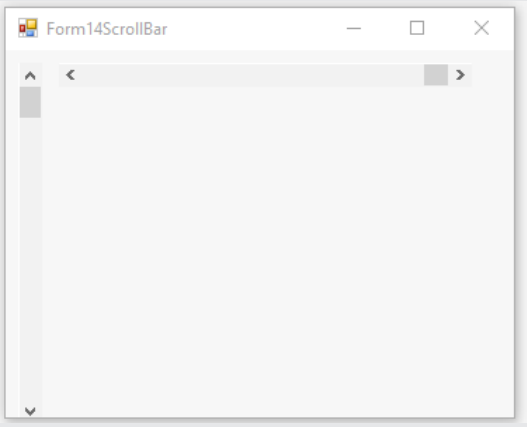
- *Maximum*, *Minimum*, *Value*
  Sie legen den größten bzw. kleinsten Einstellungswert fest bzw. bestimmen den aktuellen Wert (zwischen *Minimum* und *Maximum*).
- *LargeChange*, *SmallChange*
  Klickt man neben den „Schieber", so wird der aktuelle Wert (*Value*) um *LargeChange* geändert, beim Klicken auf die Begrenzungspfeile hingegen nur um *SmallChange*.
- Eine wichtige Rolle spielt auch das *Scroll*-Ereignis, das immer dann eintritt, wenn die Position des Schiebers verändert wurde (*e.NewValue* enthält den neu gewählten Wert).

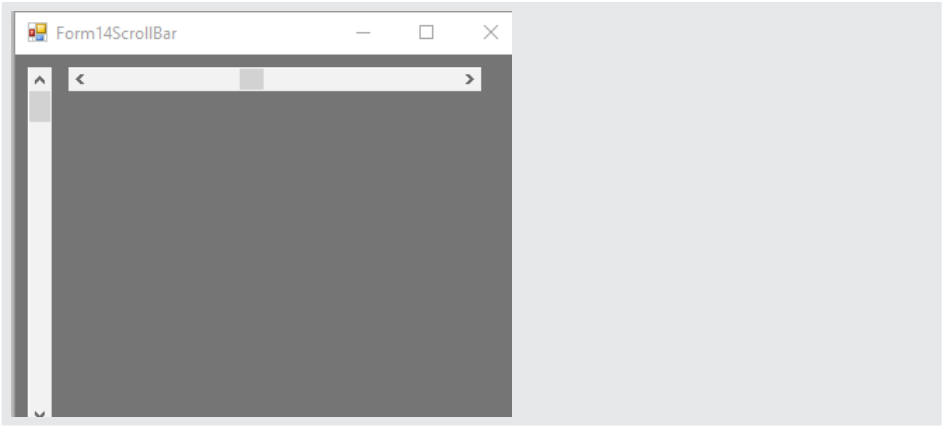
Beispiel 3.18: Einstellen der Formularhintergrundfarbe (Graustufe) mit einer horizontalen Scrollbar (*Minimum = 0, Maximum = 255*)

C#

```
private void hScrollBar1_Scroll(object sender, ScrollEventArgs e)
{
    BackColor = Color.FromArgb(255, e.NewValue, e.NewValue, e.NewValue);
}
```

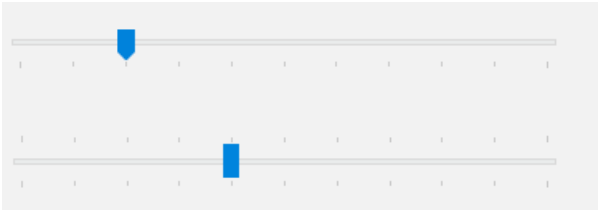
Ergebnis





3.2.15 TrackBar

Neben den Scrollbars bietet sich auch die *TrackBar* für das schnelle Einstellen von Werten an. Hier zwei Varianten:



Die Breite kann bei *AutoSize = False* verändert werden. Ob die *TrackBar* horizontal oder vertikal erscheinen soll, können Sie mittels der *Orientation*-Eigenschaft festlegen. Die Anzahl der Teilstriche ergibt sich aus der Division der *Maximum*-Eigenschaft durch den Wert von *TickFrequency*. Bei der Standardeinstellung *Maximum = 10* und *TickFrequency = 1* ergeben sich also zehn Teilstriche.

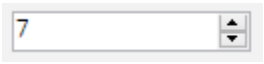
Mit der *TickStyle*-Eigenschaft bestimmen Sie, auf welcher Seite die Teilstriche angezeigt werden sollen. Hier die Konstanten der *TickStyle*-Enumeration:

TickStyle	Erklärung
None	Keine Teilstriche
TopLeft	Teilstriche oben (bei horizontaler Ausrichtung) bzw. links (bei vertikaler Ausrichtung)
BottomRight	Teilstriche unten (bei horizontaler Ausrichtung) bzw. rechts (bei vertikaler Ausrichtung, Standardeinstellung)
Both	Teilstriche auf beiden Seiten

Da die übrigen wichtigen Eigenschaften (*Maximum*, *Minimum*, *Value*, *SmalChange*, *LargeChange*) sowie Ereignisse (*Scroll*, *ValueChanged*) weitgehend mit denen der Scrollbar übereinstimmen, möchten wir an dieser Stelle auf die vorhergehenden Abschnitte verweisen.

3.2.16 NumericUpDown

Wer hatte nicht schon die Aufgabe, einen Integer- oder Währungswert abzufragen, und war an den diversen Fehlereingabemöglichkeiten gescheitert? *NumericUpDown* verspricht Abhilfe. Das Steuerelement entspricht der Kombination einer einzeiligen Textbox mit einer vertikalen Scrollbar.



Angefangen vom Definieren eines zulässigen Bereichs (*Maximum*, *Minimum*) über die Anzahl der Nachkommastellen (*DecimalPlaces*) bis hin zum Inkrement (*Increment*) bzw. zur Anzeige von Tausender-Trennzeichen (*ThousandsSeparator*) ist alles vorhanden, was das Programmiererherz begehrt. Last but not least soll die *Value*-Eigenschaft nicht vergessen werden, hier fragen Sie den eingegebenen Wert ab.

HINWEIS: Im Unterschied zu den Steuerelementen *ScrollBar* und *TrackBar* sind die gleichnamigen Eigenschaften von *NumericUpDown* vom Datentyp *decimal* anstatt *int*.

Von besonderem Interesse sind auch hier die Ereignisse *Scroll* (beim Klicken auf eine der beiden Schaltflächen) und *ValueChanged* (beim Ändern des Werts).

Beispiel 3.19: Meldung beim Überschreiten eines Limits

C#

```
private void NumericUpDown1_ValueChanged(object sender, EventArgs e)
{
    if (numericUpDown1.Value > 1000)
    {
        MessageBox.Show("Wertebereichsüberschreitung!");
    }
}
```

3.2.17 DomainUpDown

Für die Auswahl von Werten aus einer vorgegebenen Liste können Sie anstatt einer *ListBox* oder einer *ComboBox* auch eine *DomainUpDown*-Komponente verwenden. Dieses Control kombiniert die Features eines *NumericUpDown*-Steuerelements mit denen einer *ComboBox* (lässt sich allerdings nicht aufklappen). Rein äußerlich scheint es zunächst keinen Unter-

schied zu *NumericUpDown* zu geben, die Funktionalität ist jedoch grundsätzlich verschieden, bestehen doch die Einträge nicht aus Zahlen, sondern aus Strings (Reihenfolge kann mittels *Sort*-Eigenschaft alphabetisch sortiert werden).

Die zulässigen Auswahlwerte übergeben Sie der Komponente in der Collection *Items*, den gewählten Wert ermitteln Sie mit der *Text*-Eigenschaft oder Sie können auch den Listenindex mit *SelectedIndex* abrufen.

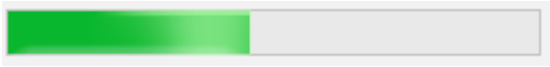
Beispiel 3.20: Der erste Listeneintrag wird angezeigt.

C#

```
domainUpDown1.SelectedIndex = 0;
```

3.2.18 ProgressBar

Ist Ihr Computer bei manchen Aufgaben nicht schnell genug, ist es sinnvoll, dem Nutzer ein Lebenszeichen zum Beispiel in Gestalt eines Fortschrittbalkens zu geben. Genau diese Aufgabe übernimmt die *ProgressBar*:



Die Eigenschaften *Minimum*, *Maximum* und *Value* sind vergleichbar mit denen der *ScrollBar*. Sie können *Value* einen Wert zuweisen, andererseits aber auch die Methode *PerformStep()* aufrufen.

Beispiel 3.21: Bei jedem Klick auf den *Button* bewegt sich der Balken weiter und hat nach zehnmaligem Klicken das Maximum erreicht.

C#

```
private void Button1_Click(object sender, EventArgs e)
{
    progressBar1.Maximum = 100;
    progressBar1.Step = 10;
    progressBar1.PerformStep();
}
```

HINWEIS: Mit der *Style*-Eigenschaft (*Blocks*, *Continuous*, *Marquee*) können Sie das Aussehen des Balkens verändern.

3.2.19 RichTextBox

Wollten Sie nicht schon immer einmal Ihre Anwendungen mit einem kleinen Texteditor vervollständigen, der auch verschiedene Schriftarten, Schrift- und Absatzformate sowie Grafiken zulässt? Dann sollten Sie sich unbedingt das *RichTextBox*-Control ansehen.

HINWEIS: Mit RTF ist das „Rich Text"-Format gemeint, das als Austauschformat zwischen verschiedenen Textverarbeitungsprogrammen fungiert.

Viele Eigenschaften entsprechen denen der *TextBox*, so ist der gesamte Inhalt in der *Text*-Eigenschaft enthalten. Mit diversen *Selection...*-Eigenschaften lassen sich die markierten Textabschnitte formatieren.

Beispiel 3.22: Selektierten Text in Fett- und Kursivschrift formatieren

C#

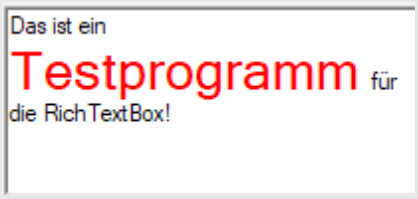
```
richTextBox1.SelectionFont = new Font(richTextBox1.SelectionFont,
             richTextBox1.SelectionFont.Style ^ FontStyle.Bold);
richTextBox1.SelectionFont = new Font(richTextBox1.SelectionFont,
             richTextBox1.SelectionFont.Style ^ FontStyle.Italic);
```

Beispiel 3.23: Schriftgröße und -farbe ändern, Kursivschrift und markierte Zeile zentrieren

C#

```
richTextBox1.SelectionFont = new Font(richTextBox1.SelectionFont.Name, 20f);
richTextBox1.SelectionColor = Color.Red;
richTextBox1.SelectionAlignment = HorizontalAlignment.Center;
```

Ergebnis



Zum Laden bzw. Abspeichern von RTF-Dateien stehen die Methoden *LoadFile* und *SaveFile* zur Verfügung.

Beispiel 3.24: Laden der Datei *test.rtf* aus dem Anwendungsverzeichnis

C#

```
private void Button1_Click(object sender, EventArgs e)
{
    try
    {
        richTextBox1.LoadFile(Application.StartupPath + "\\test.rtf");
```

```
    }
    catch (System.IO.IOException ex)
    {
         MessageBox.Show(ex.Message);
    }
}
```

Bei der *RichTextBox* handelt es sich nur auf den ersten Blick um einen kompletten Texteditor, denn Sie haben sicherlich bereits festgestellt, dass Sie ja auch noch ein Menü bzw. eine Toolbar benötigen.

3.2.20 ListView

Die *ListView* ist eine ziemlich komplexe Komponente, sie ermöglicht Ihnen die Anzeige einer Liste mit Einträgen, die optional auch mit einem Icon zwecks Identifikation des Typs ausgestattet werden können, wozu zusätzlich eine oder mehrere *ImageList*-Komponenten erforderlich sind. Außerdem kann zu jedem Eintrag eine kleine Checkbox hinzugefügt werden, wodurch sich eine bequeme und übersichtliche Auswahlmöglichkeit ergibt.

Die Einsatzgebiete einer *ListView* sind äußerst vielgestaltig, z. B. Darstellen von Datenbankinhalten oder Textdateien. Außerdem kann die Komponente auch Nutzereingaben entgegennehmen, z. B. Dateiauswahl. Als Windows-Nutzer haben Sie garantiert schon mit diesem Control gearbeitet. Jeder einzelne Ordner auf dem Desktop ist im Grunde ein *ListView*-Objekt und fungiert als eine Art Container für eine Anzahl von einzelnen Items, die über Grafik und Text verfügen können.

ListViewItem

Das *ListViewItem*-Objekt repräsentiert einen einzelnen Eintrag (Item) in der *ListView*. Jedes *Item* kann mehrere *Subitems* haben, die zusätzliche Informationen bereitstellen.

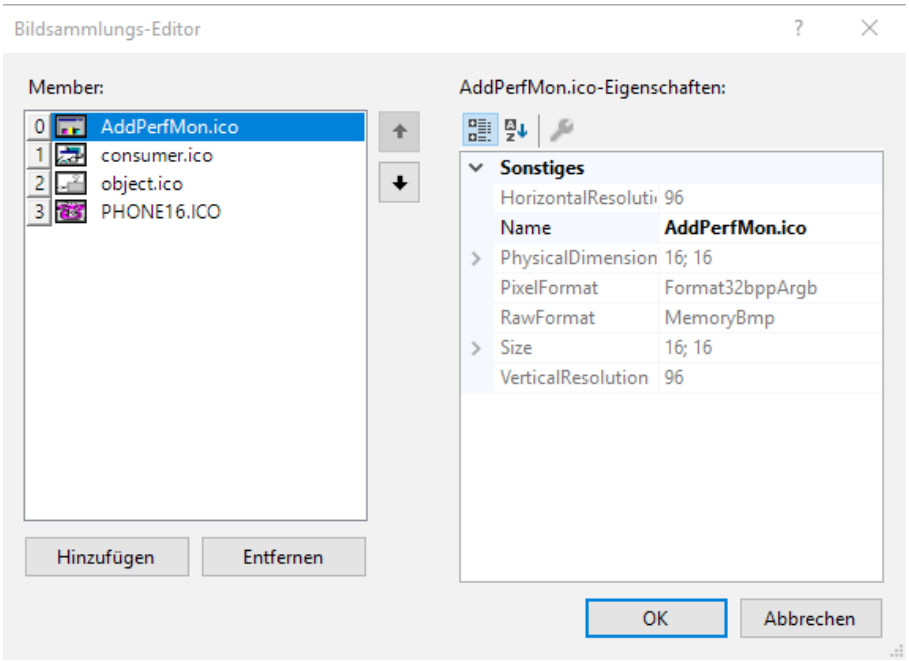
Die Anzeige der Items ist auf vier verschiedene Arten möglich:

- mit großen Icons,
- mit kleinen Icons,
- mit kleinen Icons in einer vertikalen Liste,
- Gitterdarstellung mit Spalten für Untereinträge (Detailansicht).

Die *ListView*-Komponente erlaubt einfache oder mehrfache Selektion, Letztere funktioniert ähnlich wie bei einer *ListBox*-Komponente.

ImageList

Meist wird die *ListView* zusammen mit einer (oder auch mehreren) *ImageList*-Komponenten eingesetzt, welche die Bilddaten speichern. Die Zuweisung erfolgt über die Eigenschaften *LargeImageList* bzw. *SmallImageList*. Jede *ImageList*-Komponente hat eine *Images*-Auflistung, die Sie über das Eigenschaften-Fenster (*F4*) erreichen und die Sie mithilfe des *Image-Auflistungs-Editor*s mit diversen Icons füllen können.



Übersichten zur ListView

Eigenschaft	Bedeutung
View	… ermöglicht die Einstellung des Anzeigemodus (siehe oben): *List, SmallIcon, LargeIcon, Details*
LargeImageList *SmallImageList* *StateImageList*	Auswahl der *ImageList*-Objekte, welche die Bilddateien für große (32 x 32), kleine (16 x 16) Icons bzw. für die Darstellung der Checkbox bereitstellen (nur für *CheckBoxes* = *True*)
CheckBoxes	… Ein- bzw. Ausblenden der Checkbox (*True/False*)
CheckItems	… Zugriff auf die *CheckedListViewItemCollection*, um die aktivierten Einträge festzustellen
Columns	… Zugriff auf die *ColumnHeaderCollection* (nur für *View* = *View.Details*)
Items	… Hinzufügen/Entfernen von Einträgen durch Zugriff auf die *ListView-ItemCollection*, die entsprechende Methoden bereitstellt
ImageIndex	… der Bildindex
LabelEdit	… erlaubt/verbietet das Editieren von Einträgen (*True*)
Sorting	… alphabetisches Sortieren der Einträge (*None*, *Ascending*, *Descending*)
AllowColumnReorder	nur für Detailansicht (*View* = *View.Details*): … erlaubt Vertauschen der Spalten zur Laufzeit durch Anfassen mit der Maus (*True*)

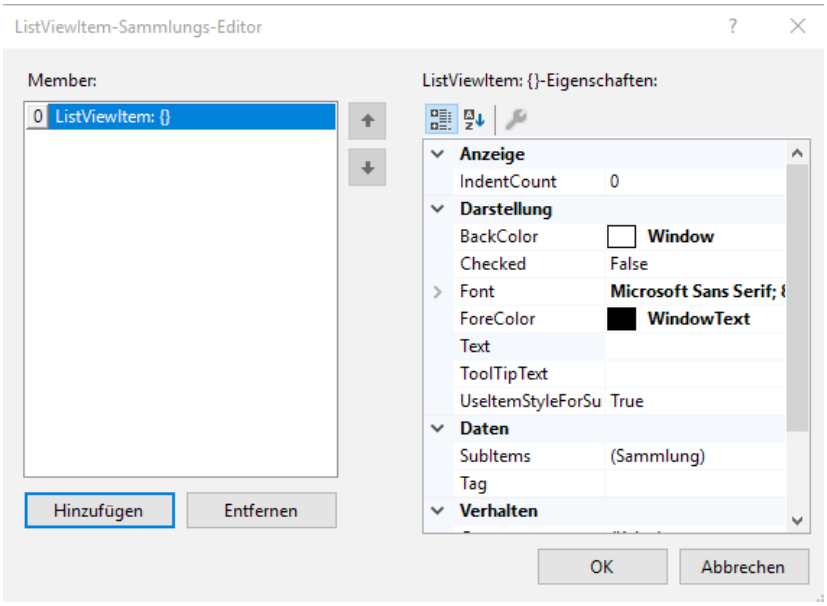
FullRowSelect	… komplette Zeile wird selektiert (*True*)
GridLines	… Anzeige von Gitterlinien (*True*)
HeaderStyle	… versteckt Spaltentitel (*None*), zeigt ihn an (*NoneClickable*) bzw. Spaltentitel funktioniert wie ein Button (*Clickable*)
HideSelection	… selektierte Einträge bleiben markiert, wenn die *ListView* den Fokus verliert (*False*)

Methode	Beschreibung
BeginUpdate *EndUpdate*	… erlaubt das Hinzufügen von mehreren Items, ohne dass *ListView* jedes Mal neu gezeichnet wird
Clone	… überträgt Items in andere *ListView*
GetItemAt	… ermöglicht das Bestimmen des *Items* durch Klick mit der Maus auf ein *SubItem* (*View = View.Details*)
EnsureVisible	… ermöglicht die Anzeige eines gewünschten Eintrags nach Neuaufbau

Ereignis	… wird ausgelöst
BeforeLabelEdit *AfterLabelEdit*	… bevor bzw. nachdem ein Eintrag editiert wurde (nur bei *LabelEdit = True*), kann zur Gültigkeitsüberprüfung benutzt werden
ItemActivate	… wenn ein bestimmtes Item ausgewählt wurde, kann zum Auslösen von Aktionen benutzt werden
ColumnClick	… wenn in einen Spaltentitel geklickt wurde, kann z. B. zum Sortieren verwendet werden
ItemCheck	… wenn auf eine *CheckBox* geklickt wurde (nur für *CheckBoxes = True*)

Die Vorgehensweise scheint recht einfach zu sein:

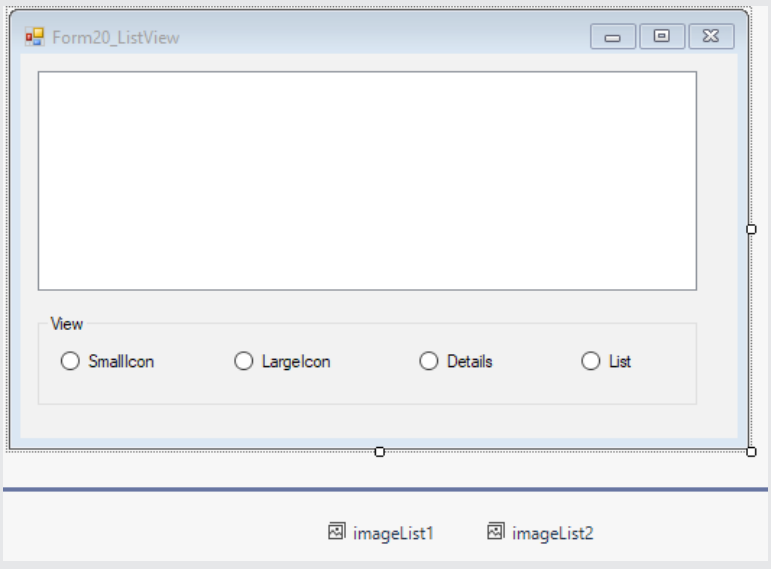
- Verknüpfen Sie die *ListView* mit einer *ImageList* mit großen Icons (z. B. 32 × 32) per *LargeImageList*-Eigenschaft und mit einer weiteren *ImageList* die kleinen Icons (16 × 16) per *SmallImageList*-Eigenschaft.
- Neue Einträge erstellen Sie zur Entwurfszeit über die *Items*-Eigenschaft, die auch einen eigenen Editor bereitstellt. Wie Sie der folgenden Abbildung entnehmen können, verfügt jeder Eintrag wiederum über eine Auflistung *SubItems*, die den einzelnen Spalten in der Detailansicht entsprechen.
- Wichtig sind vor allem die Eigenschaften *ImageIndex* und *Text*, da beide das Aussehen des *Items* beeinflussen.
- Möchten Sie in der Detailansicht weitere Daten (Spalten) anzeigen, können Sie diese mit der *Columns*-Eigenschaft hinzufügen. Eingetragen werden später die *Text*-Eigenschaften der *SubItems*.
- Die Art der Anzeige legen Sie mit der *View*-Eigenschaft der *ListView* fest (Liste, Details etc.).



Beispiel 3.25: Stundenplan

C#

Es soll ein einfacher „Stundenplan“ erstellt werden. Die folgende Abbildung gibt einen Überblick über die Anordnung der Komponenten. Wichtig sind die *ListView*-Komponente (oben) und die beiden *ImageList*-Komponenten, die automatisch im Komponentenfach abgelegt werden.



Füllen Sie mit dem Image-Auflistungs-Editor (siehe oben) die erste *ImageList* mit „großen“ und die zweite *ImageList* mit „kleinen“ Icons. Weisen Sie der *LargeImageList*- und der *SmallImageList*-Eigenschaft der *ListView* die beiden *ImageList*-Komponenten zu.

Beim Laden von *Form1* wird dem *Click*-Ereignis der vier *RadioButton*s ein gemeinsamer Event-Handler zugewiesen:

```
private void Form1_Load(object sender, EventArgs e)
{
    radioButton1.Click += new EventHandler(CommonClick);
    radioButton2.Click += new EventHandler(CommonClick);
    radioButton3.Click += new EventHandler(CommonClick);
    radioButton4.Click += new EventHandler(CommonClick);
}
```

Der gemeinsame Event-Handler:

```
private void CommonClick(object sender, System.EventArgs e)
{
    ShowListView();
}
```

Die Anzeige der *ListView*:

```
private void ShowListView()
{
    listView1.Items.Clear();
    if (radioButton1.Checked)
    {
        listView1.View = View.SmallIcon;
    }
    if (radioButton2.Checked)
    {
        listView1.View = View.LargeIcon;
    }
    if (radioButton3.Checked)
    {
        listView1.View = View.Details;
    }
    if (radioButton4.Checked)
    {
        listView1.View = View.List;
    }
```

Spalten für *Items* und *SubItems* definieren:

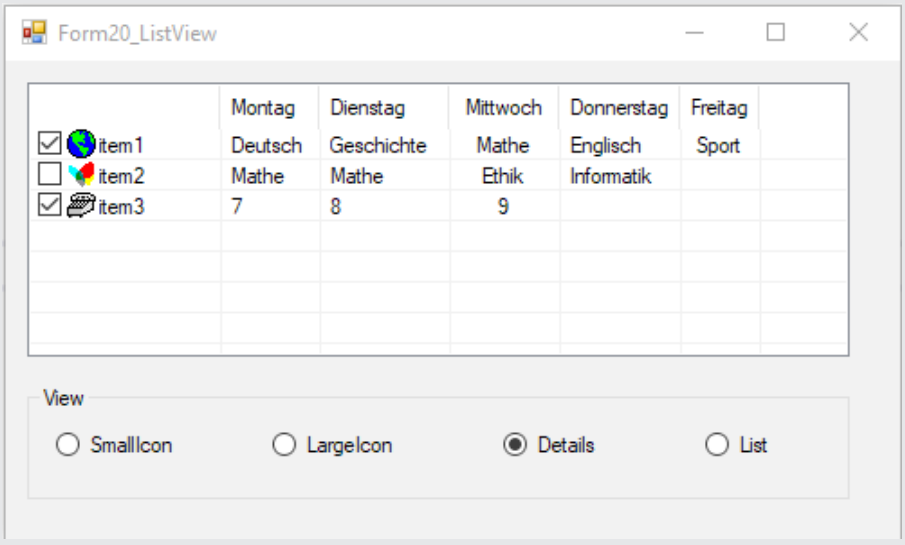
```
    listView1.Columns.Add("", -2, HorizontalAlignment.Left);
    listView1.Columns.Add("Montag", -2, HorizontalAlignment.Left);
    listView1.Columns.Add("Dienstag", -2, HorizontalAlignment.Left);
    listView1.Columns.Add("Mittwoch", -2, HorizontalAlignment.Center);
    listView1.Columns.Add("Donnerstag", -2, HorizontalAlignment.Left);
    listView1.Columns.Add("Freitag", -2, HorizontalAlignment.Center);
    listView1.LabelEdit = true;           // Editieren erlauben:
    listView1.AllowColumnReorder = true;  // Ändern der Spaltenanordnung erlauben
    listView1.CheckBoxes = true;
    listView1.FullRowSelect = true;
    listView1.GridLines = true;
    listView1.Sorting = SortOrder.Ascending;
```

Schließlich drei *Items* mit Gruppen von *SubItems* erzeugen und zur *ListView* hinzufügen:

```
    ListViewItem item1 = new ListViewItem("item1", 0);
    item1.Checked = true;
    item1.SubItems.Add("Deutsch");
    item1.SubItems.Add("Geschichte");
    item1.SubItems.Add("Mathe");
    item1.SubItems.Add("Englisch");
    item1.SubItems.Add("Sport");
    listView1.Items.Add(item1);

    ListViewItem item2 = new ListViewItem("item2", 1);
    item1.Checked = true;
    item2.SubItems.Add("Mathe");
    item2.SubItems.Add("Mathe");
    item2.SubItems.Add("Ethik");
    item2.SubItems.Add("Informatik");
    item2.SubItems.Add("");
    listView1.Items.Add(item2);

    ListViewItem item3 = new ListViewItem("item3", 2);
    item3.Checked = true;
    item3.SubItems.Add("7");
    item3.SubItems.Add("8");
    item3.SubItems.Add("9");
    listView1.Items.Add(item3);
  }
```

Ergebnis

Starten Sie das Programm und experimentieren Sie nun mit den verschiedenen Ansichten. Die Abbildung zeigt die Detailansicht:



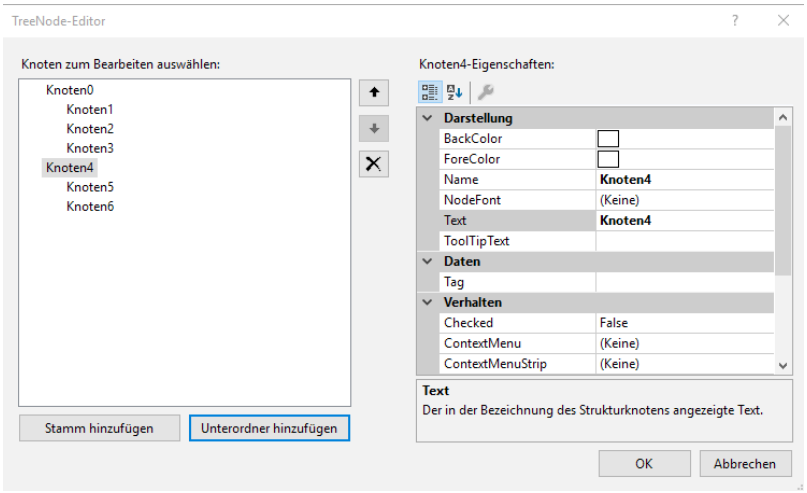
3.2.21 TreeView

Wollen Sie den Anwendern Ihrer Programme mehr bieten als nur langweilige Listendarstellungen und primitive Eingabemasken? Oder müssen Sie hierarchische Abhängigkeiten grafisch darstellen?

Falls ja, dann kommen Sie wohl kaum um die Verwendung der *TreeView*-Komponente herum. Ganz abgesehen davon, dass hier eine Baumdarstellung wesentlich übersichtlicher ist als eine Tabelle, können Sie durch den Einsatz grafischer Elemente auch noch reichlich Eindruck schinden.

Wie auch die *ListView*-Komponente dient die *TreeView* „lediglich“ als Container für eine Liste von *TreeNodes* (Knoten), die jedoch im Gegensatz zur *ListView* hierarchisch angeordnet sind. Das heißt, ausgehend von einem Knoten werden weitere Unterknoten angeordnet usw.

Zur Entwurfszeit können Sie diese Knoten mit einem eigenen Editor (Eigenschaft *Nodes*) hinzufügen (ähnlich dem Erzeugen von Verzeichnissen auf einer Festplatte):



Knoten zur Laufzeit erzeugen

Zur Laufzeit gestaltet sich dieses Vorgehen schon etwas aufwendiger und erfordert vom Programmierer einiges Vorstellungsvermögen, gilt es doch, sich mithilfe von Collections und Methoden buchstäblich durch den Baum zu „hangeln“.

Beispiel 3.26: Dynamisches Füllen einer *TreeView*

C#

Zunächst den Baum löschen und die automatische Aktualisierung ausschalten (bessere Performance):

```
private void Button1_Click(object sender, EventArgs e)
{
    treeView1.BeginUpdate;
    treeView1.Nodes.Clear;
```

Wir fügen den ersten (Root-)Knoten bzw. das erste Stammelement ein:

```
treeView1.Nodes.Add("Deutschland");
```

Da der erste Knoten den Index 0 in der *Nodes*-Collection erhält, können wir direkt über den Index auf diesen Knoten zugreifen und weitere untergeordnete Knoten erzeugen, die zur *Nodes*-Auflistung des Knotens hinzugefügt werden:

```
treeView1.Nodes[0].Nodes.Add("Bayern");
treeView1.Nodes[0].Nodes.Add("Hessen");
treeView1.Nodes[0].Nodes.Add("Baden-Württemberg");
```

Das gleiche Prinzip auf die nächstfolgende Knotenebene angewendet:

```
treeView1.Nodes[0].Nodes[0].Nodes.Add("München");
treeView1.Nodes[0].Nodes[0].Nodes.Add("Regensburg");
treeView1.Nodes[0].Nodes[0].Nodes.Add("Deggendorf");
```

Doch eigentlich wird jetzt die Schreiberei etwas zu aufwendig. Bequemer ist die folgende Variante, bei der ein *TreeNode*-Objekt von der *Add*-Methode zurückgegeben wird:

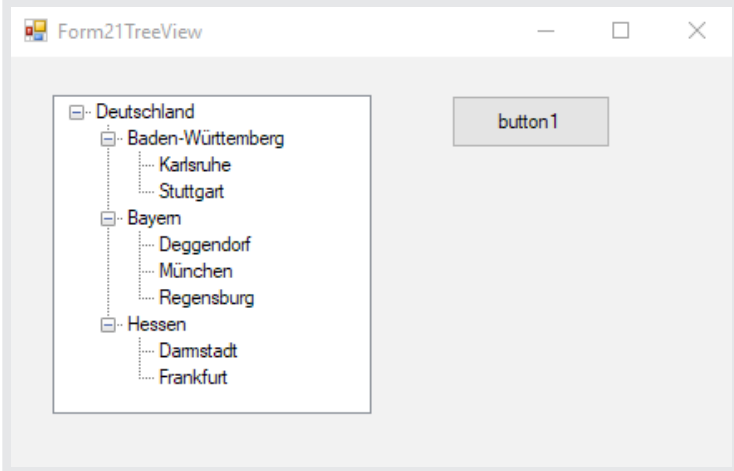
```
TreeNode n = treeView1.Nodes[0].Nodes[1];
n.Nodes.Add("Frankfurt");
n.Nodes.Add("Darmstadt");
n = treeView1.Nodes[0].Nodes[2];
n.Nodes.Add("Stuttgart");
n.Nodes.Add("Karlsruhe");
```

Es genügt also, dass wir uns auf einen bestimmten Knoten beziehen, um weitere Unterknoten zu erzeugen.

Zum Schluss noch Sortieren und die Aktualisierung einschalten, sonst ist nichts zu sehen:

```
    treeView1.Sort();
    treeView1.EndUpdate();
}
```

Ergebnis

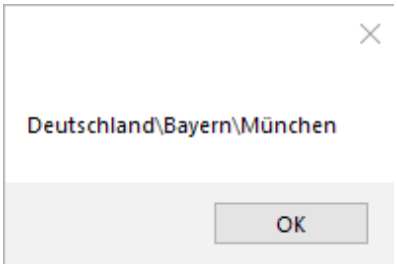


Auswerten des aktiven Knotens

Über das *AfterSelect*-Ereignis können Sie auswerten, welcher Knoten gerade aktiviert wurde. Der Knoten selbst wird als *e.Node*-Objekt als Parameter übergeben:

```
private void TreeView1_AfterSelect(object sender, TreeViewEventArgs e)
{
    MessageBox.Show(e.Node.Text);
    MessageBox.Show(e.Node.FullPath);
}
```

Während *Node.Text* lediglich die Beschriftung des aktuellen Knotens zurückgibt, enthält *Node.FullPath* den kompletten Pfad bis zur Root:



Wichtige Eigenschaften von TreeView

Die *TreeView* ist ein hochkomplexes Steuerelement und verfügt über eine Flut von Eigenschaften, Methoden und Ereignissen, bei denen man schnell die Übersicht verlieren kann. Gerade deshalb ist es sinnvoll, wenn man wenigstens die wichtigsten Klassenmitglieder kennt. Dabei müssen wir natürlich auch die untergeordnete Klasse *TreeNode*, die einen einzelnen Knoten innerhalb von *TreeView* repräsentiert, mit einbeziehen.

Eigenschaft	Beschreibung
CheckBoxes	… legt fest, ob CheckBoxen angezeigt werden (*True/False*)
FullRowSelect	… legt fest, ob die gesamte Breite farblich hervorgehoben wird (*True/False*)
HideSelection	… entfernt farbliche Hervorhebung des selektierten Knotens (*True/False*)
HotTracking	… legt fest, ob Knotenbezeichnung als Hyperlink erscheint (*True/False*)
ImageIndex	Index des Bildchens, das erscheint, wenn Knoten nicht selektiert ist
ImageList	… verweist auf die zugeordnete *ImageList*
Indent	… Einzugsbreite der untergeordneten Knoten (Pixel)
LabelEdit	… legt fest, ob Anwender Knotenbeschriftung editieren kann (*True/False*)
Nodes	… Auflistung mit allen *TreeNode*-Objekten der untergeordneten Ebene
SelectedImage-Index	… Index des Bildchens, das bei selektiertem Knoten angezeigt wird
SelectedNode	… liefert selektiertes *TreeNode*-Objekt
ShowLines	… gibt an, ob Linien zwischen den Knoten gezeichnet werden sollen (*True/False*)

Wichtige Methoden von TreeView

Methode	Beschreibung
BeginUpdate()	… deaktiviert das Neuzeichnen des Controls
CollapseAll()	… reduziert alle Knoten
EndUpdate()	… aktiviert das Neuzeichnen des Controls
ExpandAll()	… expandiert alle Knoten
Sort()	… sortiert alle Knoten

Wichtige Ereignisse von TreeView

Beim Aufklappen und Schließen der Knoten löst *TreeView* Ereignispärchen aus.

Ereignisse	Ereignis wird ausgelöst,
BeforeExpand *AfterExpand*	… bevor/nachdem Knoten geöffnet worden ist.
BeforeSelect *AfterSelect*	… bevor/nachdem Knoten ausgewählt wurde.
BeforeCollaps *AfterCollaps*	… bevor/nachdem Knoten geschlossen wurde.
NodeMouse-Click	… wenn Benutzer auf einen Knoten klickt.

Wichtige Eigenschaften von TreeNode

Eigenschaft	Beschreibung
FirstNode	Das erste untergeordnete *TreeNode*-Objekt in der *Nodes*-Auflistung des aktuellen Knotens
Index	Position des aktuellen Knotens in der *Nodes*-Auflistung des übergeordneten Knotens
IsExpanded	*True*, falls *TreeNode*-Objekt expandiert ist
IsSelected	*True*, falls *TreeNode*-Objekt selektiert ist
LastNode	Das letzte untergeordnete *TreeNode*-Objekt in der *Nodes*-Auflistung des aktuellen Knotens
NextNode	Das nächste gleichrangige *TreeNode*-Objekt
Nodes	Auflistung, die alle *TreeNode*-Objekte der untergeordneten Ebene enthält
Parent	Das übergeordnete *TreeNode*-Objekt
PrevNode	Das vorhergehende gleichrangige *TreeNode*-Objekt

Wichtige Methoden von TreeNode

Methode	Beschreibung
Collapse()	Das *TreeNode*-Objekt wird reduziert.
Expand()	Das *TreeNode*-Objekt wird expandiert.
ExpandAll()	Alle untergeordneten *TreeNode*-Objekte werden expandiert.
Toggle()	*TreeNode* wechselt zwischen reduziertem und erweitertem Zustand.

3.3 Container

Mit diesen Komponenten können Sie ein übersichtliches und anpassungsfähiges Outfit der Bedienoberfläche Ihrer Anwendung erreichen.

3.3.1 FlowLayout/TableLayout/SplitContainer

Diese Komponenten ermöglichen ein flexibles Formularlayout und wurden bereits im Abschnitt 2.2.7 des Vorgängerkapitels näher beschrieben.

3.3.2 Panel

Das auf den ersten Blick etwas unscheinbar wirkende Panel dürfte eines der wichtigsten Gestaltungsmittel für Dialogoberflächen sein. Die wichtigste Fähigkeit dieser Komponente: Sie kann weitere Komponenten in ihrem Clientbereich aufnehmen. Damit verhält sie sich fast wie ein Formular, es gibt eine *Controls*-Auflistung und sogar eine *AutoScroll*-Eigenschaft ist vorhanden!

Wie auch bei der *GroupBox* gilt:

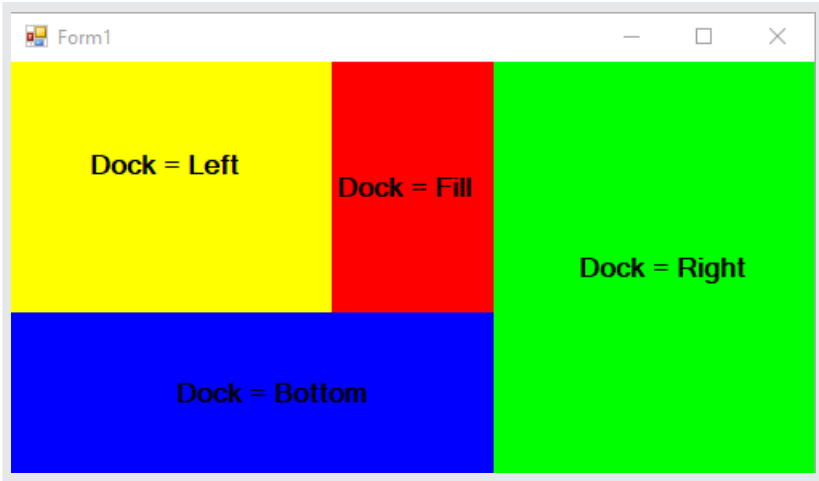
HINWEIS: Beachten Sie beim Entwurf, dass zuerst das *Panel* angelegt werden muss und erst dann Steuerelemente innerhalb desselben abgelegt werden können.

Oberflächen gestalten

Im Zusammenhang mit der *Dock*- bzw. *Anchor*-Eigenschaft bieten sich fast unbegrenzte Möglichkeiten, den Clientbereich des Formulars in einzelne Arbeitsbereiche aufzuteilen und dynamisch auf Größenänderungen zu reagieren.

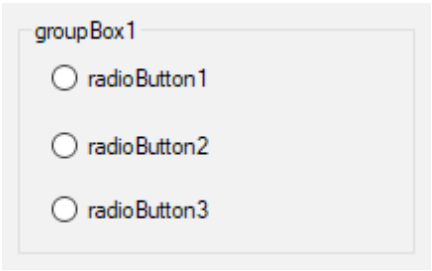
HINWEIS: Nutzen Sie die *SplitContainer*-Komponente, kann der Nutzer zur Laufzeit die Größe der Panels ändern.

Beispiel 3.27: Mehrere mit Dock ausgerichtete *Panels*



3.3.3 GroupBox

Wenn man mehrere Steuerelemente mit einer *GroupBox* umgibt, können diese zu einer Einheit zusammengefasst werden. Insbesondere beim Gruppieren von *RadioButtons* dürfte die *GroupBox* die erste Wahl sein.



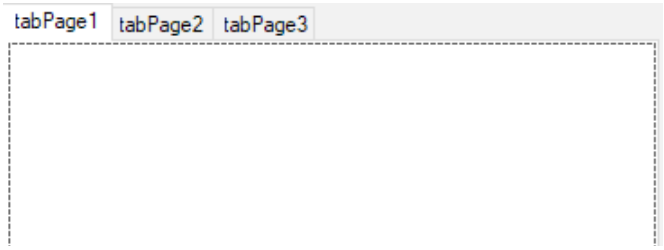
Wie auch das *Panel* verfügt die *GroupBox* über eine eigene *Controls*-Auflistung, zu der weitere Elemente mit *Add* bzw. *AddRange* hinzugefügt werden können.

Eine *GroupBox* reiht sich zwar mit ihrer *TabIndex*-Eigenschaft in die Tabulator-Reihenfolge der übrigen Steuerelemente des Formulars ein. Für die in der *GroupBox* enthaltenen Elemente gibt es jedoch eine eigene Reihenfolge, die mit *TabIndex = 0* beginnt.

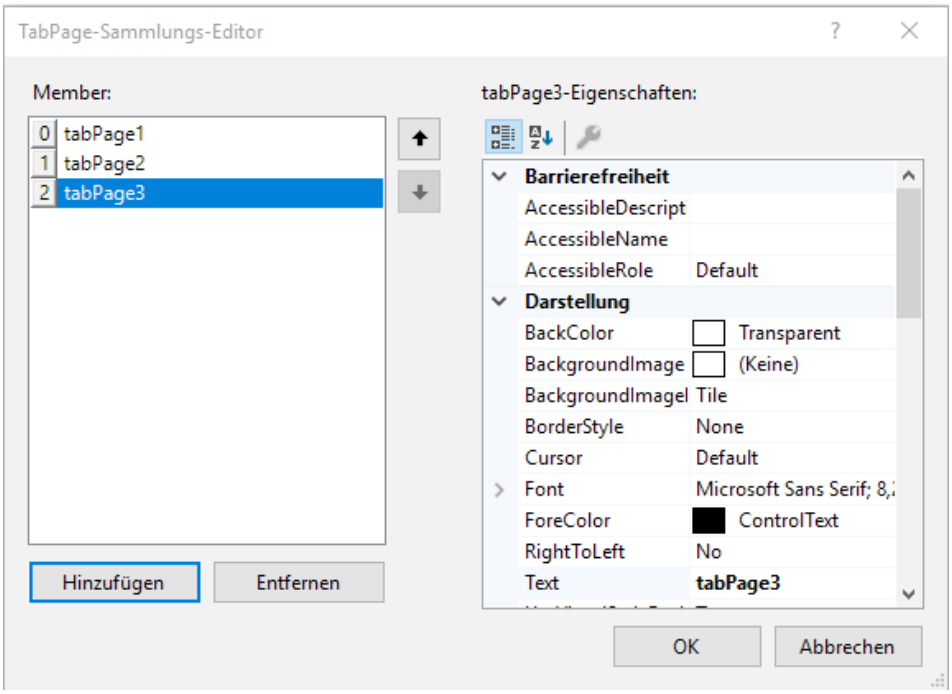
HINWEIS: Beachten Sie beim Entwurf, dass zuerst die *GroupBox* angelegt werden muss und erst dann Steuerelemente innerhalb derselben abgelegt werden können.

3.3.4 TabControl

Diese Komponente bietet die Funktionalität eines *Panels* und die Möglichkeit, über ein Register auf weitere Seiten zuzugreifen:



Für das Zuweisen der Eigenschaften und für das Hinzufügen neuer Registerblätter steht Ihnen der „TabPage-Auflistungs-Editor“ zur Verfügung (erreichbar über die Eigenschaft *TabPages*):



Um per Code neue Registerkarten (*TabPage*-Objekte) hinzuzufügen, verwenden Sie (wie könnte es bei einem Container-Control auch anders sein) die *Add*-Methode der *Controls*-Auflistung des *TabControl*-Objekts.

Beispiel 3.28: Eine Registerkarte mit der Beschriftung „XYZ“ wird zu einem *TabControl* hinzugefügt.

C#

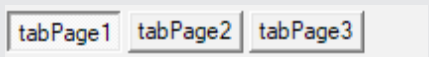
```
private void Form4TabPages_Load(object sender, EventArgs e)
{
    TabPage tabPg = new TabPage("XYZ");
    tabControl1.Controls.Add(tabPg);
}
```

Wichtige Eigenschaften

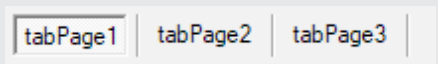
Eigenschaft	Beschreibung
Alignment	… bestimmt Anordnung der Karteireiter
Appearance	… bestimmt Erscheinungsbild des *TabControl*s
ImageList	… die *ImageList* mit den auf den Registerkarten anzuzeigenden Bildern
ItemSize	… bestimmt die Größe der Registerkarten
MultiLine	… legt fest, ob Karteireiter mehrzeilig sein können
Padding	… bestimmt Abstand zwischen Beschriftung und Rand eines Karteireiters
RowCount	… liest Anzahl der Zeilen eines Karteireiters
SelectedIndex	… legt Index der aktuellen Registerkarte fest
SelectedTab	… bestimmt aktuell ausgewählte Registerkarte
TabPages	… liest Auflistung der Registerkarten des *TabControl*s

Beispiel 3.29: *Appearance*

Appearance = Buttons



Appearance = FlatButtons



Wichtige Ereignisse

Wenn sich die aktuelle Registerkarte ändert, haben wir es mit folgender Ereigniskette zu tun:

Deselecting → *Deselected* → *Selecting* → *Selected*

Ereignisse	Beschreibung
Deselecting / Deselected	Übergabe eines Objekts vom Typ *TabControlCancelEventArgs*
Selecting / Selected	Übergabe eines Objekts vom Typ *TabControlEventArgs*

Mit diesen Ereignissen können Sie auf einen Seitenwechsel reagieren (z. B. Sichern der Eingabewerte oder Initialisieren der enthaltenen Controls).

Mit der *Cancel*-Eigenschaft von *TabControlCancelEventArgs* können Sie den eingeleiteten Vorgang abbrechen (siehe auch *FormClosing*-Event).

Außer den oben aufgeführten Ereignissen ist noch das *SelectedIndexChanged*-Ereignis erwähnenswert, das bei Ändern der *SelectedIndex*-Eigenschaft auftritt.

3.3.5 ImageList

Um es gleich vorwegzunehmen, das *ImageList*-Steuerelement allein ist nur von begrenztem Nutzen. Es fungiert lediglich als (zur Laufzeit unsichtbarer) Container für mehrere (gleich große) Bitmaps, die Sie über ihren Index in der Liste ansprechen und in folgenden Steuerelementen anzeigen lassen können:

- *Label*, *Button*
- *ListView*, *TreeView*
- *ToolBar*
- *TabControl*

Um diese Controls mit der *ImageList* zu verbinden, setzen Sie einfach die jeweils vorhandene *ImageList*-Eigenschaft. Dies kann zur Entwurfszeit erfolgen, Sie dürfen aber auch zur Laufzeit diese Eigenschaft zuweisen.

Bevor Sie die Grafik in die Komponente einlesen, sollten Sie sich für die endgültige Größe (Eigenschaft *ImageSize*) entscheiden, da alle Grafiken auf diese Werte skaliert werden, egal wie groß sie vorher waren.

Über die *Images*-Collection rufen Sie zur Entwurfszeit einen eigenen Editor auf, in den Sie die Grafiken einfügen und, was auch wichtig ist, sie dann sortieren können.

Möchten Sie die Grafiken zur Laufzeit laden, verwenden Sie die *Add*-Methode der *Images*-Collection.

Beispiel 3.30: Hinzufügen von Grafiken zur Laufzeit

C#

```
private void Form5ImageList_Load(object sender, EventArgs e)
{
    imageList1.Images.Add(new Bitmap("RadioButton.bmp"));
```

Beispiel 3.31: Anzeige in einem Button

C#

```
button1.Text = String.Empty;
button1.ImageList = imageList1;
button1.ImageIndex = 0;
```

Ergebnis

HINWEIS: Verwenden Sie die Eigenschaft *ImageAlign*, um die Grafik in der jeweiligen Komponente auszurichten.

3.4 Menüs & Symbolleisten

In diesem Toolbox-Abschnitt finden Sie diverse komplexe Komponenten, die die bequeme Bedienbarkeit einer Anwendung ermöglichen.

3.4.1 MenuStrip und ContextMenuStrip

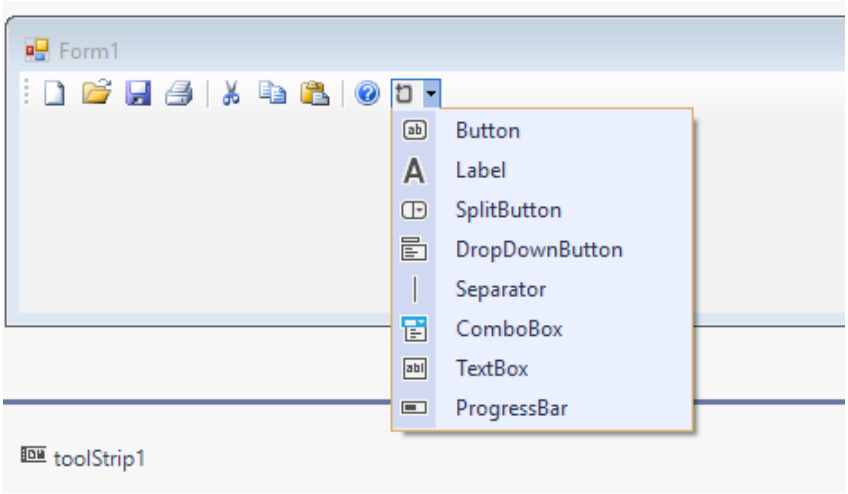
Auf beide Menü-Komponenten, die ab .NET 2.0 die veralteten *MainMenu* und *ContextMenu* abgelöst haben, wurde bereits in Kapitel 2 ausführlicher eingegangen. Im Zusammenhang damit sind u. a. die folgenden Objekte von Bedeutung:

- *ToolStripMenuItem*
- *ToolStripComboBox*
- *ToolStripTextBox*

HINWEIS: Beachten Sie, dass sowohl *ToolStripComboBox* als auch *ToolStripTextBox* keine Untermenüs enthalten können.

3.4.2 ToolStrip

Dieses Steuerelement ist Nachfolger der *ToolBar* und erlaubt einen komfortablen Entwurf von frei konfigurierbaren Werkzeugleisten, die man mit unterschiedlichen Schalt- und Anzeigeelementen (*Button*, *Label*, *SplitButton*, *ProgressBar* …) besetzen kann.



HINWEIS: Die Standardelemente eines *ToolStrip* können Sie ganz einfach anlegen, indem Sie mit der rechten Maustaste in den *Toolstrip* klicken und im Kontextmenü den Eintrag *Standardelemente einfügen* auswählen.

3.4.3 StatusStrip

Hier handelt es sich um den Nachfolger der altbekannten *StatusBar*, wie sie in der Regel an den unteren Rand des Formulars angedockt wird. Genauso wie das *ToolStrip* kann man diese Komponente mit unterschiedlichen Schalt- und Anzeigeelementen (*Button*, *Label*, *SplitButton*, *ProgressBar*, …) bestücken.

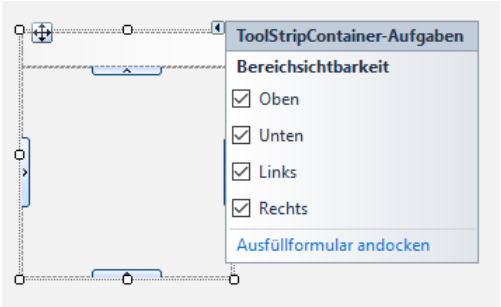
3.4.4 ToolStripContainer

Diese Container-Komponente übernimmt das flexible Positionieren von *MenuStrip* und *StatusStrip*, die zur Entwurfszeit in einen der vier Seitenbereiche (bzw. Panels) des *ToolStripContainer*s gezogen werden. Der Zentralbereich enthält die eigentlichen Steuerelemente des Formulars.

Die Seitenbereiche sind Objekte vom Typ *ToolStripPanel* und können durch Klick auf die entprechende Markierung geöffnet werden.

In der Regel setzen Sie die *Dock*-Eigenschaft des *ToolStripContainer*s auf *Fill*, sodass diese Komponente den gesamten Clientbereich des Formulars einnimmt.

Alternativ können Sie aber auch im SmartTag-Fensterchen den Eintrag „Ausfüllformular andocken“ wählen, um festzulegen, an welchen Seitenrändern Menü- und Symbolleisten zur Laufzeit angedockt werden können.



3.5 Daten

Die in diesem Abschnitt enthaltenen Controls sollen in der Regel das Programmieren von ADO.NET-Anwendungen erleichtern. Eine Beschreibung und Anwendungsbeispiele finden Sie im Buch in Kapitel 18.

3.5.1 DataSet

Mit dieser Komponente erzeugen Sie eine Instanz eines typisierten oder aber eines nicht-typisierten *DataSet*s. Das *DataSet* ist das zentrale Objekt der ADO.NET-Technologie.

3.5.2 DataGridView/DataGrid

Dieses Control ist der Nachfolger des *DataGrid*. Letzteres steht nicht nur aus Kompatibilitätsgründen weiterhin zur Verfügung, es bietet auch den Vorteil, dass mehrere Tabellen und ihre Beziehungen gleichzeitig angezeigt werden können. Das *DataGridView* hingegen verfügt über ein deutlich umfangreicheres Objekt- und Ereignismodell, so können z. B. die Spaltenelemente auch aus anderen Komponenten bestehen, z. B. aus ComboBoxen.

HINWEIS: Das *DataGridView* ist nicht zwingend nur für den Einsatz in Datenbankanwendungen konzipiert, sondern für die tabellenförmige Darstellung nahezu beliebiger Daten geeignet.

3.5.3 BindingNavigator/BindingSource

Diese Steuerelemente sind im Zusammenhang mit der Datenbindung von Steuerelementen von Interesse und werden in Kapitel 6 besprochen.

3.5.4 Chart

Microsoft bietet ebenfalls eine eigene *Chart*-Komponente an, die in fast gleicher Form unter ASP.NET zur Verfügung steht.

So lassen sich mit der Komponente 35 verschiedene Diagrammtypen in 2D/3D-Darstellung anzeigen. Sie als Programmierer haben Einfluss auf Farben, Schatten, Betrachtungspunkte etc., es dürfte für jeden etwas dabei sein.

Beispiel 3.32: Chart mit Daten füllen

C#

```
using System.Windows.Forms.DataVisualization.Charting;
...
        private void button1_Click(object sender, EventArgs e)
        {
```

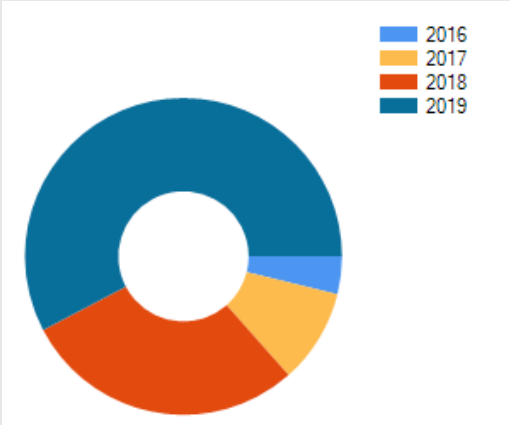
Der Standardreihe eine neue Bezeichnung zuweisen:

```
            chart1.Series[0].Name = "Umsätze Projekt";
```

Vier Datenpärchen übergeben und den ChartType setzen:

```
            chart1.Series[0].Points.AddXY(2016, 10);
            chart1.Series[0].Points.AddXY(2017, 25);
            chart1.Series[0].Points.AddXY(2018, 75);
            chart1.Series[0].Points.AddXY(2019, 150);
            chart1.Series[0].ChartType = SeriesChartType.Doughnut;
        }
```

Ergebnis



Wie Sie sehen, ist mit wenigen Zeilen ein recht ansprechendes Diagramm erstellt, das Sie zum Beispiel mit

```
chart1.Printing.PrintPreview();
```

in einer Druckvorschau anzeigen können oder auch gleich mit

```
chart1.Printing.Print(false);       // ohne Druckerauswahl
```

auf dem Standarddrucker ausgeben können.

Selbstverständlich ist auch eine direkte Datenbindung an diverse Datenquellen möglich, was bereits auf den Datenbankentwickler als Hauptzielgruppe hinweist.

3.6 Komponenten

In diesem Abschnitt sind einige häufig benötigte Controls enthalten, die die Arbeit des Programmierers unterstützen und die nicht direkt auf dem Formular, sondern im Komponentenfach ihren Platz finden.

3.6.1 ErrorProvider

Mit diesem Control können Sie auf einfache Weise eine Eingabevalidierung realisieren, um den Benutzer Ihrer Programme sofort auf ungültige Eingaben hinzuweisen (in diesem Fall erscheint ein Warnsymbol, z. B. neben einer *TextBox*).

C#

```
        private void TextBox1_Validating(object sender, CancelEventArgs e)
        {
```

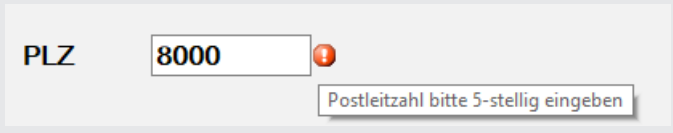
ErrorProvider setzen wenn kein plausibler Wert eingetragen wird:

```
            if (textBox1.Text.Length != 5)
            {
                errorProvider1.SetError(textBox1,
                            "Postleitzahl bitte 5-stellig eingeben");
                e.Cancel = true;
            }
```

Ansonsten ErrorProvider wieder löschen (nicht vergessen, sonst bleibt er auch bei korrekter Eingabe stehen):

```
            else
            {
                errorProvider1.Clear();
            }       }
```

Ergebnis



3.6.2 HelpProvider

Hierbei handelt es sich um eine nicht sichtbare Komponente zur Einbindung der Hilfefunktionalität in ein Formular.

3.6.3 ToolTip

Über dieses Control steuern Sie, wie und vor allem wann Tooltips (die kleinen gelben Hinweisfähnchen) angezeigt werden. Der eigentliche Tooltip wird immer bei den jeweiligen Eigenschaften der Anzeige-Controls verwaltet.

3.6.4 BackgroundWorker

Diese Komponente erlaubt es Ihnen, auf einfache Weise einen Hintergrundthread zu starten.

3.6.5 Timer

Diese Komponente löst in bestimmten Zeitabständen das *Tick*-Ereignis aus. Wesentlich ist die *Interval*-Eigenschaft, sie legt die Zeit (in Millisekunden) fest, der Wert 0 (null) ist nicht zulässig.

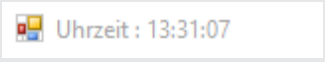
HINWEIS: Im Unterschied zu anderen Komponenten hat die *Enabled*-Eigenschaft standardmäßig den Wert *False*, zum Einschalten des *Timer*s müssen Sie diese Eigenschaft auf *True* setzen!

Beispiel 3.33: Die Uhrzeit wird per Timer im Formularkopf angezeigt (*Interval = 1000*).

C#

```
private void Timer1_Tick(object sender, EventArgs e)
{
    Text = $"Uhrzeit : { DateTime.Now.ToLongTimeString()}";
}
```

Ergebnis



3.6.6 SerialPort

Dieses Control ermöglicht den Zugriff auf die serielle Schnittstelle des Rechners und ermöglicht damit die Steuerung von peripheren Geräten, die über ein solches RS 232-Interface verfügen.

3.7 Drucken

Relativ bescheiden ist das Angebot an Steuerelementen für alle Aktivitäten rund um die Druckausgabe (siehe dazu Kapitel 5), was jedoch nicht bedeutet, dass Sie beim Drucken nicht umfangreiche Unterstützung erfahren.

3.7.1 PrintPreviewControl

Dieses Steuerelement bündelt alle Aktivitäten rund um die Anzeige einer Druckvorschau. An dieser Stelle wollen wir allerdings nicht vorgreifen, sondern Sie gleich an das Kapitel 5 (Druckausgabe) verweisen.

3.7.2 PrintDocument

Mit dieser Komponente sind Sie in der Lage, beliebige Grafiken zu Papier zu bringen. Den kompletten Überblick über die Verwendung finden Sie ebenfalls im Kapitel 5.

3.8 Dialoge

Diese Komponenten kapseln die bekannten Windows-Dialoge.

3.8.1 OpenFileDialog/SaveFileDialog/FolderBrowserDialog

Hier handelt es sich um die Dialoge zum Öffnen und zum Abspeichern von Dateien sowie zum Blättern in Verzeichnissen, mehr dazu im Datei-Kapitel im Buch (Abschnitt 12.5).

3.8.2 FontDialog/ColorDialog

Von Schriftartendialog und Farbdialog haben Sie sicher bereits während der Programmentwicklung Gebrauch gemacht, da die entsprechenden Eigenschaften bzw. Enumerationen bequem damit eingestellt werden können. Wie Sie diese Komponenten richtig einsetzen, zeigt das Grafik-Kapitel 4.

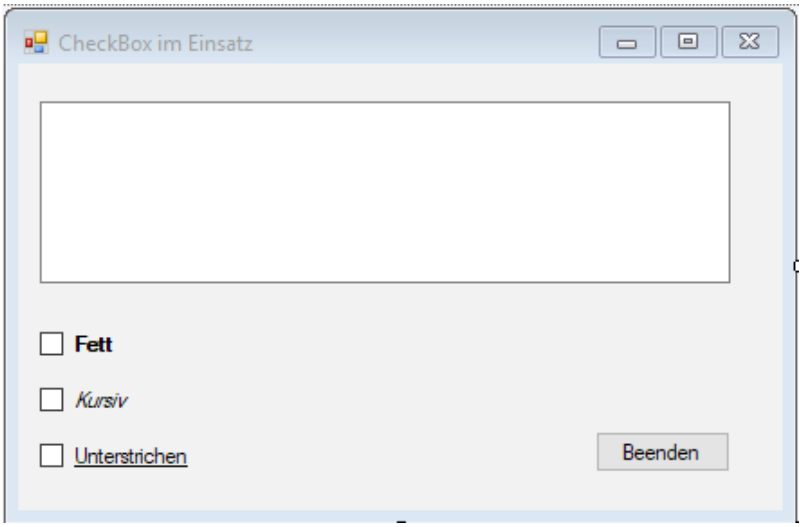
3.9 Praxisbeispiele

3.9.1 Mit der CheckBox arbeiten

Dieses kleine Beispiel demonstriert den Einsatz des *CheckBox*-Steuerelements anhand einer durchaus sinnvollen Aufgabenstellung: Der Schriftstil (fett, kursiv, unterstrichen) des Inhalts einer *TextBox* soll geändert werden. En passant wird auch der Umgang mit dem *Font*-Objekt erklärt.

Oberfläche

Nur eine *TextBox* (*MultiLine=True*), drei *CheckBox*en und ein *Button* sind erforderlich.



Quellcode

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

Alle drei *CheckBox*en teilen sich einen gemeinsamen Eventhandler für das *CheckedChanged*-Ereignis. Geben Sie den Rahmencode komplett selbst ein (also nicht durch die IDE generieren lassen) und weisen Sie erst dann auf der „Ereignisse“-Seite des Eigenschaftenfensters diesen Eventhandler für jede der drei *CheckBox*en einzeln zu.

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

Der resultierende Schriftstil wird durch bitweise ODER-Verknüpfung schrittweise zusammengebaut:

```
        FontStyle ftStyle = FontStyle.Regular;
        if (chkFett.Checked)
        {
            ftStyle |= FontStyle.Bold;
        }
        if (chkKursiv.Checked)
        {
            ftStyle |= FontStyle.Italic;
        }
        if (chkUnderstrichen.Checked)
        {
            ftStyle |= FontStyle.Underline;
        }
```

Ein neues *Font*-Objekt wird auf Basis des geänderten Schriftstils erzeugt und der *TextBox* zugewiesen:

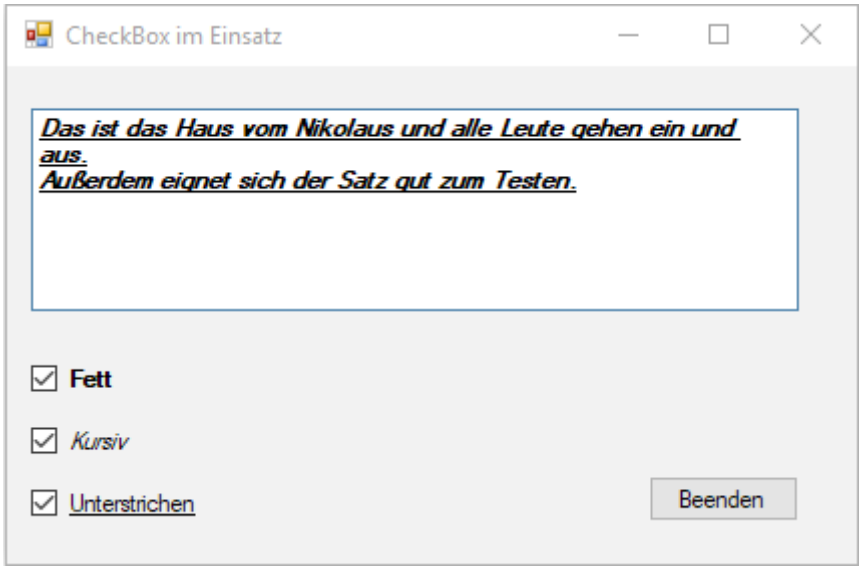
```
        textBox1.Font = new Font(textBox1.Font, ftStyle);
    }
```

Die „Beenden“-Schaltfläche:

```
    private void BtnBeenden_Click(object sender, EventArgs e)
    {
        Close();
    }
}
```

Test

Da alle drei Fontstile miteinander kombiniert werden können, gibt es insgesamt acht Möglichkeiten (eine davon zeigt die folgende Abbildung).



3.9.2 Steuerelemente per Code selbst erzeugen

Der Windows Forms-Designer erlaubt zwar einen bequemen visuellen Entwurf von Benutzerschnittstellen, allerdings kann es sich manchmal durchaus lohnen, wenn man (zumindest teilweise) auf die Dienste des Designers verzichtet und die Steuerelemente „per Hand" programmiert. Dabei gewinnt man nicht nur einen tieferen Einblick in die Prinzipien der objekt- und ereignisorientierten Programmierung, sondern kann auch elegante Oberflächen gestalten, die Aussehen und Funktionalität zur Laufzeit ändern. Ein besonderer Vorteil ergibt sich beim Entwurf von zahlreichen, funktionell gleichartigen, Steuerelementen (Control-Arrays), denn hier bietet Ihnen der Designer kaum Unterstützung.

Die folgende Demo löst das gleiche Problem wie im Vorgängerbeispiel (Zuweisung des Schriftstils für eine *TextBox*), wobei die Steuerelemente per Code erzeugt werden. Die drei *CheckBox*en sind dabei als Steuerelemente-Array organisiert.

Oberfläche

Gönnen Sie Ihrem Windows Forms-Designer eine Pause, das nackte Formular (*Form1*) genügt!

Quellcode

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

Die benötigten globalen Variablen:

```
    private TextBox tb = new TextBox();    // Verweis auf TextBox
    private CheckBox[] checkBoxes = null;  // Verweis auf CheckBox-Array
```

Initialisiertes Array mit den verwendeten Schriftstilen:

```
    private FontStyle[] ftStyles = { FontStyle.Regular, FontStyle.Bold,
                                     FontStyle.Italic, FontStyle.Underline };
```

Im Konstruktor des Formulars werden die benötigten Steuerelemente erzeugt und mit ihren wesentlichen Eigenschaften initialisiert:

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

TextBox erzeugen:

```
        int xpos = 25;
        int ypos = 10;       // linke obere Ecke der TextBox
        tb.Location = new Point(xpos, ypos);
        tb.Multiline = true;
        tb.Width = 200;
        tb.Height = 40;
```

Keinesfalls vergessen darf man das Hinzufügen des Steuerelements zur *Controls*-Auflistung des Formulars (ansonsten bleibt es verborgen):

```
        Controls.Add(tb);
```

Alle *CheckBox*en erzeugen:

```
        int anz = ftStyles.Length-1;
        checkBoxes = new CheckBox[anz];
        for (int i = 0; i < anz; i++)
        {
            checkBoxes[i] = new CheckBox();
            checkBoxes[i].Location = new Point(xpos, ypos + 50 + i * 25);
            checkBoxes[i].Text = ftStyles[i+1].ToString();
```

Gemeinsame Ereignisbehandlung zuweisen:

```
            checkBoxes[i].CheckedChanged += new EventHandler(CBChanged);
        }
```

Alle *CheckBox*en zum Formular hinzufügen

```
        Controls.AddRange(checkBoxes);

        tb.Text = "Dies ist ein Text zum Testen des Programms!";
    }
```

Die Ereignisbehandlung für alle *CheckBox*en:

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

Zunächst normalen Schriftstil einstellen:

```
        FontStyle ftStyle = ftStyles[0];
```

Alle *CheckBox*en durchlaufen:

```
        for (int i = 0; i < anz; i++)
        {
            CheckBox cb = checkBoxes[i];
            if (cb.Checked)
```

Den resultierenden Schriftstil durch bitweises ODER zusammensetzen:

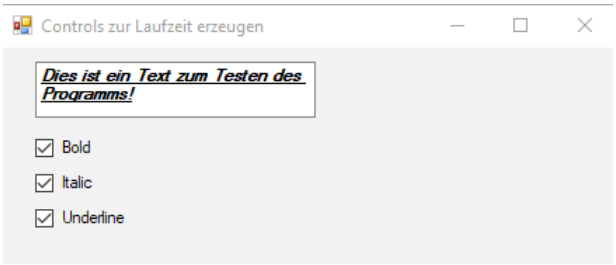
```
                ftStyle |= ftStyles[i + 1];
        }
```

Neues *Font*-Objekt erzeugen und der *TextBox* zuweisen:

```
        tb.Font = new Font(tb.Font, ftStyle);
    }
}
```

Test

Es gibt keine wesentlichen Unterschiede zum Vorgängerbeispiel. Aus Gründen der Einfachheit wurde auf eine deutsche Beschriftung der drei *CheckBox*en und auf eine „Beenden"-Schaltfläche verzichtet:

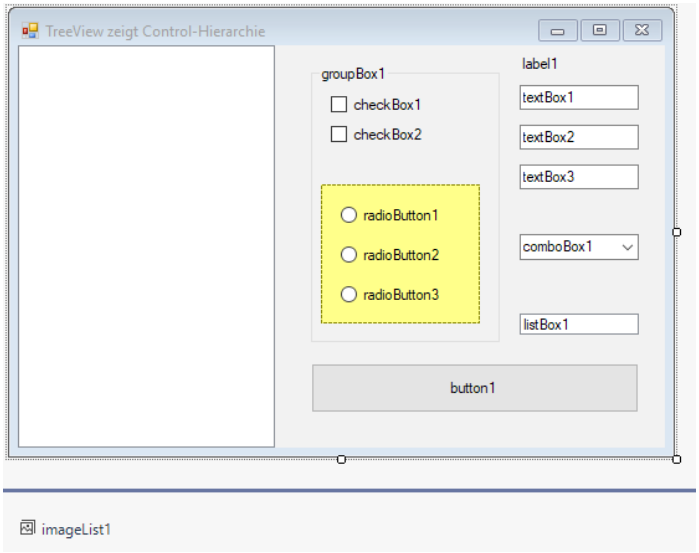


3.9.3 Controls-Auflistung im TreeView anzeigen

Das *TreeView*-Steuerelement gehört mit zu den komplexesten Windows Forms-Controls. Es ist zur Anzeige beliebig verschachtelter hierarchischer Strukturen, wie z. B. Verzeichnisbäume, bestimmt. Auch die *Controls*-Auflistung des Formulars liefert ein hervorragendes Beispiel, um den grundlegenden Umgang mit dieser Komponente zu demonstrieren.

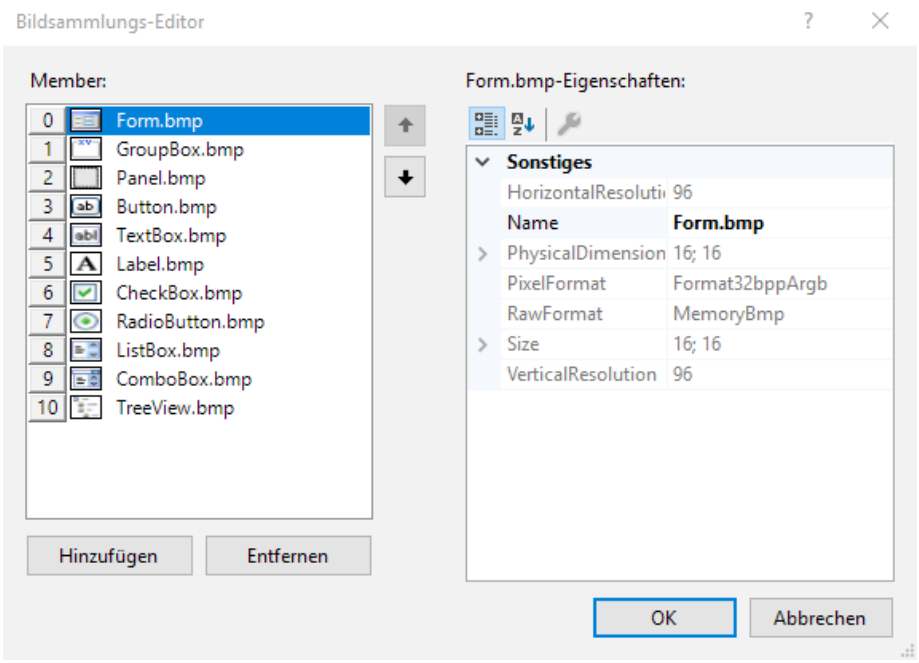
Oberfläche

Eine *TreeView*-Komponente (*Dock* = *Left*) sowie eine vom Prinzip her beliebige Anzahl von Steuerelementen gestalten die Oberfläche von *Form1*. Um eine aussagekräftige Tiefe der Hierarchieebenen zu erhalten, haben wir eine *GroupBox* mit aufgenommen, in welcher u. a. auch ein *Panel* enthalten ist. Beide Komponenten verfügen, genauso wie das Formular, über eigene *Controls*-Auflistungen.



Um die Strukturansicht attraktiv und übersichtlich zu gestalten, sollte jeder Knoten mit einem Icon ausgestattet werden, welches das entsprechende Steuerelement symbolisiert. Mit dem an die *ImageList* angeschlossenen „Image-Auflistungs-Editor“ ist das Hinzufügen der dafür benötigten Bilddateien (16 × 16 Pixel) im Handumdrehen erledigt.

HINWEIS: Vergessen Sie nicht, die *ImageList*-Eigenschaft der *TreeView*-Komponente mit der *ImageList* zu verbinden!



Quellcode

```
using System.Collections;
...
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }
```

Alles passiert im Ladeereignis des Formulars:

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

TreeView mit dem Wurzelknoten initialisieren (der entspricht *Form1*):

```
        treeView1.Nodes.Add(Name);
```

Array zum Speichern der Knoten (pro Control ein Knoten):

```
        TreeNode[] nodesForm = new TreeNode[Controls.Count];
```

Die folgende Anweisung ist dann überflüssig, wenn sich, wie in unserem Fall, das Bildchen für *Form1* an Position 0 in der *ImageList* befindet:

```
        // treeView1.Nodes[0].ImageIndex = 0;
```

Übrige Knoten erzeugen:

```
        SetNodes(Controls,treeView1.Nodes[0]);
    }
```

Alle Controls des Formulars werden durchlaufen, um den *TreeView* aufzubauen. Um tatsächlich alle Steuerelemente zu erfassen, muss sich die *SetNodes*- Methode immer wieder selbst aufrufen:

```
    private void SetNodes(IList controls, TreeNode node)
    {
        foreach (Control ctrl in controls)
        {
            int index = node.Nodes.Add(new TreeNode(ctrl.Name));
            SetImage(node, ctrl, index);   // das richtige Icon zuordnen
            if (ctrl.Controls.Count > 0)
            {
                SetNodes(ctrl.Controls, node.Nodes[index]);
                // rekursiver Aufruf!
            }
        }
    }
```

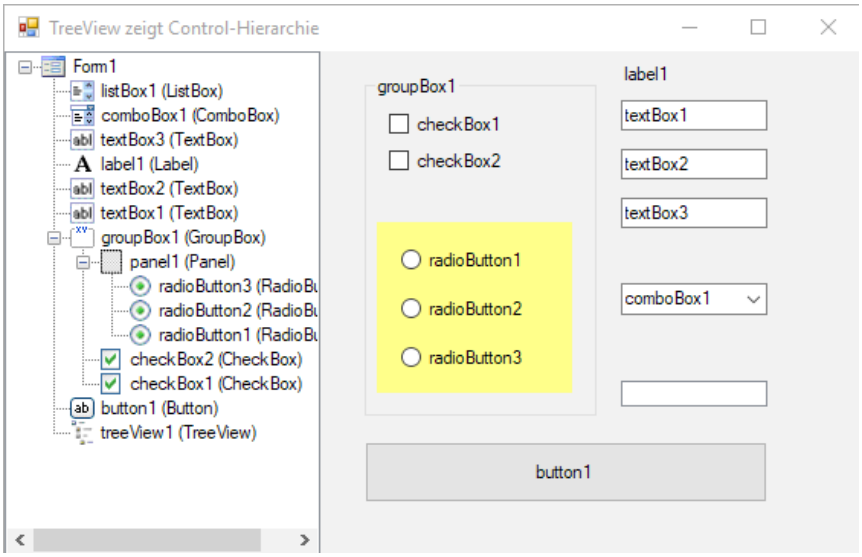
Die folgende Hilfsmethode lädt das zum Control passende Icon aus der *ImageList* und beschriftet den Knoten entsprechend:

```
    private static void SetImage(TreeNode node, Control ctrl, int index)
    {
        if (ctrl is GroupBox)
        {
            node.Nodes[index].ImageIndex = 1;
            node.Nodes[index].Text = ctrl.Name + " (GroupBox)";
        }
        else if (ctrl is Panel)
        {
            node.Nodes[index].ImageIndex = 2;
            node.Nodes[index].Text = ctrl.Name + " (Panel)";
        }
        else if (ctrl is Button)
        {
            node.Nodes[index].ImageIndex = 3;
            node.Nodes[index].Text = ctrl.Name + " (Button)";
        }
        else if (ctrl is TextBox)
        {
            node.Nodes[index].ImageIndex = 4;
            node.Nodes[index].Text = ctrl.Name + " (TextBox)";
        }
        else if (ctrl is Label)
        {
            node.Nodes[index].ImageIndex = 5;
            node.Nodes[index].Text = ctrl.Name + " (Label)";
```

```
            }
            else if (ctrl is CheckBox)
            {
                node.Nodes[index].ImageIndex = 6;
                node.Nodes[index].Text = ctrl.Name + " (CheckBox)";
            }
            else if (ctrl is RadioButton)
            {
                node.Nodes[index].ImageIndex = 7;
                node.Nodes[index].Text = ctrl.Name + " (RadioButton)";
            }
            else if (ctrl is ListBox)
            {
                node.Nodes[index].ImageIndex = 8;
                node.Nodes[index].Text = ctrl.Name + " (ListBox)";
            }
            else if (ctrl is ComboBox)
            {
                node.Nodes[index].ImageIndex = 9;
                node.Nodes[index].Text = ctrl.Name + " (ComboBox)";
            }
            else if (ctrl is TreeView)
            {
                node.Nodes[index].ImageIndex = 10;
                node.Nodes[index].Text = ctrl.Name + " (TreeView)";
            }
        }
    }
```

Test

Unmittelbar nach Programmstart ist zunächst nur der Wurzelknoten (*Form1*) sichtbar. Nach Expandieren dieses Knotens erscheinen die Controls der ersten Ebene. Die zweite und dritte Ebene entstehen nach Expandieren der Knoten für *GroupBox* bzw. *Panel*.



4 Grundlagen Grafikausgabe

Im vorliegenden Kapitel wollen wir uns grundlegend mit dem Erzeugen, Anzeigen und Verarbeiten von Grafiken im Rahmen von Windows Forms-Anwendungen beschäftigen. C# bietet dem Programmierer drei grundsätzliche Varianten an:

- Anzeige von „vorgefertigten" Grafikdateien (z. B. Bitmaps) in der *PictureBox*-Komponente
- Einsatz von Grafikmethoden (z. B. *DrawLine*, *DrawEllipse* …) mit dem *Graphics*-Objekt
- Arbeiten mit GDI-Funktionen

Während bei der ersten Variante die Grafik bereits zur Entwurfszeit (Design Time) entsteht, wird sie bei den beiden Letzteren erst zur Laufzeit (Run Time) erzeugt. Erst hier kann man von eigentlicher „Grafikprogrammierung" sprechen, da diese ausschließlich per Quellcode (also ohne Zuhilfenahme der visuellen Entwicklungsumgebung von C#) funktioniert.

HINWEIS: Auch wenn in diesem Kapitel bereits die Grundlagen für die Druckausgabe gelegt werden, die eigentliche Beschreibung der Vorgehensweise und der nötigen Komponenten finden Sie erst in Kapitel 5.

Wem die Ausführungen dieses Kapitels nicht weit genug gehen oder wer bereits die Grundlagen der Grafikprogrammierung beherrscht, den verweisen wir gleich auf Kapitel 7 weiter. Dort gehen wir gezielt auf einige Spezialthemen rund um die Grafikprogrammierung ein.

4.1 Übersicht und erste Schritte

Nachdem mit dem GDI (*Graphics Design Interface*) die in Win32-Anwendungen übliche Grafikausgabe etwas in die Jahre gekommen war, wurde mit der Einführung des .NET-Frameworks auch eine neue Grafikschnittstelle unter dem Namen GDI+ implementiert.

4.1.1 GDI+ – ein erster Einblick für Umsteiger

GDI+ erfordert schon auf Grund seiner objektorientierten Schnittstelle ein ganz anderes Vorgehen als bei den alten GDI-Anwendungen. Umsteiger von „alten" Programmiersprachen haben es also erfahrungsgemäß etwas schwerer, sich mit den Konzepten von GDI+ anzufreunden. Aus diesem Grund zunächst einige Anmerkungen zu den Grundkonzepten von GDI+ für den Umsteiger.

Ein zentrales Grafikausgabe-Objekt

Im Gegensatz zur Vorgehensweise in GDI, bei der Sie die Grafikmethoden von einzelnen Komponenten genutzt haben, arbeiten Sie jetzt mit einem einzigen *Graphics*-Objekt, das Sie bestimmten Objekten zuordnen können bzw. von diesen ableiten. Nur dieses Objekt verfügt über relevante Grafikmethoden und -eigenschaften.

Die Grafikausgabe ist zustandslos

Eine der wichtigsten und zugleich einschneidendsten Änderungen betrifft die Organisation der Grafikausgabe. Haben Sie bisher zunächst die Parameter für Linien (Breite, Farbe, Style etc.), Pinsel oder Schriftarten gesetzt und nachfolgend mit diesen Grafikausgaben getätigt, ist dafür nun jede einzelne Grafikmethode zuständig. Das heißt, für eine Grafikmethode ist es vollkommen unerheblich, welche Linienart vorher genutzt wurde. Alle erforderlichen Parameter werden an die Methode bei **jedem** Aufruf übergeben. Die gleiche Vorgehensweise trifft auch auf Texte oder Pinsel zu.

Prinzipieller Ablauf

Wurde bisher nach dem Schema

- Grafikobjekte (Pen, Brush etc.) erzeugen,
- beliebige Zeichenfunktionen (z. B. LineTo) aufrufen,
- Grafikobjekte löschen

verfahren, müssen bei einer zustandslosen GDI+-Programmierung genau diese Operationen **für jeden** einzelnen Methodenaufruf ausgeführt werden, da GDI+ die GDI-Objekte im alten Zustand zurücklassen muss.

Beispiel 4.1: Zeichnen einer Linie im Formular

C#

Neuen Stift erzeugen:

```
Pen pen = new Pen(Color.Aqua);
```

Graphics-Objekt für das Formular erzeugen:

```
Graphics g = CreateGraphics();
```

Linie mit dem neuen Stift zeichnen:

```
g.DrawLine(pen, 10, 10, 100, 100);
```

Wichtige Features

Die wichtigsten Features auf einen Blick:

- GDI+ bietet eine verbesserte Farbverwaltung sowie mehr vordefinierte Farben.
- GDI+ unterstützt eine breite Palette an Bildformaten (.bmp, .gif, .jpeg, .exif, .png, .tiff, .ico, .wmf, .emf).
- Antialiasing-Funktionalität
- Farbverläufe für Pinsel
- Splines
- Bildtransformationen (Rotation, Translation, Skalierung)
- Gleitkommaunterstützung
- Unterstützung des Alpha-Kanals und damit auch Alpha-Blending
- …

4.1.2 Namespaces für die Grafikausgabe

Bevor Sie im weiteren Verlauf mit der Fehlermeldung „Der Typ XYZ ist nicht definiert" konfrontiert werden, möchten wir Ihnen die im Zusammenhang mit der Grafikausgabe relevanten Namespaces vorstellen:

- *System.Drawing*
- *System.Drawing.Drawing2D*
- *System.Drawing.Imaging*
- *System.Drawing.Printing*
- *System.Drawing.Design*
- *System.Drawing.Text*

Die ausführliche Behandlung der einzelnen Strukturen und Objekte erfolgt in den weiteren Abschnitten, hier nur eine kurze Übersicht.

System.Drawing

Dieser standardmäßig eingebundene Namespace enthält die meisten Klassen, Typen, Auflistungen etc., die Sie für die Basisfunktionen der Grafikausgabe benötigen.

Typ	Beschreibung
Color	… verwaltet ARGB-Farbwerte (Alpha-Rot-Grün-Blau), Konvertierungsfunktionen sowie diverse vordefinierte Farbkonstanten
Point *PointF*	… verwaltet 2-D-Koordinaten (x, y) als Integer- bzw. als Floatwert
Rectangle *RectangleF*	… verwaltet die Koordinaten eines Rechtecks als Integer- bzw. Floatwert
Size *SizeF*	… verwaltet die Größe eines rechteckigen Bereichs (Breite, Höhe) als Integer- bzw. Floatwert

Objekt	Beschreibung
Graphics	**Das zentrale Grafikausgabe-Objekt**
Pen	Objekt für die Definition des Linientyps
Brush, Brushes, SolidBrush, TextureBrush	Objekte für die Definition des Füllstils (Pinsel) von Objekten (Kreise, Rechtecke etc.)
Font, FontFamily	Objekt für die Definition von Schriftarten (Farbe, Größe etc.)
Bitmap, Image	Objekt für die Verwaltung von Bitmaps bzw. Grafiken

System.Drawing.Drawing2D

Dieser Namespace bietet Funktionen für Farbverläufe sowie Unterstützung für 2D- und Vektorgrafiken (Matrix für geometrische Transformationen).

System.Drawing.Imaging

Mithilfe dieses Namespace können Sie erweiterte Funktionen wie

- Direktzugriff auf Bitmap-Daten (Pointer),
- Unterstützung für Metafiles,
- Farbverwaltung,
- Grafikkonvertierung,
- Abfrage von Grafikeigenschaften

realisieren.

System.Drawing.Printing

Basisklassen für die Druckausgabe. Mehr dazu in Kapitel 5 (Druckausgabe).

System.Drawing.Design

Dieser Namespace enthält einige Klassen, die für die Entwurfszeit-Oberfläche genutzt werden.

System.Drawing.Text

Abfrage von Informationen über die installierten Schriftarten.

4.2 Darstellen von Grafiken

Bevor wir Sie in den folgenden Abschnitten mit Dutzenden von Objekten, Methoden und Eigenschaften überfrachten, wollen wir uns die wohl trivialste Form der Grafikdarstellung

etwas näher ansehen. Die Rede ist von der *PictureBox*-Komponente, die in diesem Zusammenhang alle wesentlichen Aufgaben übernehmen kann. Untrennbar mit dieser Komponente ist auch das *Image*-Objekt verbunden, über das die eigentlichen Grafikdaten verwaltet werden (Laden, Manipulieren, Eigenschaften).

4.2.1 Die PictureBox-Komponente

Platzieren Sie zunächst eine *PictureBox* im Formular, können Sie diese wie jedes andere Objekt positionieren und die Größe bestimmen. Über die Eigenschaft *Image* lässt sich bereits zur Entwurfszeit eine Grafikdatei zuweisen, die nachfolgend fest in das Projekt und damit auch die EXE-Datei übernommen wird.

Wie diese Grafik formatiert, das heißt skaliert wird, bestimmen Sie mit der *SizeMode*-Eigenschaft:

Normal

CenterImage

StretchImage

Autosize

Wie Sie den obigen Abbildungen entnehmen können, skaliert die *Stretch*-Variante zwar die Grafik auf die gewünschte Größe, leider bleibt dabei aber das Seitenverhältnis der Grafik auf der Strecke.

Wie Sie die Proportion zur Laufzeit wiederherstellen können, zeigt das folgende Beispiel.

Beispiel 4.2: Bild proportional skalieren

C#

```
float xy = Convert.ToSingle(pictureBox1.Image.Width) /
     Convert.ToSingle(pictureBox1.Image.Height);
if (pictureBox1.Image.Width > pictureBox1.Image.Height)
{
   pictureBox1.Height = (int) (pictureBox1.Width / xy);
}
else
{
   pictureBox1.Width = (int) (pictureBox1.Height * xy);
}
```

Die Vorgehensweise: Sie weisen der Eigenschaft *SizeMode* den Wert *Stretch* zu und skalieren jeweils die Außenmaße der *PictureBox* so, dass die Grafik möglichst optimal angezeigt wird.

Über die *BorderStyle*-Eigenschaft können Sie zwischen *keinem*, *einfachem* und *dreidimensionalem* Rahmen wählen.

Damit sind auch schon alle wesentlichen Eigenschaften der *PictureBox* beschrieben. Ziemlich dürftig, werden Sie sicher sagen. Aber dieser Eindruck täuscht, wenn wir uns die bereits erwähnte Eigenschaft *Image* näher ansehen, bei der es sich um ein recht komplexes Objekt handelt.

4.2.2 Das Image-Objekt

Mit dem *Image*-Objekt bietet sich nicht nur die Möglichkeit, zur Entwurfszeit eine Grafik in die *PictureBox* einzubetten, sondern es stellt auch diverse Grafikbearbeitungsfunktionen und die dazu nötigen Informationen über die Grafik bereit.

Zwei Wege führen zum *Image*-Objekt:

- Haben Sie bereits zur Entwurfszeit eine Grafik geladen, können Sie direkt über *Picture.Image* auf die gewünschten Eigenschaften/Methoden zugreifen.
- Andernfalls müssen Sie zunächst ein *Image*-Objekt erzeugen und der *PictureBox* zuweisen.

Beispiel 4.3: Erzeugen eines neuen Image-Objekts (eine Bitmap 100 x 100 Pixel)[1]

C#

```
Image img;
img = new Bitmap(100, 100);
pictureBox2.Image = img;
```

[1] Dies ist nur eine Variante!

Alternativ lässt sich ein *Image* auch aus einer Grafikdatei erzeugen:

```
Image img;
img = Image.FromFile("test.jpg");
pictureBox2.Image = img;
```

Womit wir auch schon beim Laden von Grafiken angekommen sind.

4.2.3 Laden von Grafiken zur Laufzeit

Wie bereits im vorhergehenden kurzen Beispiel gezeigt, stellt es kein Problem dar, ein *Image* aus einer bereits existierenden Datei zu laden. Je nach Dateityp handelt es sich beim *Image* nachfolgend um eine Pixel-, Vektorgrafik (WMF, EMF) oder um ein Icon (ICO).

HINWEIS: Doch Vorsicht: Nicht jedes Grafikformat unterstützt auch alle Grafikmethoden des *Image*-Objekts. Laden Sie beispielsweise eine WMF-Grafik in das Image, können Sie diese nicht mit der Funktion *RotateFlip* drehen oder spiegeln.

Beispiel 4.4: Grafikformat bestimmen

C#

Statt mit

```
Image img;
img = Image.FromFile("test.bmp");
pictureBox1.Image = img;
```

... können Sie auch direkt den gewünschten *Image*-Typ erzeugen:

```
Image img;
img = Bitmap.FromFile("test.bmp");
pictureBox1.Image = img;
```

Das Resultat ist in beiden Fällen das Gleiche.

4.2.4 Sichern von Grafiken

Ähnlich einfach wie das Laden ist auch das Speichern von Grafiken. Mit der *Image*-Methode *Save* können Sie die Grafik in einem der unterstützten Dateiformate sichern.

Beispiel 4.5: Speichern im PNG-Format

C#

```
pictureBox1.Image.Save("test1.png");
```

Auf diese Weise können Sie auch einen Dateikonverter programmieren, Sie brauchen nicht einmal eine *PictureBox* dafür.

Beispiel 4.6: Konvertieren vom BMP- ins PNG-Format

C#

```
using System.Drawing.Imaging;
...
      Image img = Bitmap.FromFile("test.bmp");
      img.Save("test2.png", ImageFormat.Png);
```

Spezielle Einstellungen

Leider, und das scheint bei fast allen universellen Bibliotheken der Fall zu sein, ist die praktische Verwendung teilweise recht eingeschränkt bzw. umständlich. Möchten Sie beispielsweise im JPEG-Format speichern, ist dies kein Problem, doch was, wenn Sie auch den Kompressionsfaktor beeinflussen wollen?

In diesem Fall kommen Sie um etwas mehr Programmierung nicht herum.

Beispiel 4.7: Festlegen eines Kompressionsfaktors von 60% für die zu speichernde JPEG-Datei

C#

```
 private void button1_Click(object sender, EventArgs e)
 {
   EncoderParameters myEncoderParameters = new EncoderParameters(1);
   myEncoderParameters.Param[0] = new EncoderParameter(
          System.Drawing.Imaging.Encoder.Quality, (long) 60);
   pictureBox1.Image.Save("c:\\test.jpg", GetEncoderInfo("image/jpeg"),
myEncoderParameters);
   pictureBox2.Image = Bitmap.FromFile("c:\\test3.jpg");
 }
```

Die notwendige Hilfsfunktion für die Rückgabe des passenden *ImageCodecInfo*-Objekts:

```
 private ImageCodecInfo GetEncoderInfo(String mt)
 {
   ImageCodecInfo[] encoders = ImageCodecInfo.GetImageEncoders();
   for (int i=0; i < encoders.Length - 1; i++)
   {
      if (encoders[i].MimeType == mt)
      {
          return encoders[i];
      }
   }
   return null;
 }
```

4.2.5 Grafikeigenschaften ermitteln

Sie können auch per Code bestimmte Grafikeigenschaften ermitteln.

Breite und Höhe der Grafik

Breite und Höhe der Grafik können Sie über die Eigenschaften *Width* und *Height* des *Image*-Objekts abrufen.

HINWEIS: Die Maße des *Image*-Objekts, das heißt der Grafik, stehen in keinem Zusammenhang mit den Maßen der *PictureBox*.

Auflösung

Die vertikale bzw. horizontale Auflösung der Grafik können Sie mit den Eigenschaften *VerticalResolution* bzw. *HorizontalResolution* abfragen. Beide geben einen Wert in *dpi* (Punkte pro Inch) zurück.

Grafiktyp

Den aktuell geladenen Grafiktyp fragen Sie über die Eigenschaft *RawFormat* ab. Rückgabewerte sind die verschiedenen Bildtypen, die von der *Image*-Komponente unterstützt werden (BMP, GIF etc.).

Interner Bitmap-Aufbau

Um welchen Typ von Bitmap (Farbtiefe) bzw. um welches Speicherformat (RGB, ARGB etc.) es sich handelt, verrät Ihnen die Eigenschaft *PixelFormat*. An dieser Stelle handelt es sich jedoch lediglich um eine schreibgeschützte Eigenschaft. Möchten Sie Einfluss auf das Speicherformat nehmen, sollten Sie sich näher mit dem Kapitel 7 beschäftigen.

4.2.6 Erzeugen von Vorschaugrafiken (Thumbnails)

Sicher sind Ihnen unter Windows auch schon die kleinen Vorschaugrafiken in den Explorerfenstern aufgefallen. Bevor Sie jetzt an aufwendige Algorithmen und Funktionen zur Skalierung von Bitmaps denken, vergessen Sie es besser wieder. GDI+ unterstützt Sie an dieser Stelle mit einer einfach verwendbaren Methode:

Syntax:

```
Image GetThumbnailImage(int thumbWidth,int thumbHeight,
                        Image.GetThumbnailImageAbort callback,IntPtr
callbackData);
```

Die beiden ersten Parameter dürften leicht verständlich sein, *callback* erwartet einen Verweis auf einen Delegate (optional *null*). *CallbackData* übergeben Sie grundsätzlich *IntPtr.Zero*.

Leider müssen Sie sich um die Proportionen des zu erzeugenden Image selbst kümmern. Übergeben Sie beispielsweise 100 für Breite und Höhe und haben Sie eine Ausgangsgrafik mit anderem Höhen-/Seitenverhältnis, wird die Vorschaugrafik gestaucht, was sicher nicht sehr professionell aussieht. Fragen Sie also vorher das Höhen-/Seitenverhältnis der Grafik ab und setzen Sie Breite und Höhe entsprechend.

Beispiel 4.8: Funktion zum Erzeugen einer proportionalen Vorschaugrafik

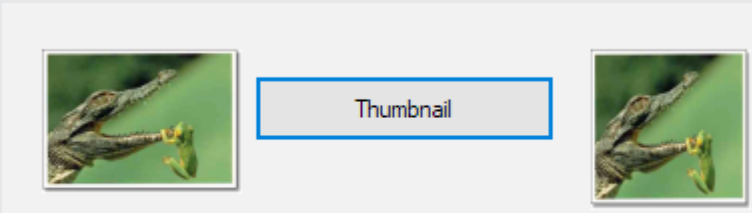
C#

```
    private Image GetThumb(int size,Image img)
    {
      float xy = Convert.ToSingle(img.Width / img.Height);
      if (xy > 1)
      {
         return img.GetThumbnailImage(size, Convert.ToInt32(size / xy), null,
IntPtr.Zero);
      }
      else
      {
         return img.GetThumbnailImage(Convert.ToInt32(size * xy), size, null,
IntPtr.Zero);
      }
    }
```

Der Aufruf ist einfach:

```
    private void button2_Click(object sender, EventArgs e)
    {
       pictureBox3.Image = GetThumb(80, pictureBox1.Image);
    }
```

Ergebnis



HINWEIS: Enthält die Ausgangsgrafik bereits eine Thumbnail-Grafik (z. B. bei JPG-Format), wird diese für die Methode *GetThumbnailImage* genutzt, um nicht die ganze Bitmap in den Speicher zu laden. Dies ist allerdings maximal bis zu einer Auflösung von 120 x 120 Pixeln sinnvoll. Benötigen Sie größere Thumbnails, sollten Sie besser die Methode *DrawImage* verwenden, um eine verkleinerte Kopie der Ausgangsbitmap zu erzeugen.

4.2.7 Die Methode RotateFlip

Mit der Methode *RotateFlip* stehen Ihnen rudimentäre Manipulationsfunktionen für die Grafikausgabe zur Verfügung (drehen, spiegeln). Übergeben Sie der Methode einfach die gewünschte Konstante und aktualisieren Sie gegebenenfalls die übergeordneten *PictureBox*-Komponenten, schon ist die Grafik gedreht oder gespiegelt.

Beispiel 4.9: Drehen um 180°, vertikal kippen

C#

```
pictureBox1.Image.RotateFlip(RotateFlipType.Rotate180FlipY);
pictureBox1.Refresh();
```

Ein „Schweizer Taschenmesser“, was die Funktionalität anbelangt, könnte man auf den ersten Blick denken, doch wer sich die Liste der verschiedenen Möglichkeiten (Konstanten) einmal näher ansieht, wird feststellen, dass einige Varianten schlicht überflüssig sind. Ausgangspunkt für die Beispielgrafiken ist folgende „aufwendige“ Grafik:

Konstante	Beschreibung	Resultat
Rotate180FlipNone	Drehung um 180 Grad	
Rotate180FlipX	Drehung um 180 Grad, horizontal kippen	
Rotate180FlipXY	Drehung um 180 Grad, horizontal und vertikal kippen	
Rotate180FlipY	Drehung um 180 Grad, vertikal kippen	
Rotate270FlipNone	Drehung um -90 Grad	
Rotate270FlipX	Drehung um -90 Grad, horizontal kippen	
Rotate270FlipXY	Drehung um -90 Grad, horizontal und vertikal kippen	
Rotate270FlipY	Drehung um -90 Grad, vertikal kippen	
Rotate90FlipNone	Drehung um 90 Grad	
Rotate90FlipX	Drehung um 90 Grad, horizontal kippen	
Rotate90FlipXY	Drehung um 90 Grad, horizontal und vertikal kippen	
Rotate90FlipY	Drehung um 90 Grad, vertikal kippen	
RotateNoneFlipX	horizontal kippen	
RotateNoneFlipXY	horizontal und vertikal kippen	
RotateNoneFlipY	vertikal kippen	

Eine Rotation um 180° (*Rotate180FlipNone*) entspricht einem Kippen in vertikaler und horizontaler Richtung (*RotateNoneFlipXY*). Auch bei der Kombination *Rotate90FlipXY* bzw. *Rotate270FlipXY* scheint mit den Entwicklern der Spieltrieb durchgegangen zu sein. Das gleiche Ergebnis kann auch mit *Rotate270FlipNone* bzw. mit *Rotate90FlipNone* erreicht werden.

Unangefochtener „Spitzenreiter" in der Tabelle ist *Rotate180FlipXY*.

4.2.8 Skalieren von Grafiken

Eine Grafik mit GDI+ zu skalieren, stellt eines der kleinsten Probleme dar[2]. Mit einer einzigen Zeile Code können Sie beispielsweise die Größe einer Grafik verdreifachen.

Beispiel 4.10: Bitmap auf dreifache Größe skalieren

C#

```
pictureBox2.Image = new Bitmap(pictureBox1.Image,
                    new Size(pictureBox1.Image.Width * 3, pictureBox1.Image.Height
* 3));
```

Wir nehmen diese Anweisung jetzt noch einmal auseinander bzw. deklarieren die Objekte für eine bessere Übersicht:

Eine temporäre Bitmap deklarieren:

```
Bitmap b;
```

Eine Variable für die neue Größe deklarieren:

```
Size s;
```

Die neue Größe bestimmen:

```
        s.Width = pictureBox1.Image.Width * 3;
        s.Height = pictureBox1.Image.Height * 3;
```

Die neue Bitmap erzeugen:

```
        b = new Bitmap(PictureBox1.Image, s);
```

Die Bitmap dem Image zuweisen:

```
        pictureBox2.Image = b;
```

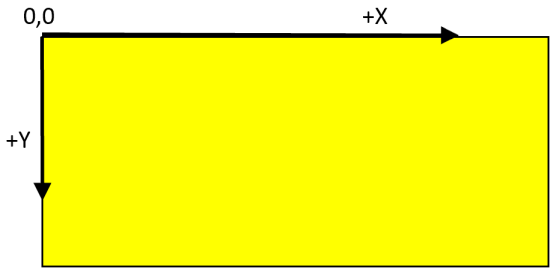
Damit sind die wichtigsten Möglichkeiten von *PictureBox* und *Image* aufgezählt, wir können uns der eigentlichen Grafikprogrammierung, das heißt dem Erzeugen von Grafiken, zuwenden.

[2] Sieht man einmal von potenziellen Speicherproblemen bei großen Bitmaps ab.

4.3 Das .NET-Koordinatensystem

Bevor Sie sich mit dem Zeichnen von Linien, Kreisen oder der Ausgabe von Bitmaps beschäftigen, sollten Sie einen Blick auf das .NET-Koordinatensystem werfen.

Auf den ersten Blick bietet sich nichts Ungewohntes, der Koordinatenursprung für alle Grafikmethoden liegt in der linken oberen Ecke des jeweils gewählten Ausgabefensters (das kann auch ein Steuerelement sein). Positive x-Werte werden nach rechts, positive y-Werte nach unten abgetragen:



Die Maßeinheit für Ausgaben ist standardmäßig das Pixel. Alternativ kann es jedoch bei einer Druckausgabe von Vorteil sein, die Maßeinheit zum Beispiel in Millimeter zu ändern. Nutzen Sie dazu die Eigenschaft *PageUnit* des *Graphics*-Objekts. Folgende Werte sind zulässig:

Wert	Beschreibung
Display	1/100 Zoll
Document	1/300 Zoll
Inch	Zoll (2,54 cm)
Millimeter	Millimeter
Pixel	(Standard) Bildschirmpixel
Point	1/72 Zoll

GDI+ unterscheidet im Zusammenhang mit der Grafikausgabe drei verschiedene Koordinatensysteme:

- globale Koordinaten,
- Seitenkoordinaten und
- Gerätekoordinaten.

4.3.1 Globale Koordinaten

Hierbei handelt es sich um das ursprüngliche Koordinatensystem. Dieses wird über bestimmte Transformationen in Seiten- und damit auch Gerätekoordinaten umgewandelt.

Ausgangspunkt bei der Maßeinheit Pixel: Ein Pixel in globalen Koordinaten entspricht zunächst einem Pixel auf dem endgültigen Ausgabegerät (Bildschirm, Drucker).

Beispiel 4.11: Globale Koordinaten

C#

Es soll ein Kreis mit dem Radius 50 (Pixel) auf dem Formular gezeichnet werden (an dieser Stelle müssen wir leider etwas vorgreifen, mit den folgenden Anweisungen wird im *Paint*-Ereignis des Formulars der Kreis gezeichnet):

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
   e.Graphics.DrawEllipse(new Pen(Color.Black), -50, -50, 100, 100);
}
```

Ergebnis

Das Ergebnis auf dem Bildschirm:

4.3.2 Seitenkoordinaten (globale Transformation)

Möchten Sie den Nullpunkt des Koordinatensystems verschieben, sodass der gesamte Kreis sichtbar wird, müssen Sie den Nullpunkt verschieben, das heißt, eine globale Transformation durchführen.

Translation (Verschiebung)

Mithilfe der Methode *TranslateTransform* lässt sich diese Aufgabe bewältigen.

Beispiel 4.12: Translation

C#

Verschieben des Nullpunkts um jeweils 50 Pixel in x- und y-Richtung

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
   e.Graphics.TranslateTransform(50, 50);
   e.Graphics.DrawEllipse(new Pen(Color.Black), -50, -50, 100, 100);
}
```

Ergebnis

Das Ergebnis der Grafikausgabe:

Skalierung (Vergrößerung/Verkleinerung)

Ähnlich verhält es sich mit einer Skalierung zwischen globalen und Seitenkoordinaten: Möchten Sie beispielsweise den Kreis auf dem Bildschirm doppelt so groß ausgeben, können Sie dies mit der Eigenschaft *PageScale* steuern.

Beispiel 4.13: Skalierung

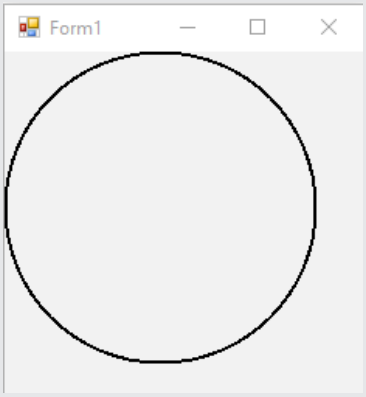
C#

Ein Pixel global soll zwei Pixeln Seiteneinheit entsprechen:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    e.Graphics.TranslateTransform(50, 50);
    e.Graphics.PageScale = 2;
    e.Graphics.DrawEllipse(new Pen(Color.Black), -50, -50, 100, 100);
}
```

Ergebnis

Wie Sie sehen, sind alle Zeichenanweisungen, inklusive der Strichstärke, von dieser Skalierung betroffen:



Alternativ können Sie mit *ScaleTransform* auch unterschiedliche Skalierungsfaktoren für x und y einführen.

Beispiel 4.14: Skalierung in x-Richtung (zweifach)

C#

```
e.Graphics.ScaleTransform(2, 1);
e.Graphics.TranslateTransform(50, 50);
e.Graphics.DrawEllipse(new Pen(Color.Black), -50, -50, 100, 100);
```

Ergebnis

HINWEIS: Sollen auch Images im korrekten Verhältnis wiedergegeben werden, müssen Sie *ScaleTransform* verwenden!

Rotation

Als letzte Variante bleibt das Drehen des Koordinatensystems. Verantwortlich dafür ist die Methode *RotateTransform*, der Sie einfach den Drehwinkel übergeben.

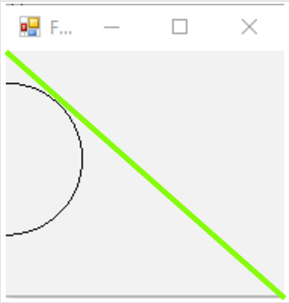
Beispiel 4.15: Drehen des Koordinatensystems um 45° (Uhrzeigersinn)

C#

```
e.Graphics.RotateTransform(45);
e.Graphics.TranslateTransform(50, 50);
e.Graphics.DrawEllipse(new Pen(Color.Black), -50, -50, 100, 100);
```

Ergebnis

Die folgende Abbildung zeigt das Resultat, die eingezeichnete Linie deutet die Position der x-Achse (y = 0) an:



Dass nicht nur einfache Objekte wie Linien oder Kreise von den Transformationen betroffen sind, zeigt das folgende Beispiel.

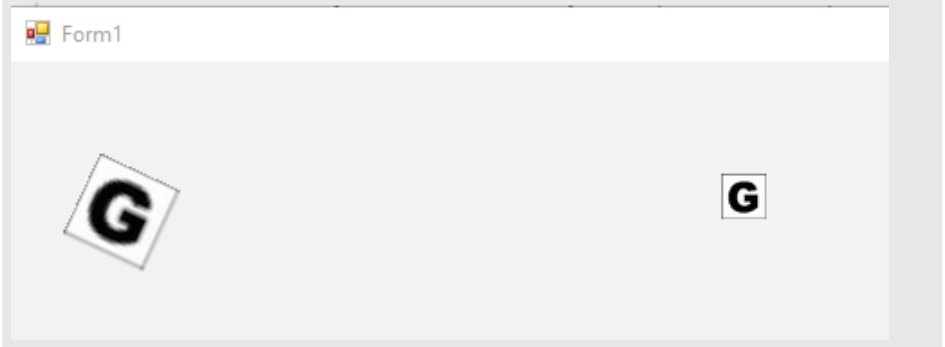
Beispiel 4.16: Bitmap-Transformation

C#

Drehen einer Bitmap um 25°, Skalieren um 100%, Verschieben des Nullpunkts in das Formular (50, 50)

```
e.Graphics.TranslateTransform(50, 50);
e.Graphics.RotateTransform(25);
e.Graphics.ScaleTransform(2, 2);
e.Graphics.DrawImage(pictureBox1.Image, 0, 0);
```

Ergebnis



4.3.3 Gerätekoordinaten (Seitentransformation)

Das dritte Koordinatensystem bezieht sich auf das endgültige Ausgabegerät, z. B. ein Drucker oder ein Bildschirm. Diese Geräte verfügen über vorgegebene Auflösungen, die in Dots per Inch (dpi) angegeben werden. Beispielsweise verfügt der Bildschirm meist über eine Auflösung von 96 dpi, Laserdrucker werden mit Auflösungen von 300 bis weit über 1000 dpi angeboten.

Solange Sie nicht mit Geräteeinheiten arbeiten, werden Sie je nach Geräteauflösung zu unterschiedlichen Ergebnissen kommen. Aus diesem Grund unterstützt das .NET-Koordinatensystem auch gerätespezifische Einheiten wie Millimeter oder Inch. Über die Eigenschaft *PageUnit* können Sie die Einheit für Ihr Koordinatensystem ändern.

Beispiel 4.17: Bildschirmanzeige in Millimetern

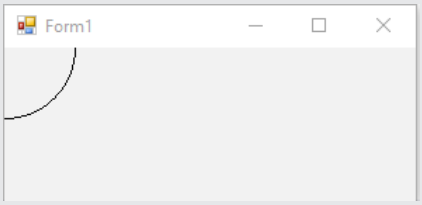
C#

```
e.Graphics.PageUnit = GraphicsUnit.Millimeter;
e.Graphics.DrawEllipse(new Pen(Color.Black), -50, -50, 100, 100);
```

Ergebnis

Die beiden folgenden Abbildungen zeigen das Ergebnis für Pixel bzw. Millimeter:

Pixel



Millimeter

Frage: Wie groß ist bei der rechten Abbildung der Radius in Pixeln?

Die Antwort: Bei einer Bildschirmauflösung von 96 dpi ergibt sich ein Faktor von ca. 3,7 Pixeln pro Millimeter. Multipliziert mit dem Radius von 50 mm erhalten Sie 188 Pixel. Sie können das berechnete Ergebnis zum Beispiel mit Paintbrush „nachmessen".

HINWEIS: Weitere Anwendungsbeispiele für Koordinatentransformationen finden Sie in den Kapiteln 5 und 7.

4.4 Grundlegende Zeichenfunktionen von GDI+

Im vorhergehenden Abschnitt hatten Sie ja bereits ersten Kontakt mit einigen Zeichenfunktionen, auf die wir nun in diesem Abschnitt näher eingehen wollen.

4.4.1 Das zentrale Graphics-Objekt

Wie bereits angedeutet, sind mit GDI+ die Zeiten des unbekümmerten Programmierens vorbei. Um den ausufernden Möglichkeiten etwas Einhalt zu gebieten und eine einheitliche Schnittstelle für die Grafikausgabe zu schaffen, wurde die *Graphics*-Klasse eingeführt.

Nur Objekte dieses Typs verfügen über Grafikausgabemethoden und die nötigen Parameter. Alle anderen Komponenten, und dazu zählen auch Formulare und Drucker, können lediglich *Graphics*-Objekte zur Verfügung stellen.

Wie erzeuge ich ein Graphics-Objekt?

Drei wesentliche Varianten bieten sich an:

- Für Formulare können Sie das *Paint* nutzen, der übergebene Parameter *e* stellt auch ein *Graphics*-Objekt zur Verfügung. -Ereignis
- Für alle anderen Komponenten nutzen Sie die *CreateGraphics*-Methode des jeweiligen Objekts. Diese gibt ein *Graphics*-Objekt zurück.
- Möchten Sie auch auf *Images* (BMP etc.) per *Graphics*-Objekt zugreifen, verwenden Sie einfach die *Graphics.FromImage*-Methode.

Beispiel 4.18: Zeichnen mit *Paint*-Ereignis

C#

```
private void Form1_Paint(object sender,PaintEventArgs e)
{
  e.Graphics.DrawEllipse(new Pen(Color.Black), -50, -50, 100, 100);
}
```

Beispiel 4.19: Verwenden von *CreateGraphics*

C#

```
private void button1_Click(object sender, EventArgs e)
{
  Graphics g = CreateGraphics();
  g.DrawEllipse(new Pen(Color.Black), 0, 0, 100, 100);
}
```

Um die Ausgabe auf einem Button statt auf dem Formular vorzunehmen, genügt es, wenn Sie das *Graphics*-Objekt mit *button1.CreateGraphics* erzeugen.

Beispiel 4.20: Verwenden von *Graphics.FromImage*

C#

```
private void button2_Click(object sender, EventArgs e)
{
   Graphicsg = Graphics.FromImage(pictureBox1.Image);
   g.DrawEllipse(new Pen(Color.Black), 0, 0, 100, 100);
   pictureBox1.Refresh();
}
```

HINWEIS: Für alle Grafikausgaben gilt: Nach dem Verdecken eines Formulars müssen Sie sich um das Aktualisieren der Bildschirmanzeige selbst kümmern. Einzige Ausnahme ist die *PictureBox*, wenn Sie, wie im letzten Beispiel gezeigt, über *Graphics* in das *Image* schreiben.

Die Invalidate-Methode

Nutzen Sie das *Paint*-Ereignis für die Ausgabe der Grafik und muss diese zwischenzeitlich aktualisiert werden, können Sie die Methode *Invalidate* zum Aktualisieren der Anzeige verwenden.

Beispiel 4.21: Mittels *Timer* soll ein kontinuierlich größer werdender Kreis gezeichnet werden.

C#

Globale Variable für den Offset:

```
private int x;
```

Das *Tick*-Ereignis:

```
   private void timer1_Tick(object sender, EventArgs e)
   {
      x++;
      Invalidate();
   }
```

Die eigentliche Zeichenroutine im *Paint*-Ereignis:

```
   private void Form1_Paint(object sender,PaintEventArgs e)
   {
      e.Graphics.DrawEllipse(new Pen(Color.Black), 0, 0, 100 + x, 100 + x);
   }
```

Im Weiteren wollen wir uns mit den wichtigsten Grafikgrundoperationen beschäftigen, auch wenn Sie bereits einige in den vorhergehenden Beispielen kennengelernt haben.

Die Eigenschaft ResizeRedraw

So bequem die Verwendung des *Paint*-Ereignisses auch ist, einen Nachteil werden Sie schnell bemerken, wenn das Formular skaliert bzw. kurzzeitig verdeckt wurde. In diesem Fall werden nur die neuen bzw. die verdeckten Flächen neu gezeichnet. Dies kann allerdings dazu führen, dass Ihre Grafik nicht mehr wie gewünscht ausgegeben wird. Um diesem Missstand abzuhelfen, steht über das Formular die Eigenschaft *ResizeRedraw* zur Verfügung. Ist diese auf *true* gesetzt, wird immer der komplette Clientbereich neu gezeichnet.

Beispiel 4.22: Auswirkung von *ResizeRedraw*

C#

Zeichnen einer Ellipse im Clientbereich des Formulars:

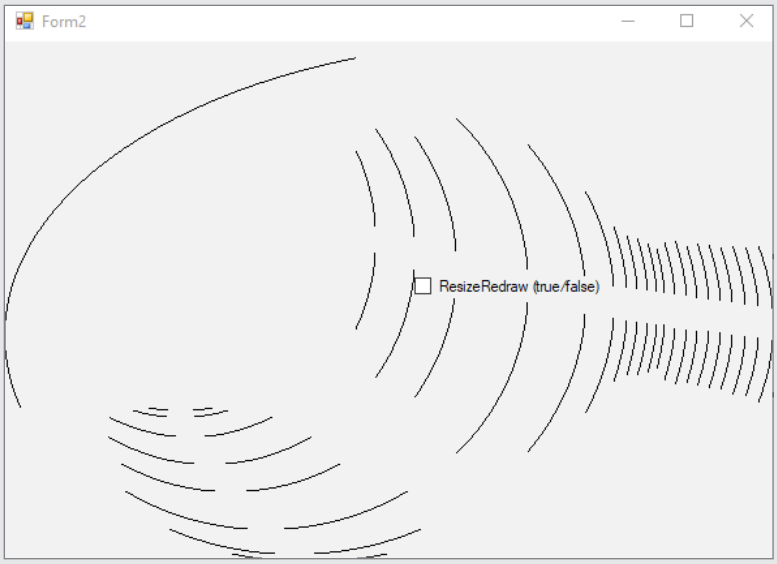
```
private void Form2_Paint(object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   g.DrawEllipse(new Pen(Color.Black),0, 0,ClientSize.Width,ClientSize.Height);
}
```

Per *CheckBox* die *ResizeRedraw-Eigenschaft* beeinflussen:

```
private void checkBox1_CheckedChanged(object sender, EventArgs e)
{
    ResizeRedraw = checkBox1.Checked;
}
```

Ergebnis

ResizeRedraw=false:



ResizeRedraw=true:

4.4.2 Punkte zeichnen/abfragen

Tja, da sieht es gleich ganz düster aus. Eine entsprechende Funktion existiert zunächst nicht. Einzige Ausnahme: die *GetPixel-/SetPixel*-Methoden des *Bitmap*-Objekts.

Syntax:

```
Color GetPixel(int x, int y)
```

Syntax:

```
void SetPixel(int x, int y, Color color)
```

Beiden Methoden übergeben Sie die gewünschten Koordinaten, *GetPixel* gibt Ihnen einen *Color*-Wert zurück, *SetPixel* müssen Sie einen *Color*-Wert übergeben.

HINWEIS: Mehr zu Farben und dem *Color*-Objekt finden Sie in Abschnitt 4.5.3.

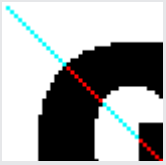
Beispiel 4.23: Setzen von Pixeln in Abhängigkeit von der vorhergehenden Farbe

C#

```
private void button4_Click(object sender, EventArgs e)
{
  Bitmap b = (Bitmap)pictureBox2.Image;
  for (int x = 1; x <=20; x++)
  {
    if (b.GetPixel(x, x).ToArgb() == Color.Black.ToArgb())
    {
      b.SetPixel(x, x, Color.Red);
    }
    else
    {
      b.SetPixel(x, x, Color.Aqua);
    }
  }
  pictureBox2.Refresh();
}
```

Ergebnis

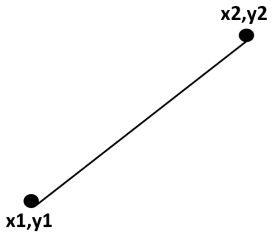
Die folgende Ausschnittsvergrößerung zeigt das Resultat:



4.4.3 Linien

Das Zeichnen von Linien gestaltet sich mit GDI+ relativ einfach, sieht man einmal davon ab, dass Sie bei **jedem** Aufruf von *DrawLine* auch einen geeigneten *Pen* übergeben müssen. An dieser Stelle wollen wir zunächst noch nicht auf die verschiedenen Stifttypen eingehen, mit denen Sie sowohl Farbe, Muster, Endpunkte etc. beeinflussen können. Mehr Infos zu diesem Thema finden Sie erst in Abschnitt 4.5.

Die weiteren Parameter der *DrawLine*-Methode: Entweder Sie übergeben jeweils ein Paar x,y-Koordinaten oder Sie verwenden gleich die vordefinierte Struktur *Point* (eine x- und eine y-Koordinate).



Beispiel 4.24: Verwendung von *DrawLine*

C#

```
Graphics g = CreateGraphics();
g.DrawLine(new Pen(Color.Black), 10, 10, 100, 100);
```

Mit *New Pen(Color.Black)* erzeugen wir einen schwarzen Stift mit der Linienbreite 1.

4.4.4 Kantenglättung mit Antialiasing

Im Zusammenhang mit der Ausgabe von Linien kommen wir auch schnell mit einem ersten „Problem“ in Berührung. Die Rede ist von der „Treppchen“- bzw. „Stufen“-Bildung von Linien, die einen von 0° bzw. 90° verschiedenen Winkel aufweisen:

Ursache ist die begrenzte Bildschirmauflösung, es können nur die Pixel gesetzt werden, die in das xy-Raster passen. Zwangsläufig kommt es bei einer Abweichung von Raster und gewünschtem Linienverlauf zu Sprüngen, die sich als hässliche Treppchen bemerkbar machen.

Mithilfe der Antialiasing-Technik können diese Effekte vermindert werden. Hintergrund dieser Technik ist ein „Glätten“ des Linienrands durch Auffüllen mit Pixeln, deren Farbe aus Linien- **und** Hintergrundfarbe berechnet wird.

GDI+ bietet Ihnen zu diesem Zweck die Methode *SetSmoothingMode*, mit der Sie die Form der Kantenglättung beeinflussen können. Zwei Extreme sind möglich:

- *SmoothingModeHighSpeed* (schnell, aber kantig)
- *SmoothingModeHighQuality* (langsam(er), aber sauber)

Beispiel 4.25: Ein- und Ausschalten der Kantenglättung

C#

```
Graphics g = CreateGraphics();
g.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.HighQuality;
g.DrawLine(new Pen(Color.Black), 10, 10, 100, 100);
g.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.HighSpeed;
```

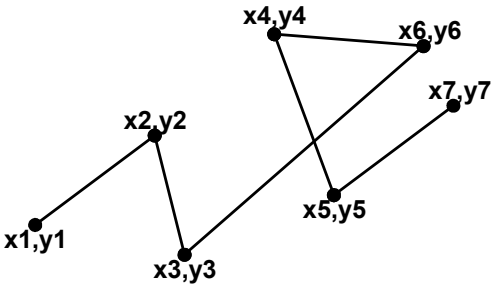
Ergebnis

Die folgende Abbildung zeigt das Ergebnis:

HINWEIS: Diese Form der Kantenglättung wirkt sich nicht auf Textausgaben, sondern nur auf alle Arten von Linien aus. Für Textausgaben verwenden Sie die Eigenschaft *TextRenderingHint*.

4.4.5 PolyLine

Möchten Sie mehr als eine Linie zeichnen und stimmen die Endpunkte mit den Anfangspunkten der folgenden Linien überein, verwenden Sie zum Zeichnen von Linien am besten die Methode *DrawLines*. Übergabewert ist wie gewohnt ein initialisierter *Pen* sowie ein Array von Punkten (*Point* oder *PointF*).



Beispiel 4.26: Zeichnen von zwei Linien

C#

```
      Graphics g = CreateGraphics();
      PointF[] punkte = {new PointF(0, 0), new PointF(100, 100), new PointF(20,
80)};
      g.DrawLines(new Pen(Color.Black), punkte);
```

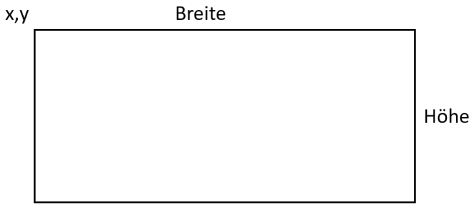
HINWEIS: Selbstverständlich können Sie auch ein frei definiertes Array verwenden, das zur Laufzeit mit Werten gefüllt wird.

4.4.6 Rechtecke

Für das Zeichnen von Rechtecken können Sie entweder die Methode *DrawRectangle* (nur Rahmen) oder *FillRectangle* (gefülltes Rechteck) verwenden.

DrawRectangle

Diese Methode erwartet als Übergabeparameter einen *Pen* sowie die Koordinaten (als x, y, Breite, Höhe oder *Rectangle*-Struktur).



Beispiel 4.27: Zeichnen von Rechtecken

C#

```
Graphics g = CreateGraphics();
g.DrawRectangle(new Pen(Color.Black), 10, 10, 100, 100);
```

Alternativ können Sie auch die folgenden Anweisungen verwenden:

```
Graphics g = CreateGraphics();
Rectangle rec = new Rectangle(10, 10, 100, 100);
g.DrawRectangle(new Pen(Color.Black), rec);
```

FillRectangle

Im Unterschied zur *DrawRectangle*-Methode erwartet *FillRectangle* einen initialisierten *Brush*, das heißt die Information, wie das Rechteck zu füllen ist.

HINWEIS: Die Methode füllt lediglich das Rechteck, es wird kein Rahmen gezeichnet!

Beispiel 4.28: *Rotes Rechteck zeichnen*

C#

```
Graphics g = CreateGraphics();
g.FillRectangle(new SolidBrush(Color.Red), 10, 10, 100, 100);
```

Wie auch bei *DrawRectangle* können Sie alternativ eine *Rectangle*-Struktur übergeben. Dies bietet sich beispielsweise an, wenn man das Rechteck auch mit einem Rahmen versehen will.

Beispiel 4.29: Verwendung *Rectangle*

C#

```
Graphics g = CreateGraphics();
Rectangle rec = new Rectangle(10, 10, 100, 100);

g.FillRectangle(new SolidBrush(Color.Red), rec);
g.DrawRectangle(new Pen(Color.Black), rec);
```

DrawRectangles/FillRectangles

Nicht genug der Pein, neben den beiden genannten Methoden können Sie auch *DrawRectangle*s und *FillRectangle*s verwenden, um gleich mehrere Rechtecke auf einmal zu zeichnen. Übergeben wird in diesem Fall ein Array von *Rectangle*-Strukturen.

Beispiel 4.30: Zeichnen zweier gefüllter Rechtecke

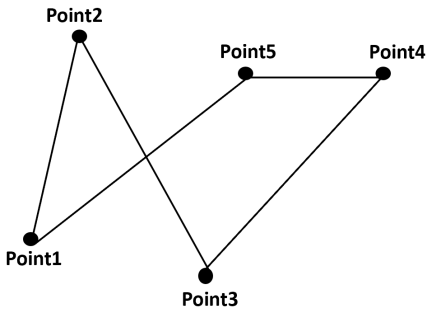
C#

```
Graphics g = CreateGraphics();
Rectangle[] rec = {new Rectangle(10, 10, 100, 100),
                   new Rectangle(50, 50, 200, 120)};

g.FillRectangles(new SolidBrush(Color.Red), rec);
```

Ergebnis

4.4.7 Polygone

Möchten Sie ein n-Eck zeichnen, nutzen Sie die Methode *DrawPolygon*. Soll dieses Vieleck auch gefüllt werden, können Sie dies mit *FillPolygon* realisieren. Beide Funktionen arbeiten ähnlich wie die Funktionen zum Zeichnen von Rechtecken, der einzige Unterschied: *DrawPolygon* bzw. *FillPolygon* erwarten als Übergabeparameter ein Array von *Point*-Strukturen (x- und y-Koordinate) sowie jeweils einen *Pen* bzw. einen *Brush*.

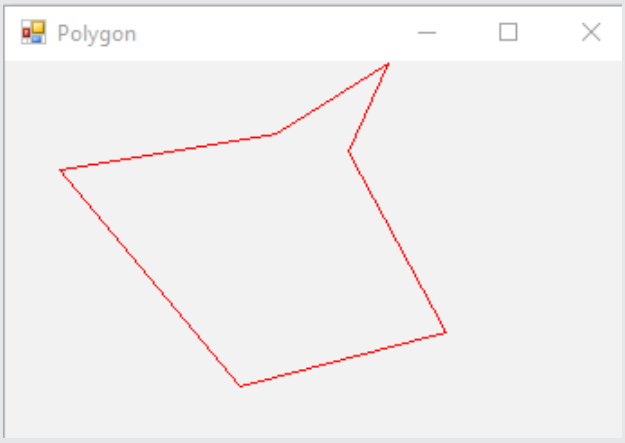


Beispiel 4.31: Zeichnen eines Vielecks

C#

```
Graphics g = e.Graphics;
Point[] ps = new Point[6];
ps[0] = new Point(30, 60);
ps[1] = new Point(150, 40);
ps[2] = new Point(212, 1);
ps[3] = new Point(190, 50);
ps[4] = new Point(244, 150);
ps[5] = new Point(130, 180);
g.DrawPolygon(new Pen(Color.Red), ps);
```

Ergebnis



HINWEIS: Die letzte Linie, zurück zum ersten Punkt, wird automatisch mit gezeichnet, um die Figur zu schließen.

4.4.8 Splines

Möchten Sie Diagramme zeichnen, stehen Sie häufig vor dem Problem, dass die entstehende Kurve ziemlich eckig ist, da Sie aus Zeitgründen nur wenige Stützpunkte berechnet haben. Das lineare Verbinden ist in diesem Fall nicht der ideale Weg. Besser ist hier die Verwendung von Splines, das heißt Linienzügen, die bei einem Wechsel des Anstiegs weiche Übergänge realisieren.

GDI+ bietet Ihnen die Methode *DrawCurve*, der Sie zunächst die gleichen Parameter wie *DrawLines* übergeben können (Stift, Point-Array).

Syntax:

```
void DrawCurve(Pen pen, Point[] points, int Start, int Anzahl, float
Spannung);
```

Zusätzlich können Sie bestimmen, ab welchem Punkt in der Liste die Spline-Kurve gezeichnet wird bzw. wie viele Punkte gezeichnet werden. Der letzte Parameter bestimmt die Spannung, das heißt, wie der Übergang zwischen zwei Teilstrecken hergestellt wird. Ein Beispiel soll für mehr Klarheit sorgen.

Beispiel 4.32: Spline zeichnen

C#

Neues *Graphics*-Objekt erzeugen:

```
Graphics g = e.Graphics;
```

Array für die Punkte erstellen:

```
PointF[] ps = new PointF[6];
```

Die Spannung, verändern Sie diesen Wert wie gewünscht:

```
float u = 0.6F;

ps[0] = new PointF(0, 100);
ps[1] = new PointF(100, 0);
ps[2] = new PointF(200, 100);
ps[3] = new PointF(300, 200);
ps[4] = new PointF(400, 100);
ps[5] = new PointF(500, 0);
```

Zunächst zeichnen wir zum Vergleich eine „normale“ Kurve:

```
g.DrawLines(new Pen(Color.Black, 2), ps);
```

Und jetzt die Spline-Kurve:

```
g.DrawCurve(new Pen(Color.Red, 2), ps, 0, 5, u);
```

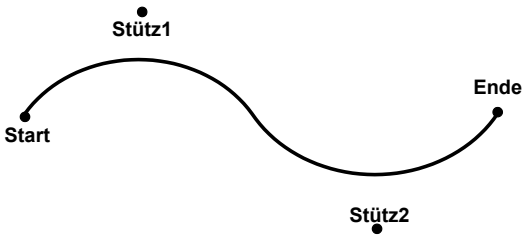
Ergebnis

4.4.9 Bézierkurven

Für einfache Zeichnungen dürften die bisherigen Zeichenfunktionen ausreichen, kompliziertere Gebilde bekommen Sie mit der Bézierfunktion in den Griff. Wie bei einer „normalen“ Linie übergeben Sie Anfangs- und Endpunkt, zusätzlich jedoch noch zwei Stützpunkte, mit denen die Linie wie ein Gummiband „gezogen“ werden kann.

Syntax:

```
void DrawBezier(Pen pen, Point Start, Point Stütz1, Point Stütz2, Point
Ende);
```



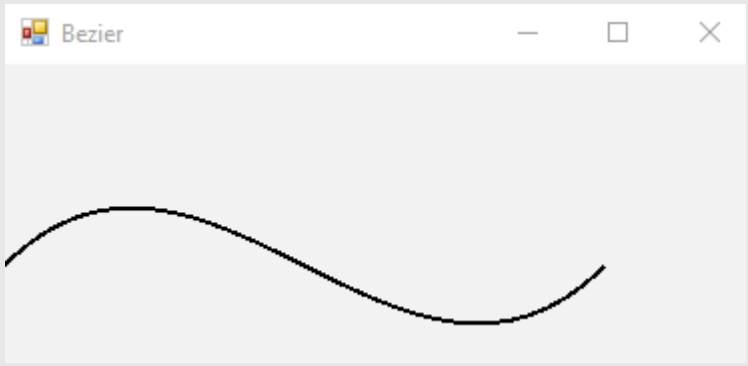
Beispiel 4.33: Zeichnen einer Bézierkurve

C#

```
    Graphics g =e.Graphics;
    Point start;
    Point stütz1;
    Point stütz2;
    Point ende;
```

```
start = new Point(0, 100);
stütz1 = new Point(100, 0);
stütz2 = new Point(200, 200);
ende = new Point(300, 100);
g.DrawBezier(new Pen(Color.Black, 2), start, stütz1, stütz2, ende);
```

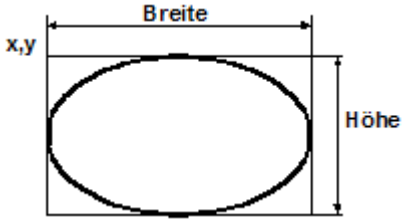
Ergebnis



HINWEIS: Mit *DrawBeziers* können Sie mehrere Bézierkurven gleichzeitig zeichnen.

4.4.10 Kreise und Ellipsen

Kreise und Ellipsen zeichnen Sie mit *DrawEllipse* bzw. *FillEllipse*, wenn Sie eine Kreis-/Ellipsenfüllung erzeugen wollen. Übergabeparameter ist eine *Rectangle*-Struktur bzw. die linke obere Ecke sowie die Breite und Höhe der Ellipse.



HINWEIS: Kreisbögen und Kuchenstücke zeichnen Sie mit eigenen Methoden.

Beispiel 4.34: Zeichnen einer Ellipse

C#

```
Graphics g = e.Graphics;
Rectangle rec = new Rectangle(10, 10, 150, 100);
g.DrawEllipse(new Pen(Color.Black, 2), rec);
```

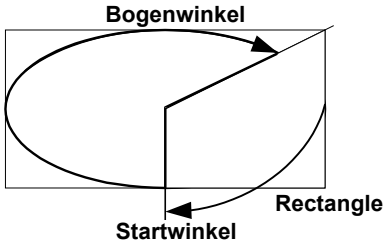
4.4.11 Tortenstück (Segment)

Die Methode *DrawPie* zeichnet ein Tortenstück, das durch eine Ellipse und zwei Linien begrenzt ist. Die Begrenzungslinien werden über den Startwinkel bzw. den Bogenwinkel bestimmt.

Syntax:

```
void DrawPie(Pen pen, RectangleF rect, float startwinkel, float
bogenwinkel);
```

HINWEIS: Winkel werden im Uhrzeigersinn abgetragen.



HINWEIS: Mit *FillPie* erstellen Sie ein gefülltes Tortenstück.

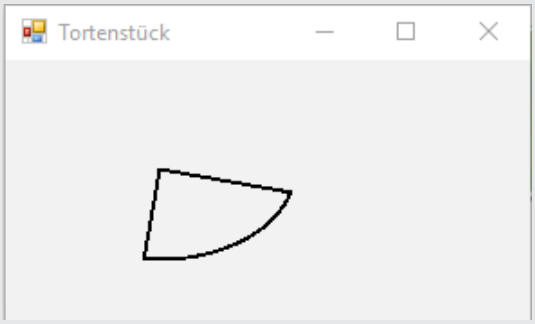
Beispiel 4.35: Tortenstück zeichnen

C#

```
Graphics g = e.Graphics;     Rectangle rec = new Rectangle(10, 10, 150, 100);
g.DrawPie(new Pen(Color.Black, 2), rec, 10, 90);
```

Ergebnis

Zeichnet folgendes Tortenstück:



Kuchendiagramme

Da die Funktionen Winkel als Parameter erwarten, gestaltet sich das Zeichnen von Kuchendiagrammen besonders leicht. Ausgehend von einer einheitlichen *Rectangle*-Struktur brauchen Sie lediglich die Winkelangaben zu variieren.

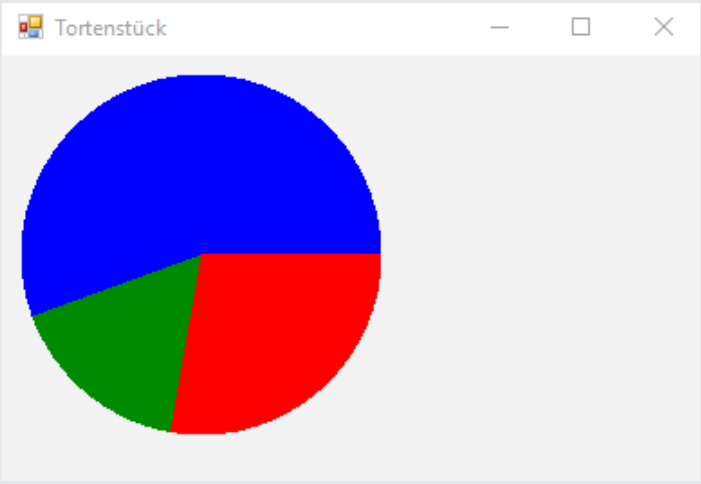
Beispiel 4.36: Zeichnen eines Kuchendiagramms (100°, 60°, 200°)

C#

```
Graphics g = e.Graphics;

Rectangle rec = new Rectangle(10, 10, 200, 200);
g.FillPie(new SolidBrush(Color.Red), rec, 0, 100);
g.FillPie(new SolidBrush(Color.Green), rec, 100, 60);
g.FillPie(new SolidBrush(Color.Blue), rec, 160, 200);
```

Ergebnis

Das Ergebnis kann sich durchaus sehen lassen:

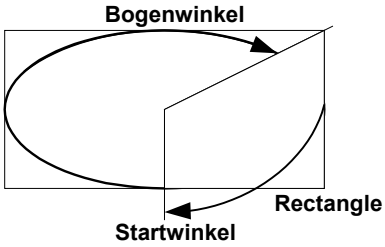


4.4.12 Bogenstück

Die Methode *DrawArc* zeichnet im Gegensatz zur Methode *DrawPie* nur den Bogen, nicht die Verbindungen zum Ellipsenmittelpunkt. Aus diesem Grund kann diese Figur auch nicht gefüllt werden.

Syntax:

```
DrawArc(Pen pen, RectangleF rect, float startwinkel, float bogenwinkel);
```



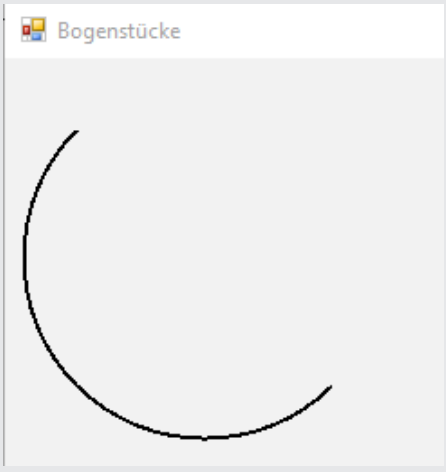
HINWEIS: Positive Winkelangaben werden im Uhrzeigersinn abgetragen.

Beispiel 4.37: Bogenstück zeichnen

C#

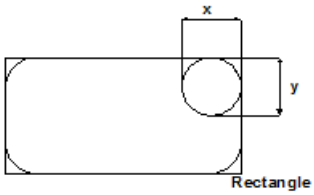
```
Graphics g = e.Graphics;
Rectangle rec = new Rectangle(10, 10, 200, 200);

g.DrawArc(new Pen(Color.Black, 2), rec, 45, 180);
```

Ergebnis



4.4.13 Wo sind die Rechtecke mit den runden Ecken?

Ersatzlos gestrichen! Es bleibt Ihnen also nichts anderes übrig, als sich eine eigene Funktion zu schreiben, oder Sie nutzen die Möglichkeiten von GDI.



Beispiel 4.38: Eine „selbst gestrickte" *RoundRect*-Methode

C#

```
private void RoundRect(Graphics g, Pen p, Rectangle rect, int x, int y)
{
  g.DrawArc(p, new Rectangle(rect.Left, rect.Top, 2 * x, 2 * y), 180, 90);
  g.DrawArc(p, new Rectangle(rect.Left + rect.Width - 2 * x, rect.Top, 2 * x, 2 * y),
            270, 90);
  g.DrawArc(p, new Rectangle(rect.Left + rect.Width - 2 * x,
            rect.Top + rect.Height - 2 * y, 2 * x, 2 * y), 0, 90);
  g.DrawArc(p, new Rectangle(rect.Left, rect.Top + rect.Height - 2 * y,
            2 * x, 2 * y), 90, 90);
  g.DrawLine(p, rect.Left + x, rect.Top, rect.Right - x, rect.Top);
  g.DrawLine(p, rect.Left + x, rect.Bottom, rect.Right - x, rect.Bottom);
  g.DrawLine(p, rect.Left, rect.Top + y, rect.Left, rect.Bottom - y);
  g.DrawLine(p, rect.Right, rect.Top + y, rect.Right, rect.Bottom - y);
}
```

Beim Aufruf der Methode übergeben Sie ein *Graphics*-Objekt, einen *Pen*, eine *Rectangle*-Struktur für das umgebende Rechteck sowie die Breite und Höhe der „Ecken":

```
Graphics g = e.Graphics;
RoundRect(g, new Pen(Color.Black, 3), new Rectangle(10, 10, 250, 150), 20, 10);
```

Ergebnis

Wie Sie sehen können, ist das Ergebnis schon recht brauchbar:

4.4.14 Textausgabe

Neben den grafischen Primitiven wie Kreis oder Linie wird auch Text unter Windows als Grafik ausgegeben. GDI+ bietet dafür die recht universelle Methode *DrawString*. Leider gibt es reichlich Variationen dieser Methode, was es dem Anfänger sicher nicht einfacher macht. Wir beschränken uns auf zwei wichtige Vertreter:

Syntax:

```
void DrawString(string s, Font font, Brush brush, PointF xy,
                StringFormat format);
```

… Ausgabe eines Textes *String* mit der Schrift *Font* und der Füllung *Brush* an der Stelle *XY*. Wenn notwendig, können Sie weitere Formatierungsanweisungen in *StringFormat* übergeben.

Syntax:

```
void DrawString(string s, Font font, Brush brush,
                RectangleF layoutRectangle, StringFormat format);
```

… Ausgabe eines Textes *String* mit der Schrift *Font* und der Füllung *Brush* in einem Rechteck *Rectangle*.

HINWEIS: Mehr Details über das Erstellen von *Font*- bzw. *Brush*-Objekten finden Sie im Abschnitt 4.5.

Beispiel 4.39: Ausgabe eines Textes an der Position 70,70

C#

```
Graphics g = e.Graphics;
g.DrawString("Textausgabe", new Font("Arial", 18),
             new SolidBrush(Color.Black), 70, 70);
```

Texteigenschaften

Haben Sie einen String, wie zum Beispiel

```
s = "Rotationswinkel 27°"
```

… und möchten Sie diesen **zentriert** ausgeben, brauchen Sie Informationen darüber, wie hoch und wie breit der auszugebende Text ist. Die Methode *MeasureString* ist in diesem Zusammenhang interessant.

Für einen vorgegebenen String mit dem gewählten Font werden Breite und Höhe in einer *SizeF*-Struktur zurückgegeben. Während *SizeF.Height* der Schrifthöhe in der jeweils gewählten Skalierung entspricht, ist *SizeF.Width* von der Anzahl der Buchstaben und der Schriftart abhängig:

Die Ausgabeposition berechnet sich wie folgt:

```
Graphics g = e.Graphics;
Font myfont = new Font("Arial", 36);
SizeF mySize = g.MeasureString("Textausgabe", myfont);
g.DrawString("Textausgabe", myfont, new SolidBrush(Color.Black),
            (Width - mySize.Width) / 2, 150);
```

Ausgabe von mehrzeiligem Text

Für die Ausgabe von mehrzeiligem Text nutzen Sie eine überladene Variante von *DrawString*, die zusätzlich eine *Rectangle*-Struktur als Parameter akzeptiert.

HINWEIS: Möchten Sie die Anzahl der entstehenden Zeilen und Spalten ermitteln, können Sie ebenfalls die Methode *MeasureString* nutzen.

Beispiel 4.40: Ausgabe von mehrzeiligem Text

C#

```
Graphics g = e.Graphics;
Font myfont = new Font("Arial", 12);
g.DrawString("Textausgabe im Rechteck", myfont, Brushes.Red,
             new RectangleF(10, 10, 60, 180));
g.DrawRectangle(new Pen(Color.Black), new Rectangle(10, 10, 60, 180));
```

Ergebnis

Textattribute

Über den Parameter *StringFormat* können Sie zusätzliche Formatierungsanweisungen an die *DrawString*-Methode übergeben. Auf die einzelnen Parameter wollen wir an dieser Stelle nicht eingehen, da Sie diese im Normalfall auch kaum gebrauchen werden. Lediglich der Parameter *DirectionVertical* dürfte Ihr Interesse wecken, zu vermuten ist, dass Sie damit einen senkrechten Text ausgeben können.

Ausgabequalität

Wem die Darstellungsqualität des Textes partout nicht passt bzw. wer sich nach einer Antialiasing-Funktion umsieht, wird bei der Eigenschaft *TextRenderingHint* fündig.

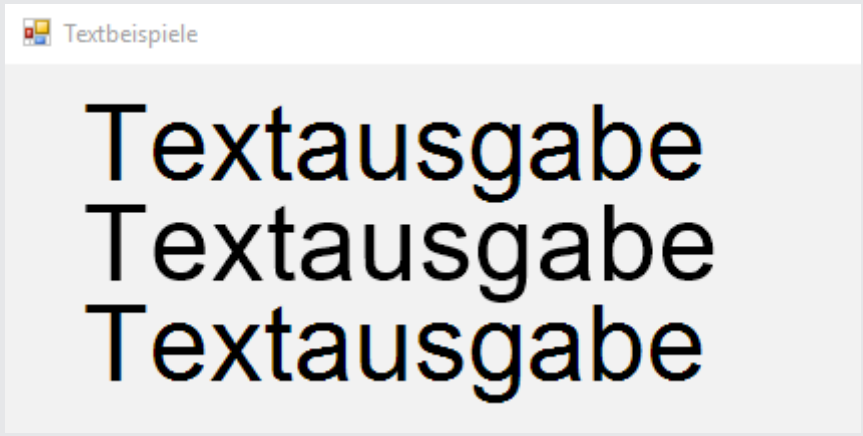
Beispiel 4.41: Antialiasing einschalten

C#

```
Graphics g = e.Graphics;
g.DrawString("Textausgabe", new Font("Arial", 40),
             new SolidBrush(Color.Black), 30, 10);
g.TextRenderingHint = TextRenderingHint.AntiAlias;
g.DrawString("Textausgabe", new Font("Arial", 40),
             new SolidBrush(Color.Black), 30, 60);
g.TextRenderingHint = TextRenderingHint.ClearTypeGridFit;
g.DrawString("Textausgabe", new Font("Arial", 40),
             new SolidBrush(Color.Black), 30, 110);
```

Ergebnis

Das Ergebnis zeigt die folgende Abbildung, die beste Darstellung dürfte der Wert *AntiAlias* liefern.



HINWEIS: Bevor Sie jetzt alle Texte auf diese Weise ausgeben, vergessen Sie bitte nicht, dass dafür auch jede Menge Rechenzeit benötigt wird.

Und wo bleibt eine Methode zum Drehen von Text?

Sie werden auch bei intensivster Suche weder beim *Font*-Objekt noch bei der Methode *DrawString* einen Weg finden, gedrehten Text auszugeben. Wer das Kapitel bis hier aufmerksam gelesen hat, wird sich an die *Graphics*-Methode *RotateTransform* erinnern[3]. Für alle anderen gilt: „Gehe zurück zum Abschnitt 4.3.2“!

HINWEIS: Möchten Sie nacheinander Text mit verschiedenen Winkeln ausgeben, entscheiden Sie mit dem optionalen Parameter *MatrixOrder*, ob die Winkel inkrementell oder absolut angegeben werden.

Beispiel 4.42: Rotation von Text um 15°

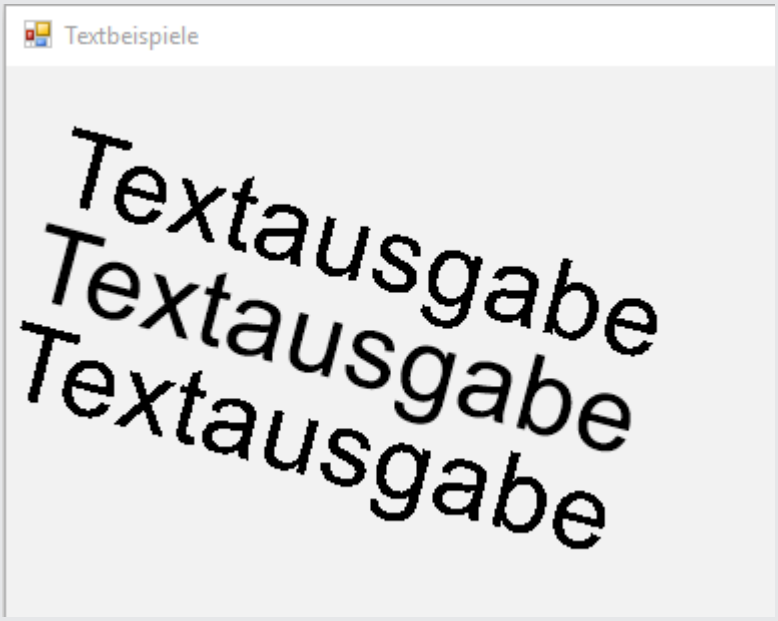
C#

```
Graphics g = e.Graphics;
g.RotateTransform(15);
g.DrawString("Textausgabe", new Font("Arial", 18),
             new SolidBrush(Color.Black), 70, 70);
```

Ergebnis

Kommen wir noch einmal auf unser Beispiel aus dem vorhergehenden Abschnitt zurück.

Testen Sie das Beispiel einmal mit gedrehtem Text, werden Sie folgendes Ergebnis erhalten:



[3] Diese Form der Umsetzung scheint allerdings doch etwas „sinnfrei“ zu sein, berechnen Sie doch bitte einmal schnell die neuen x,y-Koordinaten, wenn Sie einen Text im Winkel von 90° an einem Rechteck ausrichten wollen.

Beispiel 4.43: Absolute Angabe

C#

```
g.RotateTransform(10, System.Drawing.Drawing2D.MatrixOrder.Prepend);
```

Beispiel 4.44: Inkrementelle Angabe

C#

```
g.RotateTransform(10, System.Drawing.Drawing2D.MatrixOrder.Append);
```

4.4.15 Ausgabe von Grafiken

Unter „Ausgabe von Grafiken" möchten wir an dieser Stelle die Wiedergabe von fertigen Grafiken (Bitmaps, Icons, Metafiles) auf einem *Graphics*-Objekt verstehen.

Syntax:

```
void DrawImageUnscaled(Image image, int x, int y);
```

… zeichnet die angegebene Grafik an der durch x und y angegebenen Position.

HINWEIS: Die anderen Varianten von *DrawImageUnscaled* können Sie getrost vergessen, trotz angekündigtem Beschneidens der Grafik erfolgt dies nicht.

Beispiel 4.45: Verwendung von *DrawImageUnscaled*

C#

```
Graphics g = e.Graphics;
g.DrawImageUnscaled(pictureBox1.Image, 0, 0);
```

Skalieren

Für die skalierte Ausgabe von Grafiken können Sie *DrawImage* verwenden:

Syntax:

```
void DrawImage(Image image, int x, int y, int width, int height);
```

Syntax:

```
void DrawImage(Image image, int x, int y, Rectangle srcRect,
GraphicsUnit srcUnit);
```

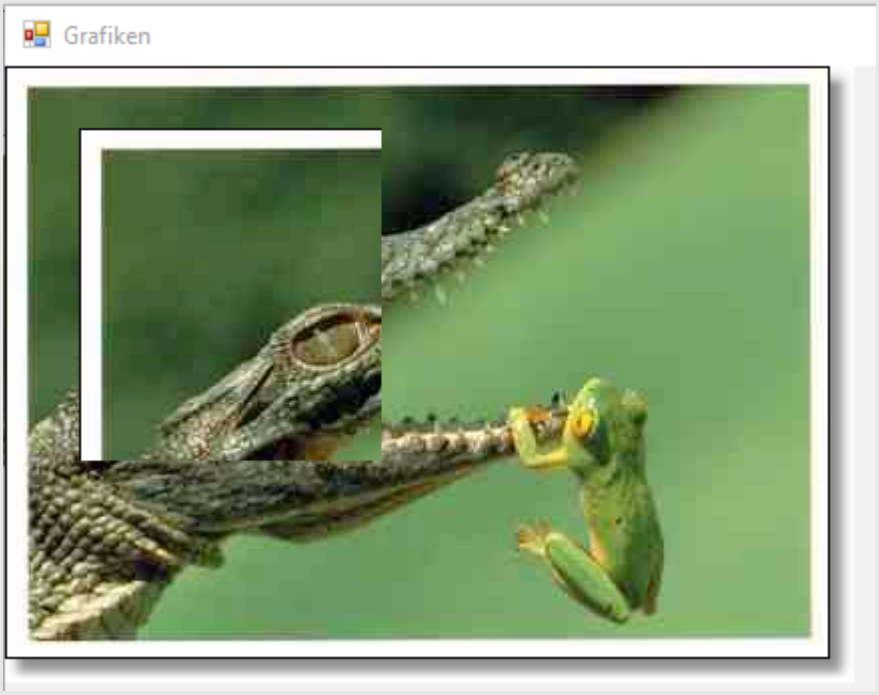
Während die erste Variante neben der Position auch die Breite und Höhe erwartet, können Sie bei der zweiten Syntaxvariante zusätzlich die Einheiten (*GraphicsUnit*) angeben (z. B. beim Drucker).

Beispiel 4.46: Verkleinerte, aber proportionale Ausgabe eines Image

C#

```
Single faktor;
Graphics g = this.CreateGraphics();
faktor = PictureBox1.Image.Height / PictureBox1.Image.Width;
g.DrawImageUnscaled(PictureBox1.Image, 0, 0);
g.DrawImage(PictureBox1.Image, 30, 30, 100, 100 * faktor);
```

Ergebnis



4.5 Unser Werkzeugkasten

Nachdem Sie sich im vorhergehenden Abschnitt mit diversen Grafikmethoden herumgeschlagen haben, wollen wir Ihnen jetzt die einzelnen „Werkzeuge“ des Grafikprogrammierers sowie deren Konfigurationsmöglichkeiten näher vorstellen.

4.5.1 Einfache Objekte

Zunächst sollten wir kurz auf einige grundlegende Strukturen im Zusammenhang mit der Grafikausgabe eingehen.

Point, PointF

Für die Angabe von Koordinaten wird bei vielen Grafikmethoden ein *Point*-Objekt (Integer) bzw. ein *PointF* (Gleitkommawert) verwendet. Die beiden wichtigsten Eigenschaften: x, y.

Beispiel 4.47: Deklaration eines neuen *Point*-Objekts (x = 10, y = 10)

C#

```
Point p = new Point(10, 10);
```

Beispiel 4.48: Direktes Verändern der x-Koordinate

C#

```
Point p = new Point(10, 10);
p.X = 100;
```

Beispiel 4.49: Verschieben des Punkts

C#

```
Point p = new Point(10, 10);
p.Offset(10, 10);
```

Beispiel 4.50: Vergleich von zwei Punktkoordinaten

C#

```
Point p1 = new Point(10, 10);
Point p2 = new Point(10, 10);
if (p1.Equals(p2))
{
    MessageBox.Show("Punkte sind gleich");
}
```

Beispiel 4.51: Konvertieren in einen Gleitkomma-Point (PointF)

C#

```
Point p1 = new Point(10, 10);
PointF p2;
p2 = p1;        // Variante 1 implizit
p2 = (PointF)p1;                 // Variante 2 explizit
```

Size, SizeF

Ähnlich wie *Point* verwaltet *Size* bzw. *SizeF* ein Koordinatenpaar, nur dass es sich in diesem Fall um die Breite (*Width*) bzw. Höhe (*Height*) handelt.

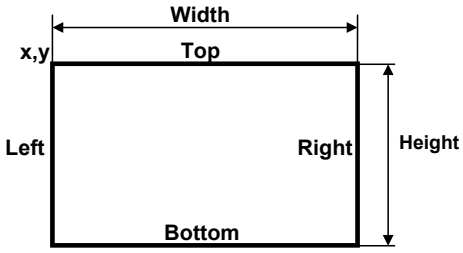
Beispiel 4.52: Deklarieren und Verändern der Breite

C#

```
Size mysize = new Size(100, 80);
mysize.Width = 130;
```

Rectangle, FRectangle

Das dritte Grundobjekt ist im Grunde die Kombination der beiden vorhergehenden Objekte. Die folgende Skizze zeigt die verschiedenen Formen des Zugriffs auf die Koordinatenpaare:



Beispiel 4.53: Deklarieren und Verwenden von *Rectangle*

C#

```
Graphics g = e.Graphics;
Rectangle rec = new Rectangle(10, 10, 200, 100);
g.FillRectangle(new SolidBrush(Color.Red), rec);
```

4.5.2 Vordefinierte Objekte

Vielleicht hatte irgendein Programmierer bei Microsoft ein Einsehen und entschied sich dafür, wenigstens einige Grafikobjekte vorzudefinieren, das heißt, ohne dass Sie diese erst mit viel Aufwand und umfangreichen Parameterlisten initialisieren müssen.

Vier Gruppen können Sie unterscheiden:

- vordefinierte Pinsel (Brushes),
- vordefinierte Stifte (Pens),
- vordefinierte Farben (Colors),
- vordefinierte Grafiken (Icons).

Vordefinierte Pinsel

Neben *Brushes* (alle relevanten Farben) können Sie auch *SystemBrushes* (die vordefinierten Systemfarben) verwenden.

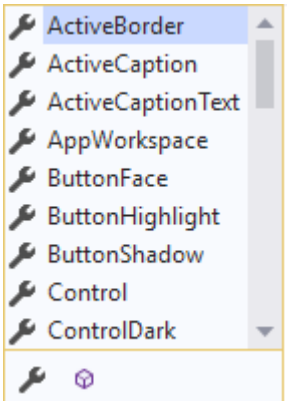
Beispiel 4.54: Erzeugen einer leicht gelb getönten Fläche (Farbe der Hints)

C#

```
Graphics g = e.Graphics;
Rectangle rec = new Rectangle(300, 10, 200, 100);
g.FillRectangle(SystemBrushes.Info, rec);
```

Vordefinierte Stifte

Über das *SystemPens*-Objekt bzw. dessen Eigenschaften rufen Sie vordefinierte Stifte (Breite = 1 Pixel) für alle Windows-Farben ab:



Beispiel 4.55: Verwendung eines System-Pens

C#

```
Graphics g = e.Graphics;
g.DrawRectangle(SystemPens.ControlText, 500, 10, 100, 100);
```

Vordefinierte Farben

Farben können Sie entweder über *SystemColors* (die Farben der Windows-Elemente) oder über *Color* abrufen.

Beispiel 4.56: Verwendung von *Color*

C#

```
Graphics g = e.Graphics;
Rectangle rec = new Rectangle(600, 10, 200, 200);
g.FillPie(new SolidBrush(Color.Red), rec, 0, 100);
```

Vordefinierte Icons

Über *SystemIcons* können Sie die wichtigsten System-Icons direkt abrufen[4]:

Eigenschaft
Application
Asterisk
Error
Exclamation
Hand
Information
Question
Warning
WinLogo

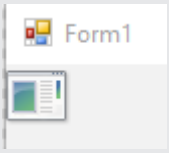
Beispiel 4.57: Verwendung eines System-Icons

C#

```
Graphics g = e.Graphics;
g.DrawIcon(SystemIcons.Application, 0, 0);
```

Ergebnis

Die Grafikausgabe:



4.5.3 Farben/Transparenz

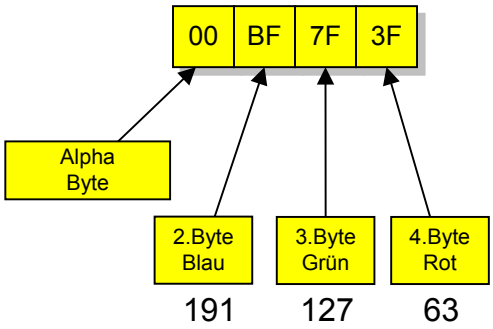
Farben setzen sich auch unter Windows aus der additiven Überlagerung der drei Grundfarben Rot, Grün und Blau zusammen (RGB). Da jeder Farbanteil in 256 Farbstufen unterteilt ist, ergibt sich eine maximale Anzahl von ca. 16 Mio. Farben. Bei dieser Auflösung ist das menschliche Auge nicht mehr in der Lage, einzelne Abstufungen wahrzunehmen, man spricht von Echtfarben.

[4] Diese unterscheiden sich etwas je nach System, deshalb hier keine Abbildungen.

ARGB-Farben

GDI+ verwendet für die Verwaltung von Farbinformationen sogenannte ARGB-Werte, das heißt 4 Byte- bzw. Integer-Werte. Auch die bereits vordefinierten Farbwerte (siehe vorhergehender Abschnitt) verwenden dieses Farbmodell.

Um zu verstehen, wie die Farbwerte gespeichert werden können, müssen wir wissen, dass zur Darstellung **eines** Bytes **zwei** Hexziffern benötigt werden. Eine einzelne Hexziffer wird durch eines der 16 Zeichen 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F dargestellt. Die drei niederwertigen Bytes geben die RGB-Farbintensität für Blau, Grün und Rot an.



Beispiel 4.58: Farbwerte

C#

Da pro Byte 256 Werte gespeichert werden können, entspricht der Wert 0x00FF0000 einem reinen Blau mit voller Intensität, der Wert 0x0000FF00 einem reinen Grün und der Wert 0x000000FF einem reinen Rot. 0x00000000 gibt Schwarz und 0x00FFFFFF Weiß an. In obiger Abbildung wird eine graublaue Farbe definiert.

HINWEIS: Möchten Sie einen Graustufenwert erzeugen, müssen die einzelnen Farbanteile jeweils den gleichen Wert aufweisen.

Was ist mit dem höchstwertigen Byte?

Bei ARGB-Werten wird in diesem Byte die Information über die Transparenz dieser Farbe gespeichert. Ein Wert von 255 entspricht der vollen Deckkraft, 0 entspricht vollständiger Transparenz. Mithilfe der Methode *FromArgb* können Sie einen beliebigen Farbwert mit der gewünschten Transparenz erzeugen.

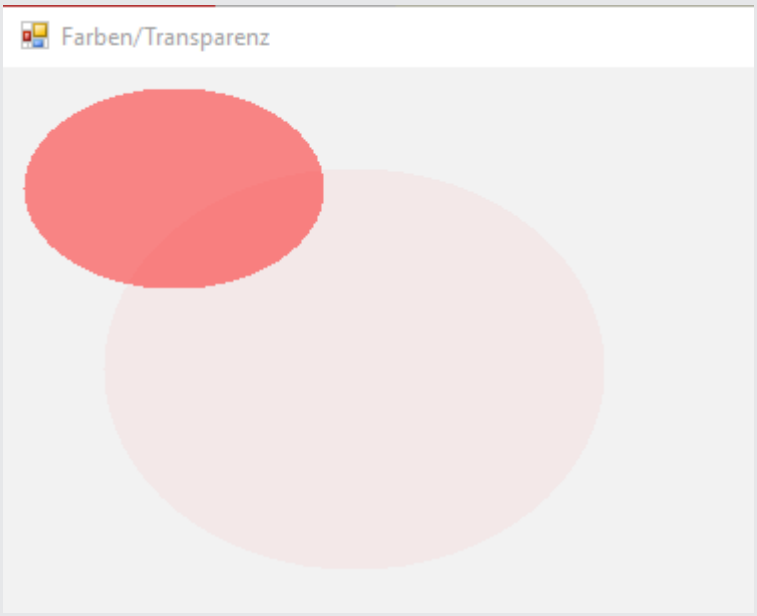
Beispiel 4.59: Erzeugen von zwei teiltransparenten Ellipsen

C#

```
Graphics g  = e.Graphics;

Color c1 = Color.FromArgb(125, Color.Red);
g.FillEllipse(new SolidBrush(c1), new Rectangle(10, 10, 150, 100));
Color c2 = Color.FromArgb(12, Color.Red);
g.FillEllipse(new SolidBrush(c2), new Rectangle(50, 50, 250, 200));
```

Ergebnis

Das Ergebnis auf dem Formular:



Möchten Sie zwei Farbwerte vergleichen, verwenden Sie die Methode *ToArgb*:

```
if (c1.ToArgb() == c2.ToArgb())
{
    MessageBox.Show("Beide Farben sind gleich!");
}
```

4.5.4 Stifte (Pen)

Für Grafikmethoden, die Linien verwenden, benötigen Sie ein initialisiertes *Pen*-Objekt. Dieses enthält unter anderem Informationen über:

- die Farbe,
- die Dicke,
- die Linienenden,

- die Verbindung zweier zusammenhängender Linien,
- den Füllstil.

Bevor wir Sie mit einer Fülle von Eigenschaften erschlagen, sollen einige Beispiele für mehr Klarheit sorgen.

Einfarbige Stifte

Einfarbige Stifte werden mit einer der vordefinierten Farben oder einem ARGB-Wert als Parameter erzeugt.

Beispiel 4.60: Erzeugen eines roten Stifts

C#

```
Pen pen = new Pen(Color.Red);
```

Beispiel 4.61: Erzeugen eines einfarbigen Pens mit 50 % Transparenz und 10 Pixeln Breite

C#

```
Pen pen = new Pen(Color.FromArgb(128, 17, 69, 137), 10);
```

Beispiel 4.62: Zeichnen von zwei aufeinanderfolgenden roten Linien (15 Pixel breit) mit einem Pfeil am Anfang und einem runden Ende

C#

```
using System.Drawing.Drawing2D;
...
Graphics g  = e.Graphics;
```

Zunächst einen einfarbigen *Pen* der Stärke 15 erzeugen:

```
Pen pen = new Pen(Color.Red, 15);
```

Die Punkte für die Linienenden festlegen:

```
PointF[] punkte = new PointF[] {new PointF(10, 10), new PointF(100, 100),
                                new PointF(50, 150)};
```

Den Linienstart festlegen (Pfeil):

```
  pen.EndCap = LineCap.Round;
```

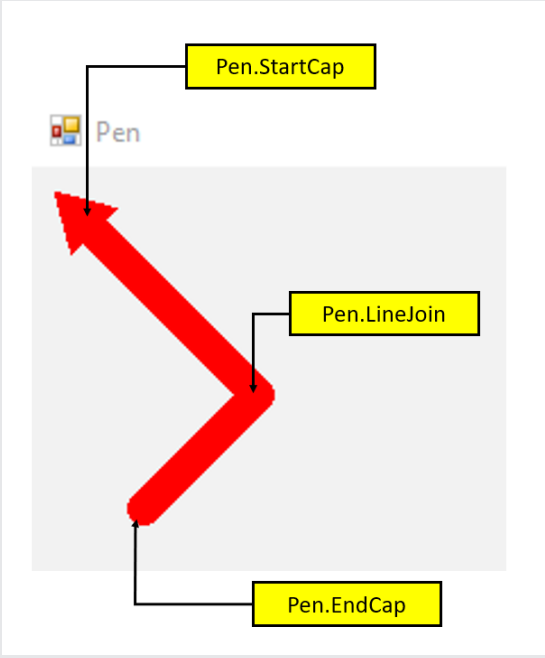
Das Linienende festlegen (rund):

```
  pen.StartCap = LineCap.ArrowAnchor;
```

Die Verbindung der zwei Teillinien festlegen (rund) und Linien zeichnen:

```
  pen.LineJoin = LineJoin.Round;
  g.DrawLines(pen, punkte);
```

Ergebnis



Einige wichtige Linieneigenschaften auf einen Blick:

Eigenschaft	Beschreibung
Alignment	… beschreibt die Position (*Center, Inset, Outset, Left, Right*) der Linie bezüglich der gedachten Ideallinie zwischen zwei Punkten 
Color	… die Füllfarbe des Stifts
LineJoin	… beschreibt den Übergang (*Bevel, Miter, MiterClipped, Round*) zwischen zwei aufeinanderfolgenden Linien. Siehe dazu vorhergehendes Beispiel.
StartCap, EndCap	… beschreibt die Form der Linienenden (*Flat, Round, Square …*). Siehe dazu vorhergehendes Beispiel.
DashCap	… beschreibt die Form der einzelnen Punkte/Linienabschnitte bei gestrichelten Linien (*Flat, Round Triangle*)
DashStyle	… beschreibt die Form von gestrichelten Linien (*Custom, Dash, DashDot, DashDotDot, Dot, Solid*)
PenType	… beschreibt die Stiftart (*SolidColor, Hatchfill, LinearGradient, PathGradient, TextureFill*). Diese Eigenschaft ist schreibgeschützt.

Stifte mit Füllung

Im Unterschied zu den bisher vorgestellten Stiften, die lediglich unterschiedliche Linienmuster bzw. Linienfarben aufweisen konnten, lassen sich auch Stifte erzeugen, die mit einem *Brush* statt mit einer Farbe initialisiert werden.

Beispiel 4.63: Erzeugen eines Stifts mit Brush als Füllung

C#

```
Graphics g = e.Graphics;
Pen pen = new Pen(Brushes.Azure, 10);
g.DrawLine(pen, 10, 10, 100, 100);
```

Auf den ersten Blick werden Sie keinen Unterschied zu den bisherigen Pens erkennen, da wir einen *SolidBrush* verwendet haben.

Etwas anders sieht die Linie allerdings aus, wenn Sie zum Beispiel einen *HatchBrush* (siehe auch folgender Abschnitt) für die Linienfüllung verwenden.

Beispiel 4.64: *HatchBrush* für Linie erzeugen

C#

Vergessen Sie nicht, den entsprechenden Namespace einzubinden:

```
using System.Drawing.Drawing2D;
...
Graphics g = e.Graphics;
HatchBrush brush = new HatchBrush(HatchStyle.SolidDiamond,
           Color.Black, Color.Yellow);
Pen pen = new Pen(brush, 20);
g.DrawLine(pen, 10, 10, 200, 100);
```

Ergebnis

Das Ergebnis unterscheidet sich doch wesentlich von den bisher bekannten Linienarten:

Wie Sie sehen, können Sie fast beliebige Stiftfüllungen erzeugen, Sie müssen nur einen entsprechenden Brush (siehe folgender Abschnitt) zur Verfügung stellen.

HINWEIS: Verwechseln Sie nicht Stifte mit Pinseln. Mit Stiften können Sie niemals eine Figur füllen, mit Pinseln können Sie keine Figuren zeichnen, sondern nur füllen.

4.5.5 Pinsel (Brush)

Damit sind wir bei einem der wohl komplexesten Zeichenobjekte angelangt. Neben dem bereits mehrfach in diesem Kapitel verwendeten *SolidBrush* und dem im vorhergehenden Abschnitt bereits angesprochenen *HatchBrush* finden Sie noch weitere drei Vertreter dieser Gattung.

Alle Pinsel auf einen Blick:

Typ	Beschreibung
SolidBrush	… ein Pinsel mit einheitlicher Farbe und ohne Muster
HatchBrush	… ein Pinsel mit einem Linienmuster sowie Vordergrund- (Linie) und Hintergrundfarbe
TextureBrush	… ein Pinsel mit Images/Bitmaps als Füllmuster
LinearGradientBrush	… ein Pinsel mit einem Farbverlauf als Füllung (Verlauf zwischen zwei Farben)
PathGradientBrush	… ein Pinsel mit mehreren Farbverläufen (Verlauf zwischen einer Farbe und mehreren anderen)

Gefüllte Objekte, basierend auf den oben genannten Pinseln, erzeugen Sie mit den entsprechenden *Fill*xyz-Methoden (z. B. *FillRectangle*) des *Graphics*-Objekts.

4.5.6 SolidBrush

Das Erzeugen eines *SolidBrush* dürfte Sie kaum vor Probleme stellen. Übergeben Sie eine Farbe oder verwenden Sie die Methode *FromArgb*, um eine neue Farbe zu definieren.

Beispiel 4.65: Roter Pinsel

C#

```
SolidBrush brush = new SolidBrush(Color.Red);
```

Beispiel 4.66: Teilweise transparenter Pinsel mit selbst definierter Farbe

C#

```
SolidBrush brush = new SolidBrush(Color.FromArgb(123, 10, 17, 36));
```

4.5.7 HatchBrush

Bevor Sie sich näher mit dieser Pinselvariante beschäftigen können, müssen Sie den notwendigen Namespace *System.Drawing.Drawing2D* einbinden.

Ein *HatchBrush* besteht aus einem Vordergrundmuster mit eigener Farbe und einer Hintergrundfarbe. Der Konstruktor:

Syntax:

```
HatchBrush(HatchStyle hatchstyle, Color foreColor, Color backColor);
```

Übergeben können Sie unter anderem folgende Konstanten:

HatchStyle
BackwardDiagonal
Cross
DarkDownwardDiagonal
DarkHorizontal
DarkUpwardDiagonal
DarkVertical
DashedDownwardDiagonal
DashedHorizontal
DashedUpwardDiagonal
DashedVertical
DiagonalBrick
DiagonalCross
Divot
DottedDiamond
DottedGrid
ForwardDiagonal

Beispiel 4.67: Erzeugen und Verwenden einer *HatchBrush*

C#

```
Graphics g = e.Graphics;
HatchBrush brush = new HatchBrush(HatchStyle.SolidDiamond,
                      Color.Red, Color.White);
g.FillRectangle(brush, 10, 10, 200, 100);
```

Ergebnis

Das erzeugte Rechteck:

HINWEIS: Eine Skalierung des Ausgabegeräts (*ScaleTransform*) hat keinen Einfluss auf die Mustergröße.

4.5.8 TextureBrush

Mit einem *TextureBrush* lassen sich Flächen mit Bitmap-Mustern füllen. Ist die Bitmap zu klein, um die gesamte Fläche auszufüllen, wird diese wiederholt dargestellt.

Beispiel 4.68: Basierend auf den System-Icons soll ein Rechteck mit Error-Symbolen gefüllt werden.

C#

```
Graphics g = e.Graphics;
TextureBrush textureBrush = new TextureBrush(SystemIcons.Error.ToBitmap());
g.FillRectangle(texttureBrush, 250, 10, 200, 100);
```

Ergebnis

Die Ausgabe:

Natürlich stellt es auch kein Problem dar, die Grafik aus einer Datei zu laden.

Beispiel 4.69: Laden der Datei *Bitmap1.bmp* als *TextureBrush*

C#

```
Graphics g = e.Graphics;
Image img  = new Bitmap("bitmap1.bmp");

TextureBrush textureBrush = new TextureBrush(img);
g.FillRectangle(textureBrush, 500, 10, 200, 100);
```

4.5.9 LinearGradientBrush

Mit dem *LinearGradientBrush* bringen Sie im wahrsten Sinne des Wortes mehr Farbe in Ihre Anwendungen. Das Grundprinzip: Sie geben zwei Farben und eine Richtung (daher das „linear“) an und GDI+ berechnet Ihnen den zugehörigen Farbverlauf.

An den Konstruktor müssen Sie folgende Werte übergeben:

Syntax:

```
LinearGradientBrush(Rectangle rect, Color startfarbe, Color endfarbe,
                    LinearGradientMode linearGradientMode);
```

Rectangle gibt ein Rechteck an (es sind auch zwei *Point*-Werte zulässig), in dem der Farbverlauf berechnet wird. Die Betonung liegt auf „berechnet“. Welche Ausgabefläche Sie später mit dem neuen *Brush* füllen, ist eine ganz andere Frage.

Start- und Endfarbe sind normale ARGB-Color-Werte mit Transparenzangabe. Das heißt, wenn Sie beispielsweise zwei gleiche Farben, aber unterschiedliche Alpha-Werte angeben, können Sie einen Farbverlauf mit zu- bzw. abnehmender Transparenz realisieren.

Folgende *LinearGradientMode*-Werte sind zulässig:

Konstante	Beschreibung
BackwardDiagonal	Farbverlauf von rechts oben nach links unten
ForwardDiagonal	Farbverlauf von links oben nach rechts unten
Horizontal	Farbverlauf von links nach rechts
Vertical	Farbverlauf von oben nach unten

Beispiel 4.70: Ein vertikaler Farbverlauf von Rot nach Gelb soll realisiert werden.

C#

```
Graphics g = e.Graphics;
Rectangle rect = new Rectangle(10, 10, 200, 135);
LinearGradientBrush myBrush = new LinearGradientBrush(rect, Color.Red,
                        Color.Yellow, LinearGradientMode.Vertical);
g.FillRectangle(myBrush, new Rectangle(10, 10, 200, 185));
```

Ergebnis

Wer ganz genau hinsieht, wird allerdings schnell „ein Haar in der Suppe“ finden. Die erste Zeile des Ausgaberechtecks ist nicht rot, sondern gelb. Die Ursache ist vermutlich die Wiederholung des Farbverlaufs, wenn der berechnete Verlauf kleiner als die Ausgabefläche ist.

4.5.10 PathGradientBrush

Hier haben wir es mit einer Spezialform von Gradienten zu tun, die es uns erlauben, recht eindrucksvolle Farbeffekte zu realisieren. Ausgehend von einer zentralen Farbe können Sie innerhalb eines *Path*-Objekts (siehe dazu Abschnitt 4.5.12) mehrere Farbverläufe zu verschiedenen Farben realisieren.

Beispiel 4.71: Erzeugen und Verwenden eines *PathGradientBrush*

C#

```
Graphics g = e.Graphics;
```

Zunächst erzeugen wir das *Path*-Objekt:

```
GraphicsPath path = new GraphicsPath();
path.AddLine(10, 10, 300, 10);
path.AddLine(300, 10, 250, 200);
path.AddLine(250, 200, 150, 250);
path.AddLine(150, 250, 50, 200);
```

Jetzt können wir mit diesem *Path* den *PathGradientBrush* erzeugen und konfigurieren:

```
PathGradientBrush brush = new PathGradientBrush(path);
```

Ein Array mit den Farben für die jeweiligen Eckpunkte im *Path*:

```
Color[] colors = new Color[] {Color.Yellow, Color.Green,
                  Color.Red, Color.Cyan, Color.Blue};
```

Die Farbe im Mittelpunkt[5] und an den Ecken des *Path*:

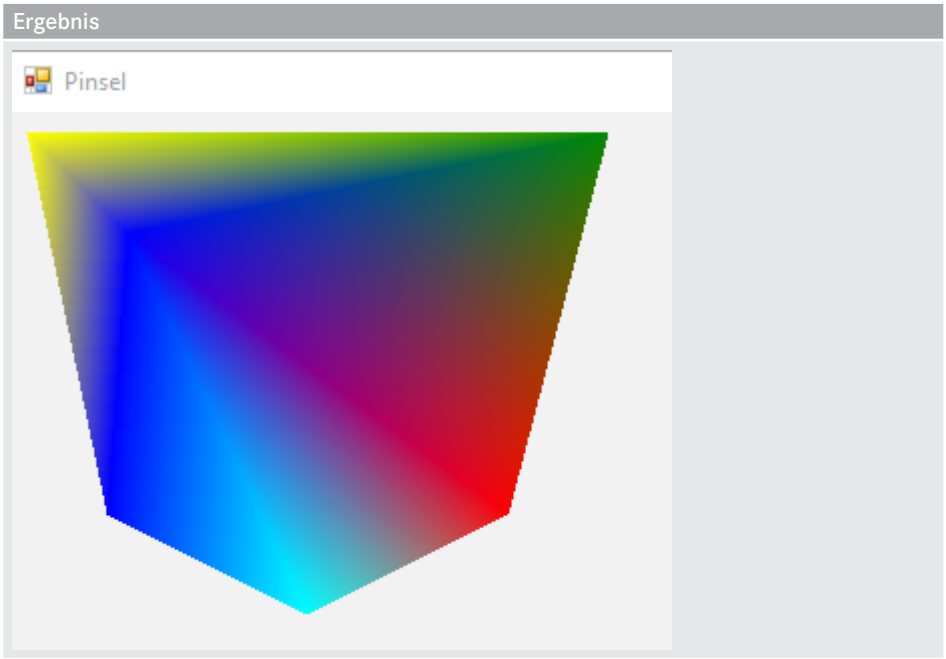
```
brush.CenterColor = Color.Blue;
brush.SurroundColors = colors;
```

Wir geben den *Path* aus:

```
brush.CenterPoint = new PointF(60, 60);
g.FillPath(brush, path);
```

[5] „Mittelpunkt“ sollten Sie nicht zu genau nehmen, über die Eigenschaft *CenterPoint* können Sie den Mittelpunkt selbst definieren.

Ergebnis



4.5.11 Fonts

Wie bei Stiften und Pinseln handelt es sich auch bei Schriften (Fonts) um Objekte, die Sie zunächst erzeugen müssen. Sage und schreibe dreizehn verschiedene Konstruktorüberladungen werden Ihnen angeboten, wir belassen es bei den beiden folgenden Varianten:

Syntax:

```
Font(string familyName, float emSize);
```

Syntax:

```
Font(string familyName, float emSize, FontStyle style);
```

Was mit *familyName* und *emSize* gemeint ist, dürfte klar sein. Über *style* können Sie eine Kombination der folgenden *FontStyle*-Werte zuweisen:

- *Bold* (fett)
- *Italic* (kursiv)
- *Regular* (normal)
- *Strikeout* (durchgestrichen)
- *Underline* (unterstrichen)

Kombinieren können Sie die Werte durch die Verknüpfung mit dem OR-Operator.

HINWEIS: Die Farbe bzw. die Füllung der Schriftart legen Sie erst bei der Ausgabe mittels *DrawString* über einen entsprechenden *Brush* fest. Damit können Sie auch Farbverläufe oder Transparenzeffekte bei Fonts erreichen.

Genug der Theorie, ein praktisches Beispiel soll die Verwendung zeigen.

Beispiel 4.72: Erzeugen und Verwenden eines neuen Font-Objekts

C#

```
Graphics g = e.Graphics;
Font f = new Font("Arial", 24, FontStyle.Italic | FontStyle.Bold);
g.DrawString("Test Schriftarten", f, new SolidBrush(Color.Blue), 10, 10);
```

Ergebnis

Ein Blick auf die Eigenschaften des *Font*-Objekts macht uns neugierig: Mit *Bold, Italic, Size* etc. stehen uns alle Möglichkeiten zur Konfiguration der Schriftart zur Verfügung. Doch ach: Alle wünschenswerten Eigenschaften sind schreibgeschützt, Sie müssen also wohl oder übel eine neue Schriftart erzeugen.

4.5.12 Path-Objekt

Path-Objekte fassen mehrere Zeichenoperationen (*DrawLine, DrawString, DrawEllipse* …) bzw. mehrere grafische Primitive (Linien, Kreise etc.) quasi in einem Objekt zusammen. Sie können *Path*s am besten mit der Gruppieren-Funktion eines Grafikprogramms vergleichen. Ähnlich wie mit der Gruppe können Sie den kompletten *Path* mit einer Anweisung auf einem *Graphics*-Objekt wiedergeben (*DrawPath* oder *FillPath*). Welche Linientypen bzw. welcher Füllstil benutzt wird, entscheiden Sie erst beim Zeichnen auf dem *Graphics*-Objekt, nicht beim Erstellen des Objekts.

Beispiel 4.73: Erzeugen eines einfachen *Path* (nur zwei Ellipsen und zwei Linien) und Wiedergabe auf dem *Form*-Objekt

C#

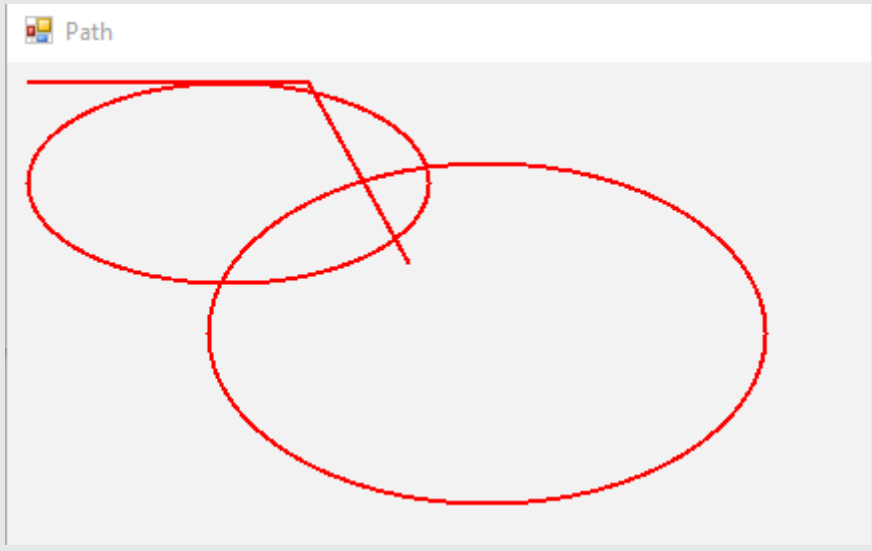
```
using System.Drawing.Drawing2D;
...
   Graphics g = e.Graphics;
   GraphicsPath path = new GraphicsPath();
```

```
path.AddEllipse(10, 10, 200, 100);
path.AddEllipse(100, 50, 278, 170);
path.AddLine(10, 10, 150, 10);
path.AddLine(150, 10, 200, 100);
```

Wie Sie sehen, wird erst bei der Wiedergabe des *Path*-Objekts ein entsprechender *Pen* zugeordnet:

```
g.DrawPath(new Pen(Color.Red, 2), path);
```

Ergebnis



Füllen

Sie können ein beliebig zusammengesetztes *Path*-Objekt mit einem Pinsel Ihrer Wahl füllen lassen. Ein Beispiel hatten Sie ja bereits in Abschnitt 4.5.10 (PathGradientBrush) kennengelernt. Das Grundprinzip ist immer gleich. Sie erzeugen ein *Path*-Objekt, fügen diesem die gewünschten Grafikanweisungen hinzu und geben dann diesen *Path* auf einem *Graphics*-Objekt aus.

Beispiel 4.74: Füllen mit einem Bitmap-Muster

C#

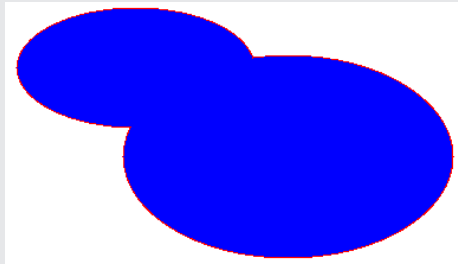
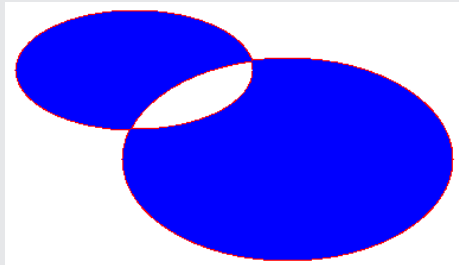
```
Graphics g = e.Graphics;
GraphicsPath path = new GraphicsPath();
TextureBrush brush = new TextureBrush(SystemIcons.Error.ToBitmap());
path.AddEllipse(10, 10, 200, 100);
path.AddEllipse(100, 50, 278, 170);
g.FillPath(brush, path);
```

Ergebnis

Fillmode

In diesem Zusammenhang ist Ihnen sicher aufgefallen, dass der sich überschneidende Bereich beider Ellipsen nicht gefüllt worden ist.

Wie überlappende bzw. sich schneidende Objekte gefüllt werden, bestimmen Sie mit der Eigenschaft *FillMode*. Statt vieler Worte über die Funktionsweise dürften die beiden folgenden Abbildungen mehr über die Funktionsweise aussagen:

FillMode = FillMode.Winding	*FillMode = FillMode.Alternate*
	

Sehen wir uns mit diesem Wissen noch einmal unser Einstiegsbeispiel an, verwenden jedoch statt der *DrawPath*- die *FillPath*-Methode.

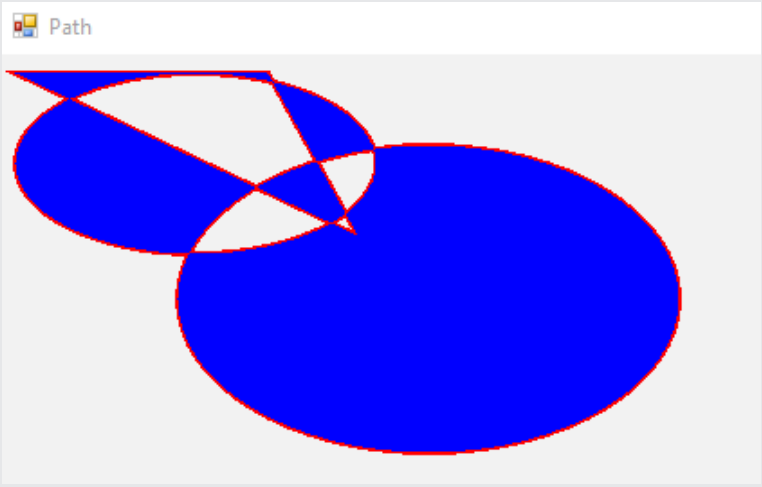
Beispiel 4.75: Füllen eines Paths

C#

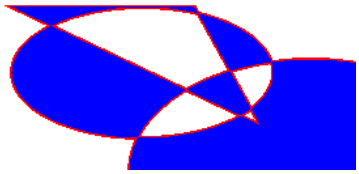
```
Graphics g = e.Graphics;
GraphicsPath path = new GraphicsPath();
path.AddEllipse(10, 10, 200, 100);
path.AddEllipse(100, 50, 278, 170);
path.AddLine(10, 10, 150, 10);
path.AddLine(150, 10, 200, 100);
path.CloseAllFigures();
g.DrawPath(new Pen(Color.Red, 3), path);
g.FillPath(Brushes.Blue, path);
```

Ergebnis

Im Ergebnis werden Sie feststellen, dass beim Füllen auch die beiden Linien eine Rolle spielen.



Zwischen den beiden Endpunkten der Linien wird zwar keine Linie gezeichnet, eine Füllung kommt unter Berücksichtigung von *Fillmode* jedoch zustande. Soll aus den beiden Linien eine in sich geschlossene Fläche erzeugt werden, rufen Sie die Methode *CloseAllFigures* auf, die die beiden Linien zu einem Dreieck verbindet (achten Sie auf die neue Verbindungslinie zwischen den beiden Linienendpunkten):



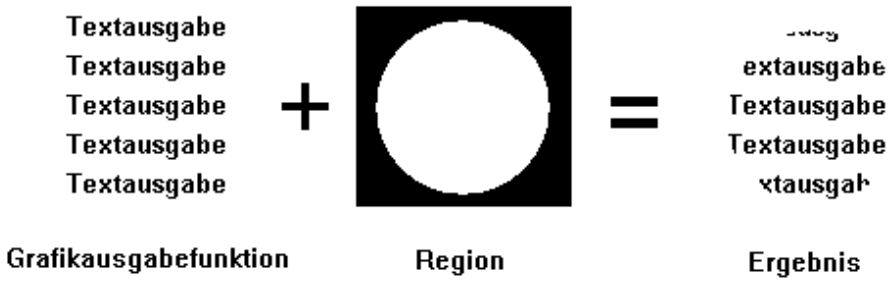
HINWEIS: Ein weiteres Einsatzgebiet für *Path*-Objekte findet sich im Zusammenhang mit dem Clipping, das wir im folgenden Abschnitt näher betrachten wollen.

4.5.13 Clipping/Region

In Ihren Programmen sind Sie es gewohnt, dass Zeichenoperationen, die über den Clientbereich des Formulars bzw. der Komponente hinausgehen, einfach abgeschnitten werden. Damit haben Sie auch schon die einfachste Form von Clipping kennengelernt.

Für bestimmte visuelle Effekte ist es häufig wünschenswert, dass diese Ausgabebereiche nicht nur rechteckig, sondern zum Beispiel auch mal rund oder aus verschiedenen grafischen Primitiven zusammengesetzt sind.

Die Abbildung zeigt das Ergebnis von Grafikausgaben in einem runden Clipping-Bereich:



Ein Clipping-Bereich ist immer einem *Graphics*-Objekt zugewiesen, mithilfe der Methode *SetClip* können Sie diesen Bereich verändern.

Unter GDI+ bieten sich mehrere Varianten an:

- Übernahme von anderem *Graphics*-Objekt (*Graphics.SetClip(Graphics)*),
- Clipping in einem Rechteck (*Graphics.SetClip(Rectangle)*),
- Clipping in einem *Path* (*Graphics.SetClip(GraphicsPath)*),
- Clipping in einer *Region* (*Graphics.SetClip(Region)*).

Im Folgenden beschränken wir uns auf die letzte Variante, die Vorgehensweise ist auch bei einem *Path* oder einem Rechteck immer gleich.

Regions

Regions können, im Zusammenhang mit dem Clipping, quasi wie eine Schablone betrachtet werden. Da Sie *Regions* recht einfach zusammensetzen können (mit unterschiedlichen Verknüpfungen), dürften sie die erste Wahl für die meisten Clipping-Aufgaben sein.

Das Erzeugen einer Region ist relativ simpel. Über den Konstruktor des *Region*-Objekts können Sie bereits entscheiden, worauf die Region aufgebaut wird:

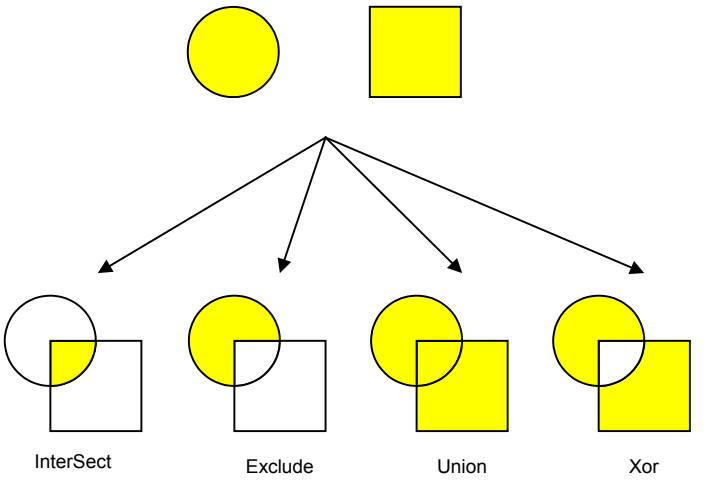
- *ein Path*-Objekt,
- ein Rechteck,
- ein anderes *Region*-Objekt.

Beispiel 4.76: Erzeugen und Füllen einer rechteckigen Region

C#

```
using System.Drawing.Drawing2D;
...
 Graphics g = e.Graphics;
 SolidBrush blueBrush = new SolidBrush(Color.Blue);
 Region reg1 = new Region(new Rectangle(100, 100, 200, 200));
 g.FillRegion(blueBrush, reg1);
```

Zwei Regionen können Sie mithilfe der *Region*-Methoden auch kombinieren:



Beispiel 4.77: Kombination zweier rechteckiger Regionen

C#

```
  Graphics g = e.Graphics;

  SolidBrush blueBrush = new SolidBrush(Color.Blue);
  Region reg1 = new Region(new Rectangle(100, 100, 200, 200));
  Region reg2 = new Region(new Rectangle(50, 50, 150, 150));
```

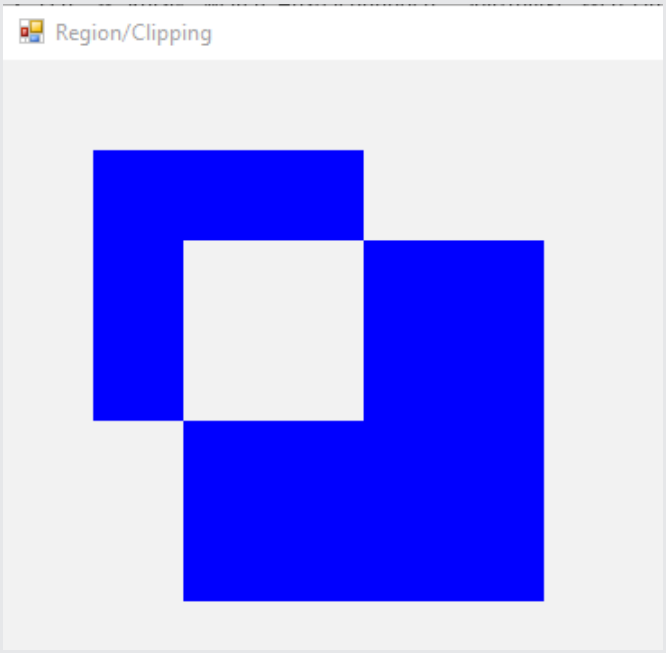
Region2 wird mit Region1 XOR-verknüpft:

```
  reg2.Xor(reg1);
```

Die Region füllen:

```
  g.FillRegion(blueBrush, reg2);
```

Ergebnis



Clipping

Nach diesem Kurzeinstieg in die Arbeit mit Regionen können wir uns wieder dem Clipping zuwenden. Nutzen Sie Regionen oder Paths, um den Ausgabebereich von Grafikoperationen (Füllen, Zeichnen) in einem *Graphics*-Objekt zu beschränken. Alle über den Clippingbereich hinausgehenden Zeichenoperationen werden abgeschnitten.

Beispiel 4.78: Erzeugen eines Clipping-Bereichs, der aus zwei Regionen besteht

C#

```
using System.Drawing.Drawing2D;
...
  Graphics g = e.Graphics;
```

Zwei Regionen erzeugen:

```
    Region reg1 = new Region(new Rectangle(100, 100, 200, 200));
    Region reg2 = new Region(new Rectangle(50, 50, 150, 150));
```

Verbinden beider Regionen:

```
    reg2.Union(reg1);
```

Clipping-Bereich festlegen:

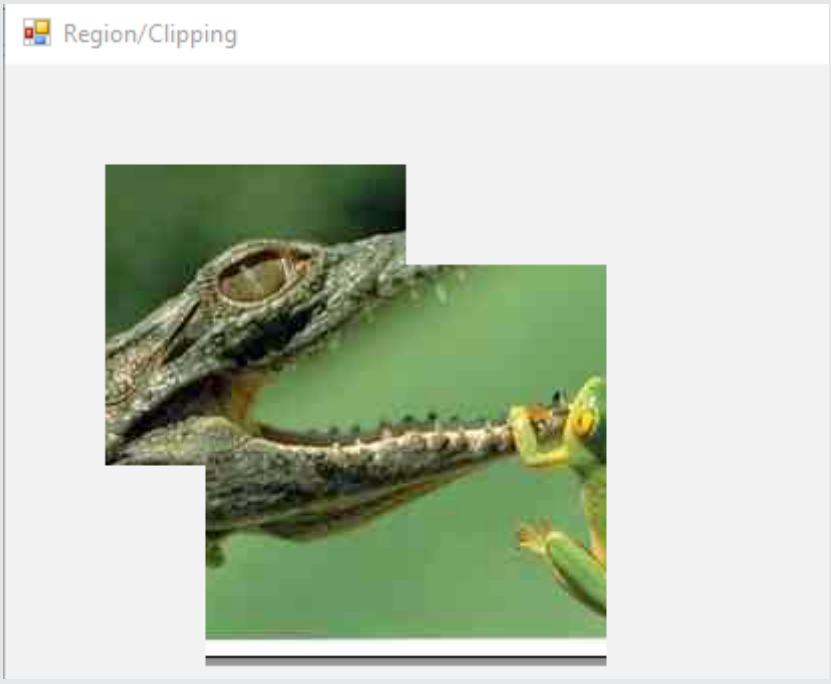
```
    g.SetClip(reg2, CombineMode.Replace);
```

Normales Zeichnen einer Bitmap, die normalerweise die gesamte Fensterfläche füllen würde:

```
g.DrawImage(pictureBox1.Image, 0, 0);
```

Ergebnis

Im Ergebnis wird die Grafikausgabe auf den gewünschten Clipping-Bereich eingeschränkt:

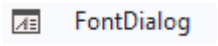


4.6 Standarddialoge

Im Zusammenhang mit der Bearbeitung von Grafiken stehen Ihnen zwei wesentliche Dialoge zur Verfügung:

- Schriftauswahl
- Farbauswahl

4.6.1 Schriftauswahl



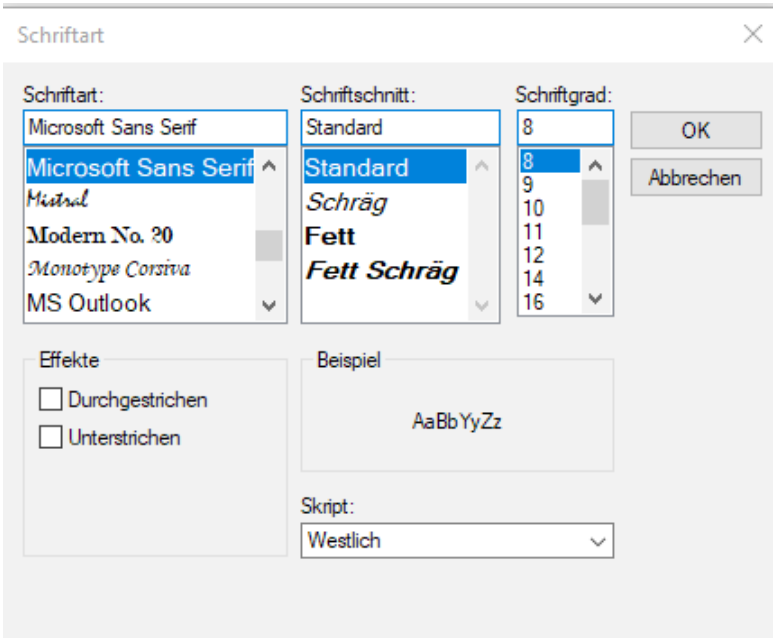
Der wohl jedem bekannte Dialog zur Auswahl einer Schrift bzw. deren Parameter (Größe, Farbe etc.) lässt sich über die entsprechende Komponente *FontDialog* in Ihre Anwendung einbinden.

Die Anzeige erfolgt zur Laufzeit mittels *ShowDialog*-Methode:

```
if (fontDialog1.ShowDialog() == DialogResult.OK)
 {  ...  }
```

Über den Rückgabewert der Methode können Sie den Status beim Beenden der Dialogbox ermitteln *(DialogResult.OK* oder *DialogResult.Abort*).

Die zweifelsfrei interessanteste Eigenschaft dieser Komponente ist *Font*. Sie können diese Eigenschaft sowohl vor dem Aufruf des Dialogs initialisieren als auch nach der Anzeige der Dialogbox auswerten.



Das Zuweisen der Fontattribute, zum Beispiel an einen Button, gestaltet sich absolut simpel:

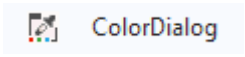
```
button1.Font = fontDialog1.Font;
```

Außer *Font* sind die in der folgenden Tabelle aufgeführten Eigenschaften für das Verhalten bzw. das Aussehen der Dialogbox von Bedeutung:

Eigenschaft	Beschreibung
AllowScriptChange	Skript (Westlich, Arabisch, Türkisch etc.) kann geändert werden.
AllowSimulations	Windows (GDI) darf Schriften simulieren.
AllowVectorFonts	Auswahl von Vektorschriftarten zulassen

AllowVerticalFonts	Vertikale Fonts sind zulässig.
Color	Die aktuelle Schriftfarbe
FixedPitchOnly	Es werden nur Schriftarten mit fester Zeichenbreite angezeigt.
FontMustExist	Es können nur vorhandene Schriftarten ausgewählt werden.
MaxSize, MinSize	Maximaler und minimaler Schriftgrad, der ausgewählt werden kann
ShowColor	Anzeige der Farbauswahl-Combobox
ShowEffects	Anzeige der Effekte (Durchstreichen, Unterstreichen, Farbe)
ShowHelp	Anzeige des Hilfe-Buttons

4.6.2 Farbauswahl



Für die Auswahl von Farben zur Laufzeit können Sie den Standarddialog *ColorDialog* verwenden. Wie auch beim *FontDialog* genügt der einfache Aufruf der Methode *ShowDialog*, um den Dialog anzuzeigen:

Über den Rückgabewert der Methode können Sie den Status beim Beenden der Dialogbox ermitteln *(DialogResult.OK* oder *DialogResult.Abort*).

Im Mittelpunkt der Komponente steht die Eigenschaft *Color* (ein ARGB-Wert), die Sie sowohl vor dem Aufruf des Dialogs initialisieren als auch nach der Anzeige der Dialogbox auswerten können.

Beispiel 4.79: Verändern der Formularfarbe

C#

```
colorDialog1.Color = BackColor;
if (colorDialog1.ShowDialog() == DialogResult.OK)
{
    BackColor = colorDialog1.Color;
}
```

Die wichtigsten Eigenschaften des Dialogs zeigt die folgende Tabelle:

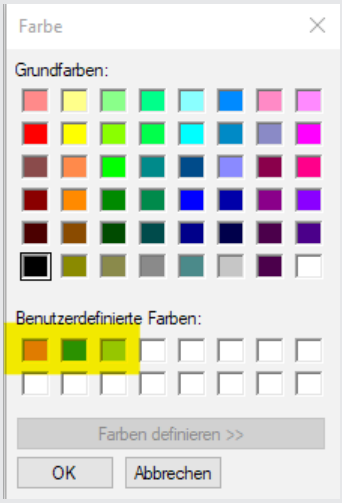
Eigenschaft	Bemerkung
AllowFullOpen, *FullOpen*	Ein-/Ausblenden der rechten Seite des Dialogs zum Definieren eigener Farben zulassen
AnyColor	Anzeigen aller verfügbaren Farben
Color	Ruft die von den Benutzern ausgewählte Farbe ab oder legt diese fest
CustomColors	Definieren oder Abfragen von nutzerdefinierten Farben (siehe Beispiel)
FullOpen	Ein-/Ausblenden der rechten Seite des Dialogs zum Definieren eigener Farben beim Öffnen des Dialogs
SolidColorOnly	Nur Volltonfarben können ausgewählt werden.

Beispiel 4.80: Zuweisen von nutzerdefinierten Farben

C#

```
colorDialog1.AllowFullOpen = false;
colorDialog1.CustomColors = new int[] {6975964, 231202, 1294476};
if (colorDialog1.ShowDialog() == DialogResult.OK)
{
 BackColor = colorDialog1.Color;
}
```

Ergebnis



HINWEIS: Wer weitere und detailliertere Informationen über die Grafikausgabe mit GDI+ sucht, der sollte sich das Kapitel 7 zu Gemüte führen.

4.7 Praxisbeispiele

4.7.1 Ein Graphics-Objekt erzeugen

In klassischen Programmiersprachen ist es üblich, mit Methoden direkt auf die Zeichenoberfläche eines Formulars oder eines *Picture*-Controls zuzugreifen. Als .NET-Programmierer müssen Sie umdenken.

Zugriff auf alle wesentlichen Grafikmethoden erhalten Sie über ein Graphics-Objekt. Im Vergleich mit anderen .NET-Objekten hat es allerdings die Besonderheit, dass man es nicht mit dem *new*-Konstruktor erzeugen kann. Woher also nehmen wir es? Das vorliegende Beispiel zeigt Ihnen vier Möglichkeiten:

- Nutzung des im Paint-Event des Formulars übergebenen *Graphics*-Objekts,
- Nutzung des in der überschriebenen *OnPaint*-Methode übergebenen *Graphics*-Objekts,
- Erzeugen eines neuen *Graphics*-Objekts mit der *CreateGraphics*-Methode des Formulars,
- Nutzung des im *Paint*-Event einer *PictureBox* (oder einer anderen Komponente, die über dieses Ereignis verfügt) übergebenen *Graphics*-Objekts.

Lassen Sie uns also ein wenig experimentieren!

Zunächst soll uns ein nacktes Windows Form genügen, auf das wir verschiedenfarbige Ellipsen zeichnen wollen. Später ergänzen wir noch weitere Steuerelemente *(Button*, *PictureBox*).

Variante 1: Verwendung des Paint-Events

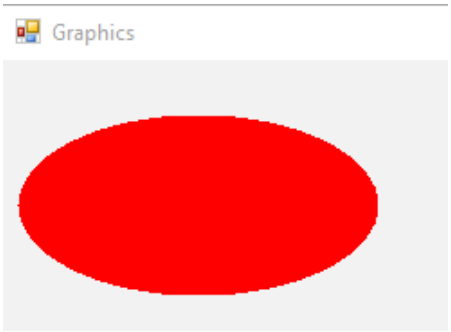
Die in der *System.Windows.Forms.Form*-Basisklasse implementierte *OnPaint*-Methode wird automatisch nach jedem Freilegen und Verdecken des Fensters aufgerufen. Sie löst das *Paint*-Ereignis aus, das wir in einem *Paint*-Eventhandler abfangen und behandeln wollen.

Über das Argument des Events ist ein *Graphics*-Objekt verfügbar:

```
    private void Form1_Paint(object sender, PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        g.FillEllipse(new SolidBrush(Color.Red), 10, 30, 200, 100);  // rote
Ellipse
    }
```

Test

Die rote Ellipse erscheint sofort nach Programmstart und ist auch nach Freilegen und Verdecken des Fensters zu sehen (siehe folgende Abbildung).



Variante 2: Überschreiben der OnPaint-Methode

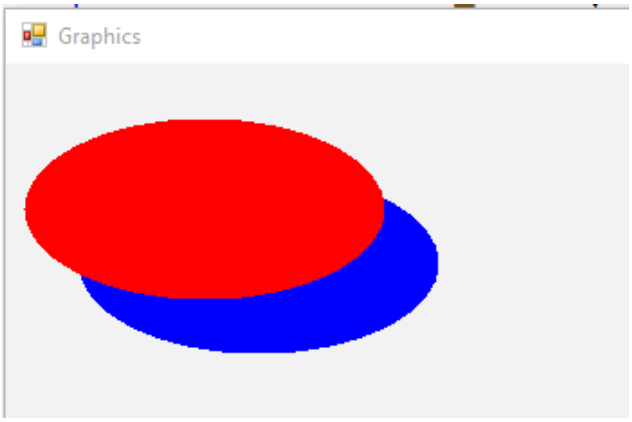
Dies ist die in der .NET-Dokumentation favorisierte Realisierung, bei der Sie keinen neuen Eventhandler verwenden müssen, sondern lediglich die *OnPaint*-Methode der Basisklasse überschreiben. Wir wollen nach diesem Prinzip eine versetzte blaue Ellipse zeichnen.

Implementieren Sie die Überschreibung wie folgt:

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillEllipse(new SolidBrush(Color.Blue), 40, 60, 200, 100);
    // blaue Ellipse
    base.OnPaint(e);  // Aufruf der Basisklassenmethode
}
```

Test

An der Reihenfolge (unten Blau, oben Rot) erkennen Sie, dass die überschriebene *OnPaint*-Methode zuerst abgearbeitet wurde und erst anschließend der bereits vorhandene *Paint*-Eventhandler:



HINWEIS: Wenn Sie die Anweisung *base.OnPaint(e)* auskommentieren, wird das *Paint*-Ereignis nicht mehr ausgelöst und nur noch die blaue Ellipse erscheint!

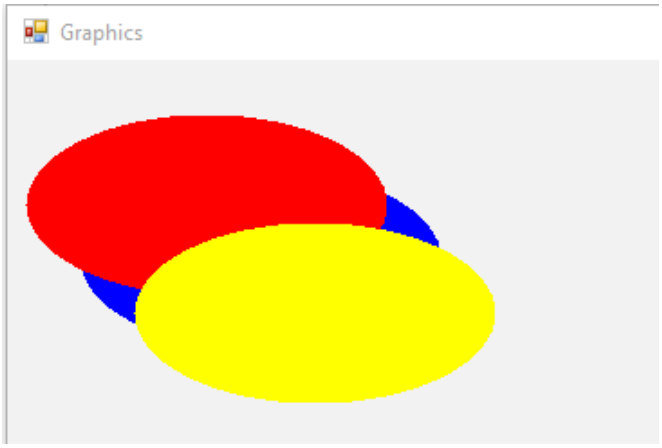
Variante 3: Graphics-Objekt mit CreateGraphics erzeugen

Diese Variante nutzt die Möglichkeit, über die *CreateGraphics*-Methode des Formulars ein neues *Graphics*-Objekt zu erzeugen. Allerdings benötigen wir hier einen *Button*, um das Zeichnen (versetzte gelbe Ellipse) zu demonstrieren.

```
private void button1_Click(object sender, EventArgs e)
{
    Graphics g = CreateGraphics();
    g.FillEllipse(new SolidBrush(Color.Yellow), 70, 90, 200, 100);
    // gelbe Ellipse
}
```

Test

Die gelbe Ellipse erscheint erst nach Klick auf den Button. Im Unterschied zur roten und blauen Ellipse (Variante 1 und 2) verschwindet diese Ellipse wieder, nachdem das Formular vorübergehend verdeckt wurde.



Variante 4: Verwendung des Graphics-Objekts einer PictureBox

Bei einer *PictureBox* – wie bei vielen anderen Komponenten auch – können Sie über die *CreateGraphics*-Methode auf die Zeichenfläche zugreifen. Sinnvoller ist allerdings auch hier die Nutzung des im *Paint*-Event übergebenen *Graphics*-Objekts, da Sie sich dann um die Restaurierung des Bildinhalts nicht weiter zu kümmern brauchen.

Ergänzen Sie die Oberfläche des Testformulars um eine *PictureBox* und erzeugen Sie einen Eventhandler für das *Paint*-Ereignis der *PictureBox*:

```
    private void pictureBox1_Paint(object sender, PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        g.FillEllipse(new SolidBrush(Color.Red), 10, 30, 200, 100);
    }
```

4.7.2 Zeichenoperationen mit der Maus realisieren

Dieses Beispiel zeigt Ihnen eine Möglichkeit, wie Sie einfache Zeichenoperationen (Linie, Ellipse, Rechteck) in ein eigenes Programm integrieren können. Dreh- und Angelpunkt ist die Verwendung einer Hintergrundbitmap, mit deren Hilfe wir einen Gummiband-Effekt beim Zeichnen realisieren.

Oberfläche

Ein Windows Form und eine *ToolStrip*-Komponente zur Auswahl der Zeichenoperation. Zusätzlich fügen Sie noch eine *ColorDialog*-Komponente zur Auswahl der Malfarbe ein.

Wir haben die Oberfläche bewusst einfach gehalten, hier geht es um das Handling der Maus-Events und die Verwendung einer Hintergrundbitmap und nicht um Schönheit im Detail.

Quelltext

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

Zunächst eine Enumeration definieren:

```
    enum Figuren
    {
      Linie,
      Ellipse,
      Rechteck
    }
```

Eine Statusvariable für die Zeichenoperation:

```
    private Figuren figur = Figuren.Linie;
```

Die Bitmap und das *Graphics*-Objekt für die Hintergrundbitmap:

```
    private Bitmap bmp;
    private Graphics bckg;
```

Der Malstift:

```
    private Pen pen;
```

Start- und Endpunkt der Zeichenoperation:

```
private Point pt1;
private Point pt2;
```

Im Form_Load-Ereignishandler erzeugen wir zunächst die Hintergrundgrafik in der maximal nötigen Größe:

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

    Size maxSize = SystemInformation.PrimaryMonitorMaximizedWindowSize;
    bmp = new Bitmap(maxSize.Width, maxSize.Height);
    g = Graphics.FromImage(bmp);
```

Mit Hintergrundfarbe füllen:

```
    g.Clear(BackColor);
```

Zeichenstift initialisieren:

```
    pen = new Pen(Color.Black);
}
```

Mit dem Drücken der Maustaste beginnt der Zeichenvorgang, wir merken uns die Position:

```
private void Form1_MouseDown(object sender, MouseEventArgs e)
{
    pt1 = e.Location;
}
```

Jede Mausbewegung bei gedrückter linker Maustaste erfordert das Wiederherstellen der Grafik vor dem Zeichenvorgang und das erneute Zeichnen mit den neuen Mauskoordinaten:

```
private void Form1_MouseMove(object sender, MouseEventArgs e)
{
    pt2 = e.Location;
    if (e.Button == MouseButtons.Left)
    {
        using (Graphics g = CreateGraphics())
        {
           g.DrawImage(bmp, 0, 0);
           Zeichne(g);
        }
    }
}
```

Erst wenn die Maustaste losgelassen wird, fügen wir das gerade gewählte Zeichenobjekt mit den aktuellen Koordinaten in die Hintergrundbitmap ein:

```
private void Form1_MouseUp(object sender, MouseEventArgs e)
{
    Zeichne(g);
}
```

Die eigentliche Zeichenroutine unterscheidet die einzelnen Zeichenobjekte:

```
private void Zeichne(Graphics graphic)
{
    switch (figur)
    {
        case Figuren.Linie:
        {
                graphic.DrawLine(pen, pt1, pt2);
                break;
        }
        case Figuren.Rechteck:
        {
                graphic.DrawRectangle(pen, pt1.X, pt1.Y,
                           pt2.X - pt1.X, pt2.Y - pt1.Y);
                break;
        }
        case Figuren.Ellipse:
        {
                graphics.DrawEllipse(pen, pt1.X, pt1.Y,
                            pt2.X - pt1.X, pt2.Y - pt1.Y);
                break;
        }
    }
}
```

Auch nach einem Verdecken des Fensters soll die Grafik wiederhergestellt werden:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.DrawImage(bmp, 0, 0);
}
```

Auswahl der Zeichenobjekte über den *ToolStrip*:

```
private void toolStripButton1_Click(object sender, EventArgs e)
{
    figur = Figuren.Linie;
}

private void toolStripButton2_Click(object sender, EventArgs e)
{
    figur = Figuren.Ellipse;
}

private void toolStripButton3_Click(object sender, EventArgs e)
{
    figur = Figuren.Rechteck;
}
```

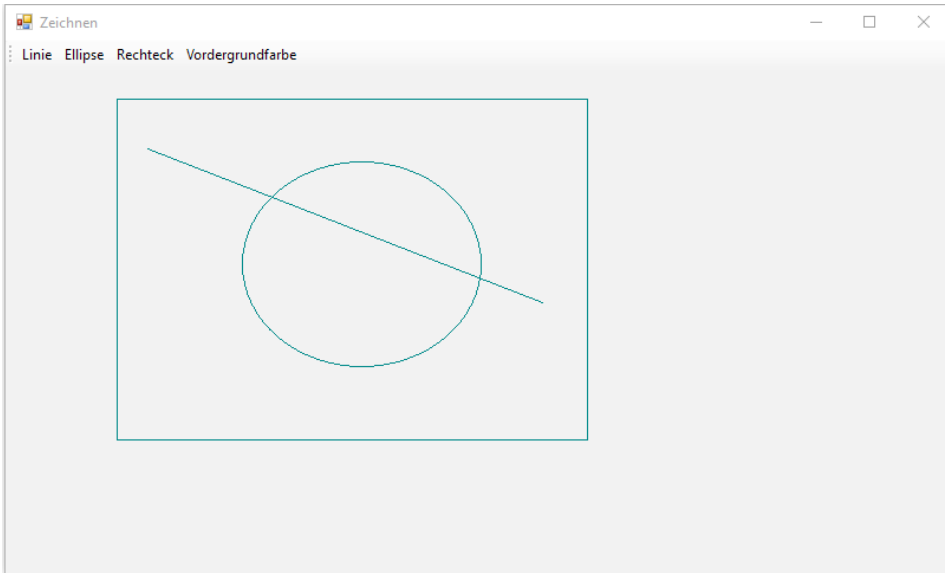
Auswahl der Malfarbe:

```
private void toolStripButton4_Click(object sender, EventArgs e)
{
    colorDialog1.AllowFullOpen = true;
    if (colorDialog1.ShowDialog() == DialogResult.OK)
```

```
        {
            pen = new Pen(colorDialog1.Color);
        }
    }
}
```

Test

Nach dem Start können Sie Ihren künstlerischen Fähigkeiten freien Lauf lassen:



5 Druckausgabe

In diesem Kapitel wollen wir uns ausgiebig mit den Möglichkeiten beschäftigen, unter C# etwas aufs Papier zu bringen. Vier grundsätzliche Varianten bieten sich an:

- Drucken über die *PrintDocument*-Komponente,
- Drucken mithilfe von OLE-Automation,
- Drucken mit den Crystal Report-Komponenten,
- Drucken mit Reporting Services.

Hier beschränken wir uns auf die beiden ersten Möglichkeiten.

HINWEIS: Zunächst jedoch sollten Sie sich mit Kapitel 4 (Grafikprogrammierung) eingehend beschäftigt haben, da wir auf diesen Grundlagen aufbauen werden.

5.1 Einstieg und Übersicht

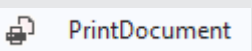
Bevor Sie mit viel Faktenwissen, endlosen Tabellen etc. gepeinigt werden, möchten wir Ihnen an einem Kurzbeispiel das Grundkonzept der Druckausgabe über *PrintDocument* demonstrieren.

5.1.1 Nichts geht über ein Beispiel

Beispiel 5.1: Druckausgabe eines 10 x 10 cm großen Rechtecks auf dem Standarddrucker

C#

Fügen Sie zunächst in Ihr Formular eine *PrintDocument*-Komponente ein:



Ergänzen Sie dann das *PrintPage*-Ereignis um folgende Zeilen:

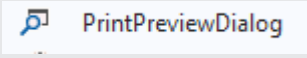
```
private void PrintDocument1_PrintPage(object sender,
                                      System.Drawing.Printing.PrintPageEventArgs e)
{
    e.Graphics.PageUnit = GraphicsUnit.Millimeter;
    e.Graphics.FillRectangle(new SolidBrush(Color.Blue), 30, 30, 100, 100);
 }
```

Fügen Sie nun noch einen *Button* ein, mit dem Sie die *Print*-Methode von *PrintDocument1* aufrufen:

```
private void Button1_Click(object sender, EventArgs e)
{
     printDocument1.Print();
}
```

Das war es auch schon, nach dem Klick auf den Button dürfte sich Ihr Drucker in Bewegung setzen. Doch was ist der Vorteil einer derartigen ereignisorientierten Programmierung beim Drucken? Die Antwort finden Sie, wenn Sie statt der Druckausgabe zunächst eine Druckvorschau am Bildschirm realisieren möchten.

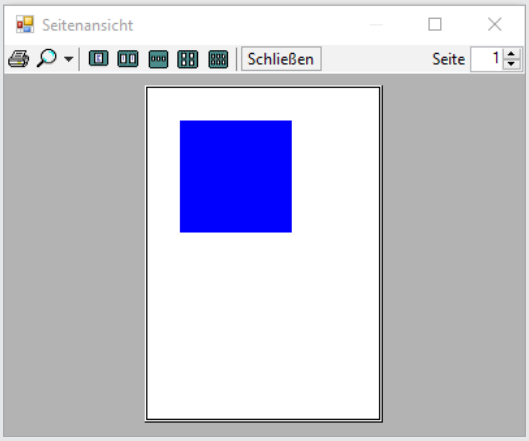
Fügen Sie einfach eine *PrintPreviewDialog*-Komponente in das Formular ein und verknüpfen Sie diese über die Eigenschaft *Document* mit der bereits vorhandenen *PrintDocument*-Komponente.



Der folgende Aufruf zeigt Ihnen bereits die Druckvorschau mit dem Rechteck an:

```
private void Button2_Click(object sender, EventArgs e)
{
    printPreviewDialog1.ShowDialog();
}
```

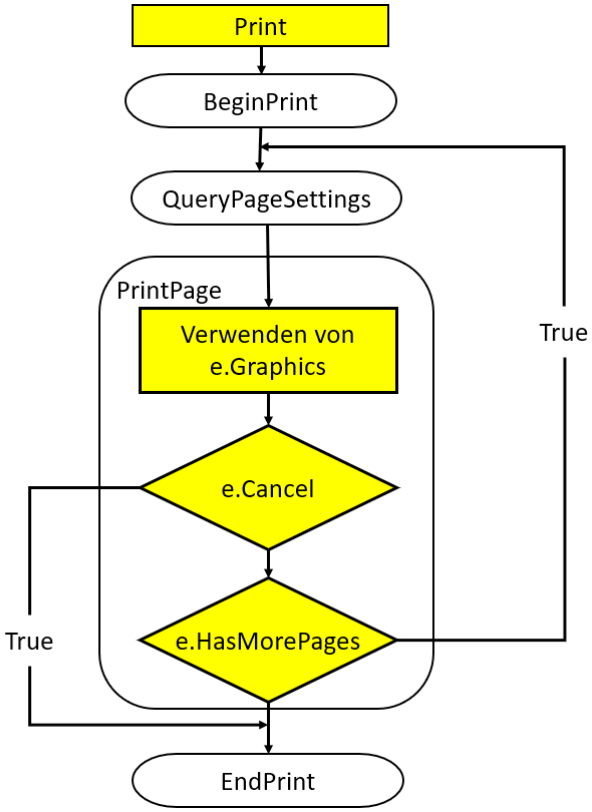
Das erzeugte Druckvorschaufenster:



HINWEIS: Eine Trennung beim Ausgabemedium (Papier, Druckvorschau am Bildschirm) gibt es nicht mehr, Sie entwickeln lediglich **eine** Ausgabelogik im *PrintPage*-Ereignis. Alle Ausgaben erfolgen systemneutral über ein dort bereitgestelltes *Graphics*-Objekt.

5.1.2 Programmiermodell

Wie Sie bereits dem vorhergehenden Beispiel entnehmen konnten, handelt es sich um ein ereignisorientiertes Programmiermodell. Die folgende Skizze soll Ihnen das Grundprinzip noch einmal verdeutlichen:



Mit dem Aufruf der *Print*-Methode wird zunächst das *BeginPrint*-Ereignis von *PrintDocument* ausgelöst. Hier bietet sich Ihnen die Möglichkeit, diverse Einstellungen einmalig zu konfigurieren. Nachfolgend wird das Ereignis *QueryPageSettings* vor dem Druck jeder Seite aufgerufen. Darauf folgt das wohl wichtigste Ereignis: *PrintPage*. Über den Parameter *e* erhalten Sie Zugriff auf das *Graphics*-Objekt des Druckers. Weiterhin legen Sie hier fest, ob weitere Seiten gedruckt werden sollen (*e.HasMorePages*) oder ob der Druck abgebrochen

(*e.Cancel*) werden soll. Steht der Druck weiterer Seiten an, wird die Ereigniskette, wie oben abgebildet, erneut durchlaufen.

Damit wird auch klar, dass Sie selbst dafür verantwortlich sind, welche Seite zu welchem Zeitpunkt gedruckt werden soll. Insbesondere im Zusammenhang mit den Druckersetup-Dialogen werden wir noch einigen Aufwand treiben müssen, aber das sind Sie ja bereits nicht anders gewohnt.

5.1.3 Kurzübersicht der Objekte

Folgende Komponenten stehen Ihnen im Zusammenhang mit der Druckausgabe in Windows Forms-Anwendungen zur Verfügung[1]:

Komponente	Beschreibung
PrintDocument	Der Dreh- und Angelpunkt der Druckausgabe. Über dieses Objekt bestimmen Sie den gewünschten Drucker, die Papierausrichtung, die Auflösung, die zu druckenden Seiten usw. Über das *PrintPage*-Ereignis erhalten Sie Zugriff auf das *Graphics*-Objekt des Druckers. Weiterhin steuern Sie hier den Druckverlauf (Anzahl der Seiten, Seitenauswahl, Abbruch). Mehr zu dieser Komponente finden Sie in den beiden folgenden Abschnitten.
PrintDialog	Der Windows-Standarddialog zur Auswahl eines Druckers sowie der wichtigsten Druckparameter (Seiten, Exemplare, Auflösung)
PageSetupDialog	Der Windows-Standarddialog zur Konfiguration der Druckausgabe (Seitenausrichtung, Papierausrichtung, Seitenränder)
PrintPreviewDialog	Eine komplette Druckvorschau, mit Navigationstasten, Zoom etc.
PrintPreviewControl	Eine Alternative für den *PrintPreviewDialog*. Bei dieser Komponente ist lediglich der Preview-Bereich vorhanden, für die Ansteuerung und Konfiguration sind Sie selbst verantwortlich.

Alle Komponenten können über die *Document*-Eigenschaft mit der *PrintDocument*-Komponente verknüpft werden. Sie müssen also die Parameter nicht „von Hand“ übergeben.

5.2 Auswerten der Druckereinstellungen

In den folgenden Abschnitten wollen wir versuchen, Ihnen die „Vorzüge“ der relativ unübersichtlichen Objektstruktur zu ersparen. Aus diesem Grund werden wir auf eine Auflistung von Eigenschaften und Methoden für die einzelnen Objekte verzichten, stattdessen stellen wir die zu lösende Aufgabe in den Vordergrund.

[1] Auf das *Chart*-Control gehen wir im Rahmen des Praxisbeispiels in Abschnitt 5.6.2 gesondert ein.

5.2.1 Die vorhandenen Drucker

Einen Überblick, welche Drucker auf dem aktuellen System installiert sind, können Sie sich über die Collection *InstalledPrinters* verschaffen.

Beispiel 5.2: Ausgabe aller Druckernamen in einer ComboBox und Markieren des aktiven Druckers

C#

```
...
using System.Drawing.Printing;
...
    private void Form1_Load(object sender, EventArgs e)
    {
       PrintDocument doc = new PrintDocument();
```

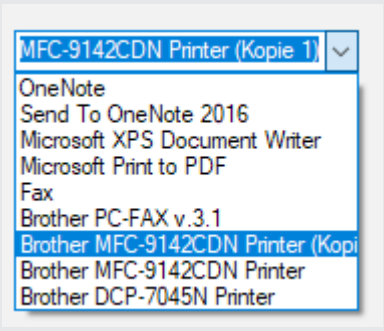
Füllen der *ComboBox*:

```
       foreach (string printername in PrinterSettings.InstalledPrinters)
       {
           comboBox1.Items.Add(printername);
       }
```

Auswahl des aktiven Druckers:

```
       comboBox1.Text = doc.PrinterSettings.PrinterName;
    }
```

Ergebnis



5.2.2 Der Standarddrucker

Möchten Sie überprüfen, ob der aktuell gewählte Drucker gleichzeitig auch der System-Standarddrucker ist, können Sie dies mithilfe der Eigenschaft *IsDefaultPrinter* realisieren.

Beispiel 5.3: Test auf Standarddrucker

C#

```
if (printDocument1.PrinterSettings.IsDefaultPrinter)
{
   MessageBox.Show("Standarddrucker");
}
```

Ergebnis

Den Standarddrucker erkennen Sie in der Systemsteuerung an dem Eintrag Standard, auch wenn dieser nicht verfügbar ist:

5.2.3 Verfügbare Papierformate/Seitenabmessungen

Geht es um die Abfrage, welche Papierarten der Drucker unterstützt, können Sie einen Blick auf die *PaperSizes*-Collection werfen. Diese gibt Ihnen nicht nur Auskunft über die Blattgröße (*Height, Width*), sondern auch über die Blattbezeichnung (*PaperName*) und den Typ (*Kind*).

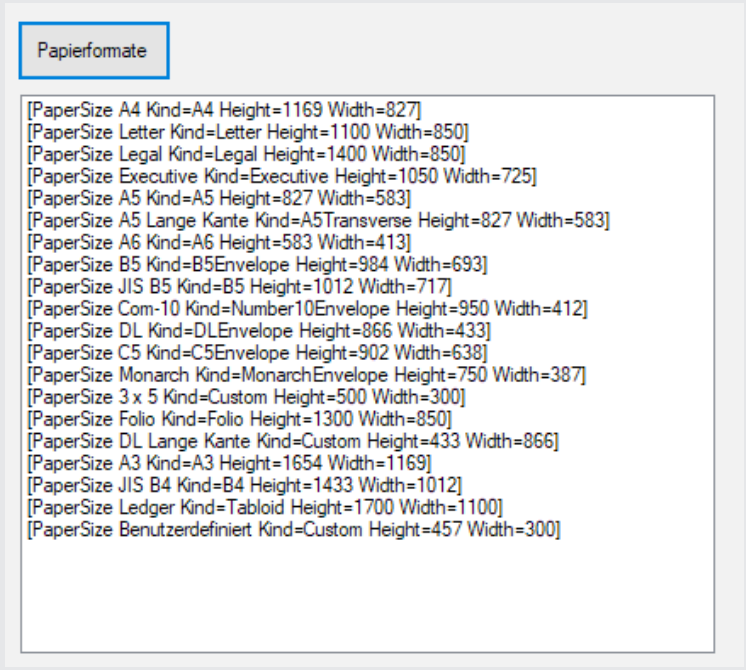
Beispiel 5.4: Anzeige aller Papierformate im Ausgabefenster

C#

```
using System.Drawing.Printing;
using System.Diagnostics;
...
    private void Button1_Click(object sender, EventArgs e)
    {
        foreach (PaperSize ps in printDocument1.PrinterSettings.PaperSizes)
        {
             listBox1.Items.Add(ps);
        }
    }
```

Ergebnis

Die Anzeige:



HINWEIS: Die Blattabmessungen werden in 1/100 Zoll zurückgegeben! Der Umrechnungsfaktor in Millimetern ist 0,254.

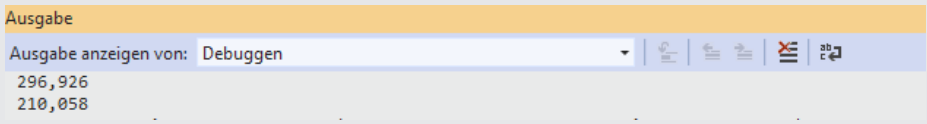
Beispiel 5.5: Anzeige der aktuellen Blattabmessungen in Millimetern

C#

```
using System.Diagnostics;
...
Debug.WriteLine(
```

```
    printDocument1.PrinterSettings.DefaultPageSettings.PaperSize.Height*0.254);
Debug.WriteLine(
    printDocument1.PrinterSettings.DefaultPageSettings.PaperSize.Width*0.254);
```

Ergebnis



Gleichzeitig steht Ihnen mit *System.Drawing.Printing.PaperKind* eine Aufzählung der Standardpapierformate zur Verfügung (Auszug):

Element	Beschreibung
A2	A2 (420 x 594 mm)
A3	A3 (297 x 420 mm)
A3Extra	A3 Extra (322 x 445 mm)
A3ExtraTransverse	A3 Extra quer (322 x 445 mm)
A3Rotated	A3 gedreht (420 x 297 mm)
A3Transverse	A3 quer (297 x 420 mm)
A4	A4 (210 x 297 mm)

5.2.4 Der eigentliche Druckbereich

Leider druckt nicht jeder Drucker bis zu den Blatträndern. Aus diesem Grund ist es wichtig, den eigentliche Druckbereich und insbesondere den Offset des Druckbereichs zu bestimmen. Verwenden Sie dazu die Eigenschaften *HardMarginX*, *HardMarginY* sowie *PrintableArea*.

HINWEIS: Vergessen Sie in diesem Zusammenhang die Eigenschaft *Margins* ganz schnell wieder, es handelt sich lediglich um theoretische Seitenränder, die Sie selbst definieren können.

Beispiel 5.6: Anzeige der physikalischen Blattränder

C#

```
private void Form1_Load(object sender, EventArgs e)
{
...
   PrintDocument pd = new PrintDocument();
   Debug.WriteLine($"Linker Rand: {pd.DefaultPageSettings.HardMarginX * 0.254}");
   Debug.WriteLine($"Oberer Rand:{pd.DefaultPageSettings.HardMarginY * 0.254) }";
```

```
   Debug.WriteLine($"Druckbreite:{pd.DefaultPageSettings.PrintableArea.Width *
0.254}");
   Debug.WriteLine($"Druckhöhe:{pd.DefaultPageSettings.PrintableArea.Height *
0.254}");
}
```

Ergebnis

Die Anzeige im Ausgabefenster:

5.2.5 Die Seitenausrichtung ermitteln

Die aktuelle Blatt- bzw. Seitenausrichtung können Sie über die Eigenschaft *Landscape* abfragen.

Beispiel 5.7: Abfrage der Seitenausrichtung

C#

```
using System.Drawing.Printing;
...
if (printDocument1.PrinterSettings.DefaultPageSettings.Landscape)
{
   MessageBox.Show("Hochformat");
};
```

5.2.6 Ermitteln der Farbfähigkeit

Ob Ihr aktuell gewählter Drucker auch in der Lage ist, mehr als nur Schwarz zu Papier zu bringen, lässt sich mit der Eigenschaft *SupportsColor* ermitteln.

Beispiel 5.8: Test auf Farbunterstützung

C#

```
using System.Drawing.Printing;
...
if (printDocument1.PrinterSettings.SupportsColor)
{
 MessageBox.Show("Unterstützt Farben");
};
```

5.2.7 Die Druckauflösung abfragen

Möchten Sie sich über die physikalische Druckauflösung des aktiven Druckers informieren, sollten Sie sich mit der Eigenschaft *PrinterResolution* näher beschäftigen.

Beispiel 5.9: Die horizontale Druckauflösung (readonly)

C#

```
MessageBox.Show(
       printDocument1.DefaultPageSettings.PrinterResolution.X.ToString());
```

Beispiel 5.10: Die vertikale Druckauflösung (readonly)

C#

```
MessageBox.Show(
        printDocument1.DefaultPageSettings.PrinterResolution.Y.ToString());
```

HINWEIS: Die Rückgabewerte entsprechen Punkten pro Zoll (Dots per Inch: dpi).

Alternativ können Sie über die *Kind*-Eigenschaft einen der folgenden Werte abrufen:

Kind	Beschreibung
Custom	Benutzerdefinierte Auflösung
Draft	Auflösung in Entwurfsqualität
High	Hohe Auflösung
Low	Niedrige Auflösung
Medium	Mittlere Auflösung

Beispiel 5.11: Ausgabe der Druckauflösung

C#

```
MessageBox.Show(
     printDocument1.DefaultPageSettings.PrinterResolution.Kind.ToString());
```

5.2.8 Ist beidseitiger Druck möglich?

Ob der Drucker duplexfähig ist, das heißt, ob er beidseitig drucken kann, ermitteln Sie über die Eigenschaft *CanDuplex*.

Beispiel 5.12: Duplexfähigkeit bestimmen

C#

```
using System.Drawing.Printing;
...
if (PrintDocument1.PrinterSettings.CanDuplex)
{
  MessageBox.Show("Duplex");
}
```

5.2.9 Abfragen von Werten während des Drucks

Statt wie in den vorhergehenden Beispielen mit der *PrintDocument*-Komponente, nutzen Sie besser den im *PrintPage*-Ereignis angebotenen Parameter *e*. Über diesen erhalten Sie Zugriff auf die gewünschten Eigenschaften.

Beispiel 5.13: Papiergröße während des Drucks bestimmen

C#

```
private void PrintDocument1_PrintPage(object sender, PrintPageEventArgs e)
{
   Debug.WriteLine(e.PageSettings.PaperSize);
}
```

5.3 Festlegen von Druckereinstellungen

Nachdem wir im vorhergehenden Abschnitt recht passiv mit den Druckeroptionen umgegangen sind und uns auf das reine Auslesen beschränkt haben, wollen wir uns im Weiteren um das Konfigurieren des Druckers kümmern.

5.3.1 Einen Drucker auswählen

Der wohl erste Schritt, wenn mehr als ein Drucker zur Verfügung steht, ist die Auswahl des Druckers. Zwei Varianten bieten sich an:

- Verwendung der Eigenschaft *PrinterName*,
- Verwendung der *PrintDialog*-Komponente (siehe Abschnitt 5.4).

HINWEIS: Nach dem Setzen der Eigenschaft bzw. vor dem endgültigen Drucken sollten Sie mit der Eigenschaft *IsValid* überprüfen, ob die Konfiguration auch realisierbar ist.

Beispiel 5.14: Setzen der *PrinterName*-Eigenschaft und nachfolgende Prüfung mit *IsValid*

C#

```
foreach (string printername in PrinterSettings.InstalledPrinters)
{
   comboBox1.Items.Add(printername);
}
comboBox1.SelectedIndex = 0;
printDocument1.PrinterSettings.PrinterName = comboBox1.Text;
printDocument1.PrinterSettings.DefaultPageSettings.Margins =
               new Margins(100, 100, 100, 100);
if (printDocument1.PrinterSettings.IsValid)
{
  printPreviewDialog1.ShowDialog();
}
```

Beispiel 5.15: Verwendung des *QueryPageSettings*-Ereignisses zur Auswahl eines Druckers

C#

```
private void PrintDocument1_QueryPageSettings(object sender,
             QueryPageSettingsEventArgs e)
 {
      e.PageSettings.PrinterSettings.PrinterName = "FRITZFax Drucker";
 }
```

5.3.2 Drucken in Millimetern

Sie werden hoffentlich nicht auf die Idee kommen, Zeichnungen in Pixeln auf dem Drucker auszugeben. Je nach Modell ist sonst Ihre Grafik mikroskopisch klein oder riesengroß. Bleibt die Frage, wie Sie die Maßeinheit auf Millimeter umstellen können. Die Lösung ist schnell gefunden, über die Eigenschaft *PageUnit* können Sie eine der folgenden Maßeinheiten auswählen:

Konstante	Beschreibung
Display	Eine Einheit entspricht 1/75 Zoll.
Document	Eine Einheit entspricht 1/300 Zoll.
Inch	Eine Einheit entspricht 1 Zoll.
Millimeter	Eine Einheit entspricht einem Millimeter.
Pixel	Eine Einheit entspricht einem Gerätepixel.
Point	Eine Einheit entspricht 1/72 Zoll (Point).

Beispiel 5.16: Setzen der Maßeinheit im *PrintPage*-Ereignis

C#

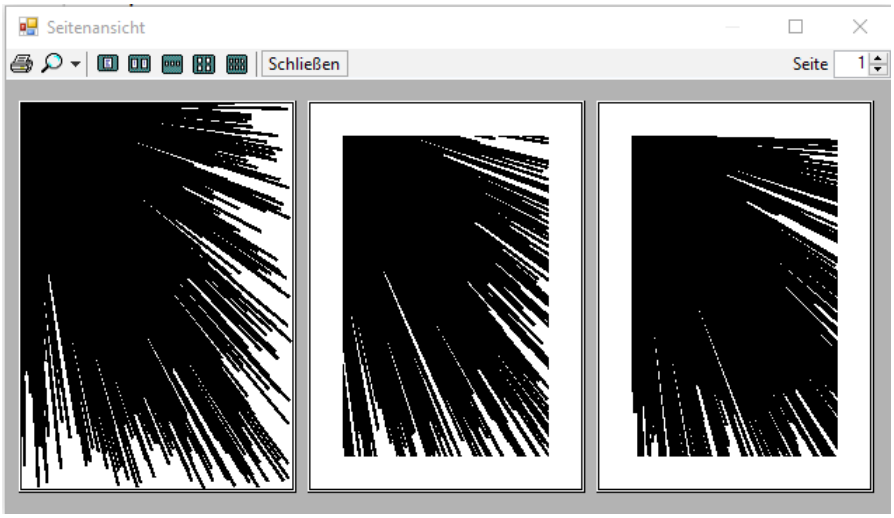
```
private void PrintDocument1_PrintPage(object sender, PrintPageEventArgs e)
{
   e.Graphics.PageUnit = GraphicsUnit.Millimeter;
```

```
   e.Graphics.DrawLine(new Pen(Color.Black, 10), 50, 100, 150, 200);
 ...
}
```

5.3.3 Festlegen der Seitenränder

Tja, welche Ränder meinen Sie denn? Geht es um Seitenränder wie zum Beispiel in Microsoft Word, nutzen Sie die Eigenschaft *Margins*. Allerdings bedeutet das Festlegen per Code oder mithilfe der Dialogbox *PageSetupDialog* noch lange nicht, dass diese Ränder auch zwingend eingehalten werden. Dafür sind Sie im *PrintPage*-Ereignis selbst verantwortlich.

Die folgende Abbildung soll Ihnen die Problematik verdeutlichen. In jedem der drei Fälle werden, beginnend mit der Koordinate 0,0 (linke obere Ecke), Zufallslinien gezeichnet, die maximal die Abmessungen des Blatts erreichen.



Beispiel 5.17: Einstellung der Seitenränder

C#

Seite 1 zeigt deutlich, dass die eingestellten Seitenränder (100,100,100,100) vollkommen ignoriert werden. Seite 2 berücksichtigt bereits die eingestellten Seitenränder durch die Verwendung eines Clipping-Bereichs und Seite 3 bringt auch den Koordinatenursprung an die richtige Position:

```
Graphics g = e.Graphics;
g.PageUnit = GraphicsUnit.Display;
Random rnd = new Random();
if (page == 2)
{
     g.SetClip(e.MarginBounds);
}
```

```
if (page == 3)
{
    g.SetClip(e.MarginBounds);
    g.TranslateTransform(e.MarginBounds.Left, e.MarginBounds.Top);
}
for (int i = 0; i <= 500; i++)
{
     g.DrawLine(new Pen(Color.Black, 10), 0, 0,
                          random.Next(e.PageBounds.Width),
                          random.Next(e.PageBounds.Height));
}
if (page == 3)
{
   page = 1;
   e.HasMorePages = false;
}
else
{
   page++;
   e.HasMorePages = true;
}
```

Damit brauchen Sie sich beim Zeichnen eigentlich nur noch um die Breite und Höhe des bedruckbaren Bereichs (*e.MarginBounds.Width* bzw. *e.MarginBounds.Height*) zu kümmern. Die linke obere Ecke ist bereits korrekt gesetzt.

HINWEIS: Die Eigenschaft *Margins* hebt natürlich keine physikalischen Grenzen auf. Wenn der Drucker einen entsprechenden Offset aufweist, müssen Sie diesen auch berücksichtigen (siehe Abschnitt 5.2.4).

5.3.4 Druckjobname

Was im Normalfall eher sekundär ist, kann in Netzwerkumgebungen bzw. Multiuser-Umgebungen für mehr Übersicht sorgen. Über die Eigenschaft *DocumentName* können Sie vor dem Drucken einen aussagekräftigen Druckjobnamen festlegen, der im Druckerspooler angezeigt wird.

Beispiel 5.18: Ändern des Druckjobnamens

C#

```
printDocument1.DocumentName = "Mein erster C#.NET-Druckversuch";
```

5.3.5 Anzahl der Kopien

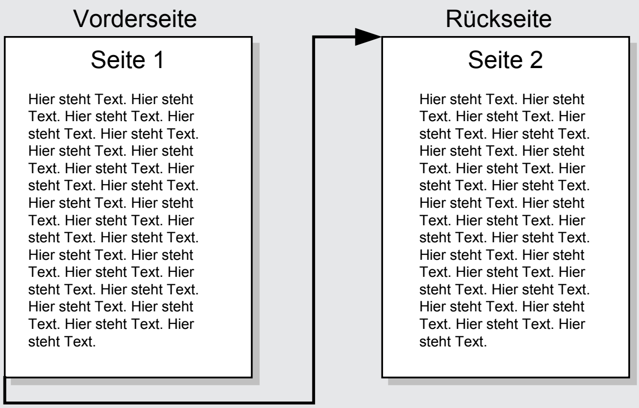
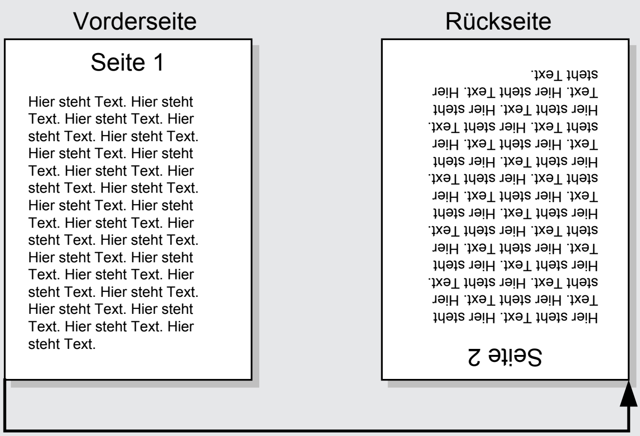
Die Anzahl der Druckkopien kann zum einen mithilfe des Dialogs *PrintDialog*, zum anderen auch per Code festgelegt werden. Nutzen Sie die Eigenschaft *Copies*.

Beispiel 5.19: Drei Kopien

C#
```
printDocument1.PrinterSettings.Copies = 3;
```

5.3.6 Beidseitiger Druck

Geht es um das beidseitige Bedrucken von Papier, was aus ökologischer Sicht sicher sinnvoller ist, müssen Sie sich zunächst vergewissern, ob der Drucker auch über dieses Feature verfügt (siehe Abschnitt 5.2.8). Nachfolgend können Sie über die *Duplex*-Eigenschaft den gewünschten Wert einstellen.

<table>
<tr><th>Konstante</th><th>Beschreibung</th></tr>
<tr><td>Default</td><td>Die Standardeinstellungen des Druckers werden genutzt.</td></tr>
<tr><td>Simplex</td><td>Der „normale“ einseitige Druck</td></tr>
<tr><td>Horizontal</td><td> 
</td></tr>
<tr><td>Vertical</td><td> 
</td></tr>
</table>

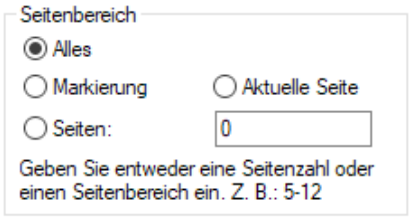
Beispiel 5.20: Einstellen der *Duplex*-Eigenschaft

C#

```
using System.Drawing.Printing;
...
printDocument1.PrinterSettings.Duplex = Duplex.Horizontal;
```

5.3.7 Seitenzahlen festlegen

Die Überschrift dürfte auf den ersten Blick etwas missverständlich klingen, da Sie doch selbst über den zu druckenden Inhalt entscheiden. Wenn Sie sich jedoch an den Druckerdialog erinnern, sind dort auch Optionen für die Seitenauswahl möglich:



Leider ist die Unterstützung dieser Option ein nicht ganz leicht verdaulicher Brocken.

Zunächst einmal unterscheiden Sie die vier gewählten Optionen (Alles, Markierung, Seiten, Aktuelle Seite) mithilfe der folgenden Konstanten über die *PrintRange*-Eigenschaft.

Konstante	Beschreibung
AllPages	Alle Seiten drucken
Selection	Die ausgewählten Seiten drucken (per Userauswahl)
SomePages	Die Seiten zwischen *FromPage* und *ToPage* sollen gedruckt werden.
CurrentPage	Die aktuelle Seite drucken (was die aktuelle Seite ist, bestimmt Ihr Programm)

Ein Beispiel zeigt die Auswertung der vier Varianten im Zusammenhang.

Beispiel 5.21: Berücksichtigung des vorgegebenen Druckbereichs

C#

```
using System.Drawing.Printing;

public partial class Form1 : Form
{
```

Eine Variable für die aktuell zu druckende Seite:

```
    int page;
```

Die aktuelle Seite bei Mehrfachauswahl:

```
int selectedIndex;
```

Die maximal druckbaren Seiten (Dokumentlänge):

```
const int maxPages = 30;
```

Beim Programmstart füllen wir zunächst eine *ListBox* mit den möglichen Seitenzahlen (1...30):

```
private void FrmSeitenzahlenFestlegen_Load(object sender, EventArgs e)
{
    for (int i = 0; i <= maxPages; i++)
    {
            listBox1.Items.Add($"Seite {i}");
    }
}
```

Druckerdialog anzeigen und im Erfolgsfall die Druckvorschau öffnen:

```
private void Button1_Click(object sender, EventArgs e)
{
    if (printDialog1.ShowDialog() == DialogResult.OK)
    {
            printPreviewDialog1.ShowDialog();
    }
}
```

Vorbereiten des „Druckvorgangs“:

```
private void PrintDocument1_BeginPrint(object sender,
                                System.Drawing.Printing.PrintEventArgs e)
{
    page = 1;
    selectedIndex = 0;
```

Zur Sicherheit prüfen wir, ob auch mindestens eine Seite ausgewählt wurde (nur bei Seitenauswahl):

```
    switch (printDialog1.PrinterSettings.PrintRange)
    {
        case PrintRange.Selection:
                if (listBox1.SelectedItems.Count == 0)
                {
                    e.Cancel = true;
                }
            break;
    }
}
```

Der eigentliche Druckvorgang:

```
private void PrintDocument1_PrintPage(object sender,
                            System.Drawing.Printing.PrintPageEventArgs e)
{
    int printPage = 0;
```

Je nach Auswahl im Druckdialog müssen wir nun die aktuelle Seite bestimmen:

```
        switch (printDialog1.PrinterSettings.PrintRange)
        {
```

Es soll die aktuelle Seite gedruckt werden (der Wert wird per *NumericUpDown* bestimmt):

```
            case PrintRange.CurrentPage:
                    printPage = (int) numericUpDown1.Value;
                    break;
```

Es soll ein Seitenbereich gedruckt werden:

```
            case PrintRange.SomePages:
                    printPage = page + e.PageSettings.PrinterSettings.FromPage -
1;
                    break;
```

Es sollen alle Seiten gedruckt werden:

```
            case PrintRange.AllPages:
                    printPage = page;
                    break;
```

Eine Seitenauswahl (*ListBox*) soll gedruckt werden:

```
            case PrintRange.Selection:
                    printPage = listBox1.SelectedIndices[selectedindex];
                    selectedIndex++;
                    break;
        }
```

Hier können Sie die Seite auswerten und die Drucklogik unterbringen:

```
        switch (printPage)
        {
            case 1:
                    break;
            case 2:
                    break;
            // ...
        }
```

Unser Beispiel zeigt stattdessen die aktuelle Seitenzahl an:

```
        e.Graphics.DrawString($"Seite: {printPage}",
               new Font("Arial", 20, FontStyle.Bold, GraphicsUnit.Millimeter),
                        Brushes.Black, 70, 50);
```

Eine Seite weiter:

```
        page++;
```

Je nach Auswahl im Druckerdialog bestimmen wir jetzt, ob es noch weitere Seiten gibt:

```
        switch (printDialog1.PrinterSettings.PrintRange)
        {
```

```
            case PrintRange.Selection:
                    e.HasMorePages = selectedIndex < listBox1.SelectedIndices.Count;
                    break;
            case PrintRange.CurrentPage:
                    e.HasMorePages = false;
                    break;
            case PrintRange.SomePages:
                    e.HasMorePages = (printPage <
                       e.PageSettings.PrinterSettings.ToPage);
                    break;
            case PrintRange.AllPages:
                    e.HasMorePages = (page <= maxPages);
                    break;
        }
    }
}
```

HINWEIS: Über die Eigenschaften *MinimumPage* und *MaximumPage* können Sie maximale Grenzen für die Auswahl des Druckbereichs festlegen.

Beispiel 5.22: Druckbereich maximal von Seite 1 bis Seite 10

C#

```
printDocument1.PrinterSettings.MinimumPage = 1;
printDocument1.PrinterSettings.MaximumPage = 10;
```

5.3.8 Druckqualität verändern

Unter diesem Punkt verstehen wir zum einen die Einstellung der dpi-Zahl des Druckers, zum anderen die Optionen bei der Ausgabe von Grafiken (*Antialiasing, CompositingQuality*).

Beispiel 5.23: Setzen der Druckauflösung

C#

```
printDocument1.DefaultPageSettings.PrinterResolution =
          printDocument1.PrinterSettings.PrinterResolutions[2];
```

5.3.9 Ausgabemöglichkeiten des Chart-Controls nutzen

An dieser Stelle wollen wir uns einem Spezialfall zuwenden. Im Mittelpunkt stehen die *Chart*-Komponente und deren Möglichkeiten zur Druckausgabe. Diese sind zwar auf den ersten Blick recht überschaubar, allerdings dürften Sie damit auch alle wichtigen Anwendungsfälle problemlos abdecken. Unser Interesse gilt vor allem der *Printing*-Eigenschaft des *Chart*-Controls. Diese bündelt alle Aktivitäten rund um die Druckausgabe.

Folgende Methoden werden bereitgestellt:

Methode	Beschreibung
PageSetup	... zeigt den bekannten Page Setup-Dialog an. Mehr zur Verwendung dieses Dialogs finden Sie in Abschnitt 5.4.2.
Print	... druckt das vorliegende Diagramm. Übergeben Sie als Parameter *true*, wird der bekannte Druckdialog zur Druckerauswahl angezeigt (siehe Abschnitt 5.4.1).
PrintPaint	Ausgabe des Diagramms auf einem *Graphics*-Objekt.
PrintPreview	Statt der direkten Druckauswahl wird eine Druckvorschau angezeigt.

Neben obigen Methoden steht über die *Printing*-Eigenschaft auch ein *PrintDocument* zur Verfügung, über das Sie die Einstellungen des Druckers auslesen oder auch beeinflussen können, wie Sie es in den beiden vorhergehenden Abschnitten bereits im Detail gesehen haben.

Beispiel 5.24: Verwendung von *Printing.PrintDocument*

C#

Anzeige, ob es sich um den Standarddrucker handelt:

```
Text = chart1.Printing.PrintDocument.PrinterSettings.
       IsDefaultPrinter.ToString();
```

5.4 Die Druckdialoge verwenden

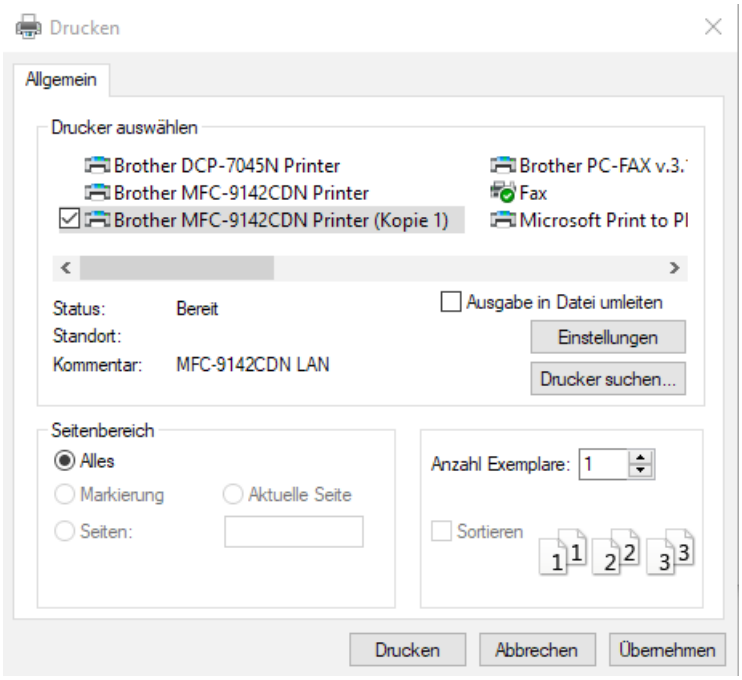
Im Folgenden wollen wir Ihnen kurz die drei wesentlichen Druckdialoge

- *PrintDialog,*
- *PageSetupDialog,*
- *PrintPreviewDialog*

und deren wichtigste Parameter im Zusammenspiel mit der Druckausgabe vorstellen.

5.4.1 PrintDialog

Der allgemein bekannte Standarddruckdialog wird mit der Komponente *PrintDialog* eingebunden.



Die Komponente selbst können Sie mittels *Document*-Eigenschaft direkt an ein *PrintDocument*-Control binden. Alle gewählten Parameter werden automatisch an *PrintDocument* übergeben.

Eigenschaft	Beschreibung
AllowPrintToFile	… aktiviert das Kontrollkästchen „Ausgabe in Datei“
AllowSelection	… aktiviert das Optionsfeld „Seiten von … bis …“
AllowSomePages	… aktiviert das Optionsfeld „Seiten“
ShowHelp	… aktiviert die Schaltfläche „Hilfe“
ShowNetwork	… aktiviert die Schaltfläche „Netzwerk“ (nur in der Theorie)
PrinterSettings	Über diese Eigenschaft können Sie Standardwerte vorgeben sowie die Einstellungen des Dialogfelds abfragen.
PrintToFile	… fragt den Wert des Kontrollkästchens „Ausgabe in Datei“ ab

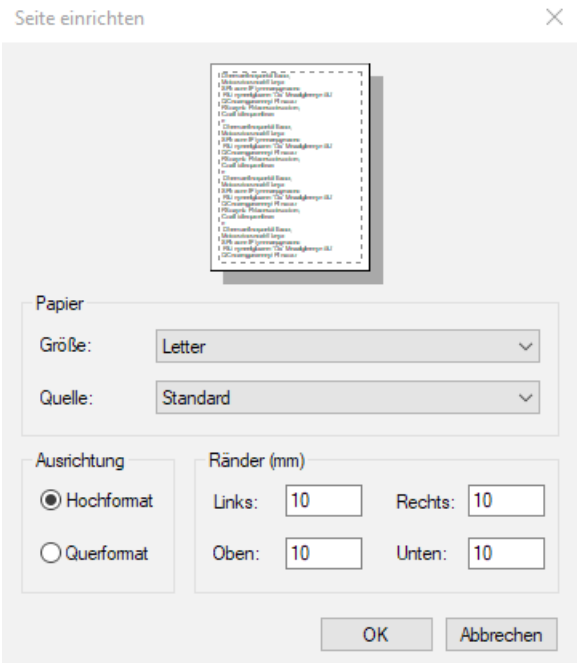
Beispiel 5.25: Anzeige des Dialogs und Abfrage des gewählten Druckers

C#

```
if (printDialog1.ShowDialog() == DialogResult.OK)
{
 MessageBox.Show(printDialog1.PrinterSettings.PrinterName, "Hinweis",
                 MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
}
```

5.4.2 PageSetupDialog

Aus vielen Programmen dürfte Ihnen der folgende Dialog bekannt sein, mit dem Sie einen Menüpunkt „Seite einrichten“ realisieren können.



Auch diese *PageSetupDialog*-Komponente können Sie mittels *Document*-Eigenschaft direkt an eine *PrintDocument*-Komponente binden, um die eingestellten Parameter automatisch zu übernehmen.

Eigenschaft	Beschreibung
AllowMargins	… aktiviert den Bereich „Ränder (mm)“
AllowOrientation	… aktiviert den Bereich „Orientierung“
AllowPaper	… aktiviert den Bereich „Papier“
AllowPrinter	… aktiviert die Schaltfläche „Drucker…“
ShowHelp	… aktiviert die Schaltfläche „Hilfe“
MinMargins	… legt die minimalen Werte für die Ränder fest
PageSettings *PrinterSettings*	… hier können Sie Standardwerte vorgeben bzw. die Werte abfragen.

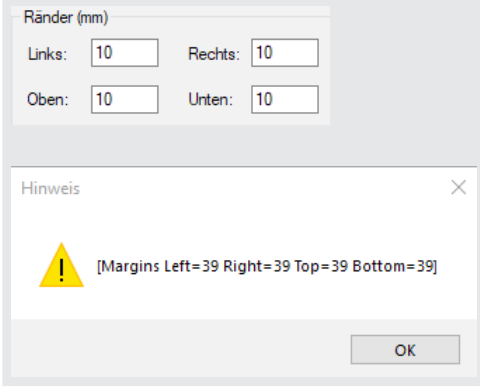
HINWEIS: Über das Ereignis *HelpRequest* können Sie auf den Button „Hilfe“ reagieren!

Beispiel 5.26: Aufruf der Dialogbox und Anzeige der neu gesetzten Ränder

C#

```
if (pageSetupDialog1.ShowDialog()== DialogResult.OK)
{
      MessageBox.Show(pageSetupDialog1.PageSettings.Margins.ToString(), "Hinweis",
            MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
}
```

Ergebnis



Probleme mit den Rändern

Doch wo viel Licht, da ist auch Schatten, ein kleiner Bug hat sich in die Komponente eingeschlichen, der aber wahrscheinlich nur in den lokalisierten Varianten von Visual Studio auftritt:

HINWEIS: Die Werte der eingestellten Ränder stimmen nicht mit den Werten der Eigenschaft *Margins* überein (aus einem Zoll Vorgabewert werden in der Anzeige 10 Millimeter und aus diesen wiederum korrekte 0,39 Zoll). Eine fragwürdige Umrechnung.

Deshalb der folgende Workaround:

Rufen Sie **vor** dem Aufruf der Dialogbox jedes Mal die folgenden Anweisungen auf:

```
pageSetupDialog1.PageSettings.Margins.Left =
   (int) (pageSetupDialog1.PageSettings.Margins.Left * 2.54);
pageSetupDialog1.PageSettings.Margins.Top =
   (int) (pageSetupDialog1.PageSettings.Margins.Top * 2.54);
pageSetupDialog1.PageSettings.Margins.Right =
   (int) (pageSetupDialog1.PageSettings.Margins.Right * 2.54);
pageSetupDialog1.PageSettings.Margins.Bottom =
    (int) (pageSetupDialog1.PageSettings.Margins.Bottom * 2.54);
pageSetupDialog1.ShowDialog();
```

Nach dem Aufruf stehen Ihnen die Seitenränder wieder in der korrekten 1/100-Zoll-Angabe zur Verfügung.

5.4.3 PrintPreviewDialog

Im Grunde ist die Verwendung des *PrintPreviewDialog* recht simpel. Nach dem Einfügen der Komponente brauchen Sie diese lediglich über die *Documents*-Eigenschaft mit der *PrintDocument*-Komponente zu verknüpfen und die Methode *ShowDialog* aufzurufen.

Bis auf die Eigenschaft *UseAntialias* (Verbessern der Anzeigequalität) können Sie kaum weitere Einstellungen vornehmen. Die Ausnahme stellt die Eigenschaft *PrintPreviewControl* dar, mit der Sie direkt das Aussehen und Verhalten der Vorschau beeinflussen können.

Beispiel 5.27: Gleichzeitige Anzeige von zehn Seiten

C#

```
printPreviewDialog1.PrintPreviewControl.Rows = 2;
printPreviewDialog1.PrintPreviewControl.Columns = 5;
printPreviewDialog1.ShowDialog();
```

HINWEIS: Mehr zur Konfiguration und Verwendung der *PrintPreviewControl*-Komponente finden Sie im folgenden Abschnitt 5.4.4.

Weiterhin dürfte die Eigenschaft *WindowState* in Zusammenhang mit der Anzeige der Dialogbox von Interesse sein. Hiermit steuern Sie die Art der Anzeige (Vollbild etc.).

Beispiel 5.28: Vollbildanzeige aktivieren

C#

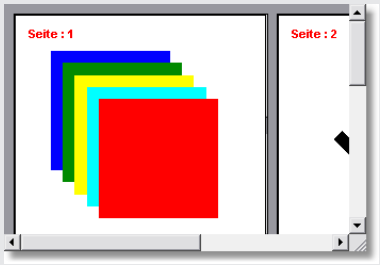
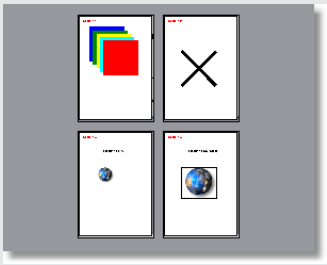
```
printPreviewDialog1.WindowState = FormWindowState.Maximized;
printPreviewDialog1.ShowDialog();
```

5.4.4 Ein eigenes Druckvorschaufenster realisieren

Wem die *PrintPreview*-Komponente zu wenig Eingriffsmöglichkeiten bietet, der kann sich mit der *PrintPreviewControl*-Komponente eine eigene Druckvorschau zusammenbauen.

Bis auf den reinen Druckvorschaubereich können Sie sich um alle Einstellungen und optischen Spielereien selbst kümmern. Die folgende Tabelle listet die wichtigsten Eigenschaften auf:

Eigenschaft	Beschreibung
AutoZoom	Ist der Wert auf *True* gesetzt, werden die Seiten so skaliert, dass die vorgegebene Anzahl von Seiten flächenfüllend dargestellt wird.

	AutoZoom = False	*AutoZoom = True*
		
BackColor	… die Hintergrundfarbe für die Druckvorschau	
Columns	… die Anzahl von Spalten, das heißt, wie viele Seiten nebeneinander dargestellt werden	
Rows	… die Anzahl von Zeilen, das heißt, wie viele Seiten untereinander dargestellt werden	
Document	… die Verknüpfung zum *PrintDocument*-Objekt	
StartPage	… die Seitenzahl der linken oberen Seite. Durch Verändern dieses Werts können Sie die weiteren Seiten anzeigen.	
Zoom	… legt explizit einen Zoomfaktor fest	

Wie Sie die *PrintPreviewControl*-Komponente im Zusammenhang verwenden, zeigt Ihnen das Praxisbeispiel in Abschnitt 5.6.1 am Kapitelende.

5.5 Drucken mit COM-Automation

Wir wollen versuchen, mithilfe der bekannten Office-Programme Druckausgaben zu realisieren. Schnell kommt der Verdacht auf, dass der Programmierer versucht, das Brett an der dünnsten Stelle anbohren zu wollen. Doch warum sollen nicht die Möglichkeiten von Office-Programmen genutzt werden, wenn doch häufig der Wunsch besteht, Reportausgaben nachträglich zu bearbeiten oder in umfangreichere Dokumentationen aufzunehmen? Nicht zuletzt bieten sich gerade die Office-Anwendungen an, wenn es um eine sinnvolle Archivierung von Dokumenten geht.

Unser Favorit ist, wie sollte es auch anders sein, Microsoft Word, ein Allround-Talent, was die Gestaltung von ansprechenden Druckausgaben angeht.

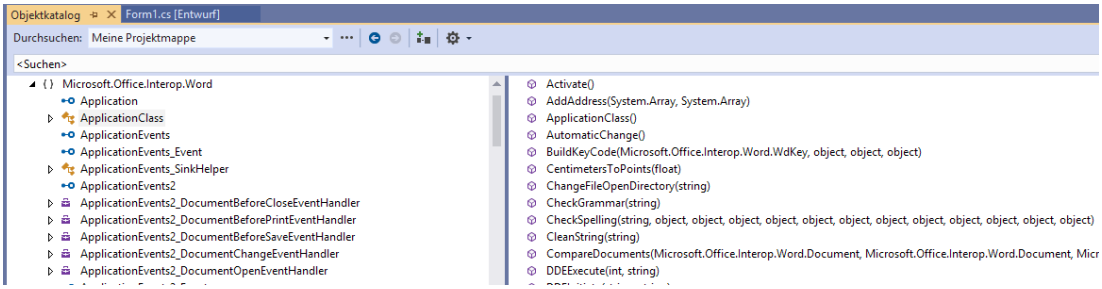
Für die im Weiteren vorgestellten Verfahren ist es sinnvoll, wenn wir kurz auf die grundlegenden Möglichkeiten und Funktionen der COM-Automation eingehen.

HINWEIS: Ein wesentlicher Nachteil der im Folgenden beschriebenen Verfahrensweise soll natürlich nicht unerwähnt bleiben. Geben Sie Ihre Anwendungen an andere Anwender weiter, muss auf dem jeweiligen Rechner natürlich auch die COM-Anwendung (Word oder Excel) installiert sein.

5.5.1 Kurzeinstieg in die COM-Automation

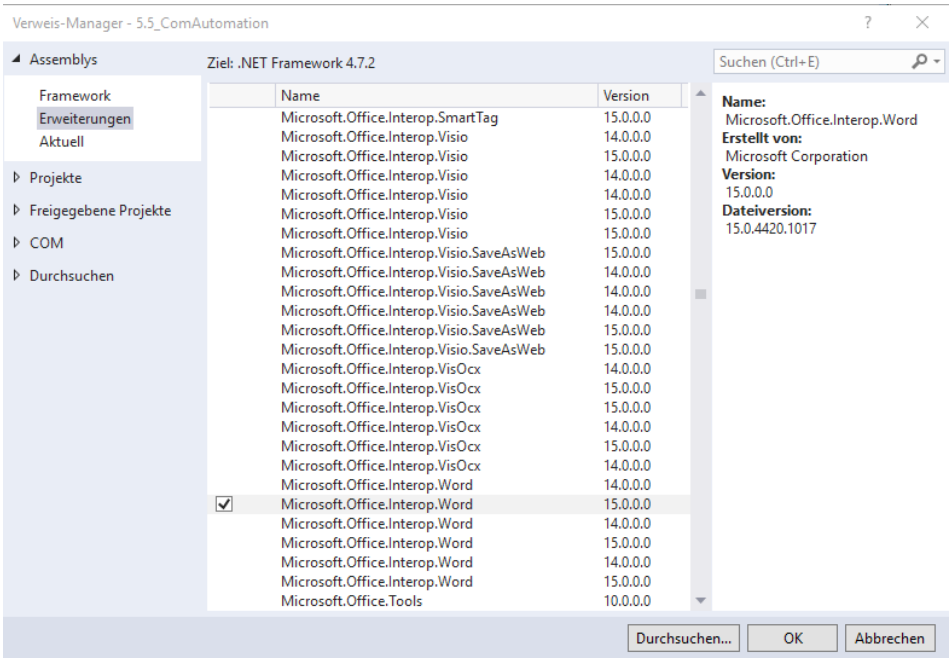
Über COM-Automation lassen sich Objekte anderer Applikationen (z. B. Word oder Excel) von Ihrem .NET-Programm quasi „fernsteuern“. Nach der Definition einer entsprechenden Objektvariablen können Sie auf Eigenschaften und Methoden dieser Objekte genauso zugreifen, als ob es sich um ein normales C#-Objekt handeln würde.

Wichtigstes Hilfsmittel für den Programmierer ist der in Visual Studio integrierte Objektkatalog (siehe folgende Abbildung).



Im Objektkatalog werden neben den Klassen alle Methoden, Eigenschaften, Ereignisse und Konstanten des jeweiligen Objekts angezeigt.

Welche Objekte angezeigt werden, hängt von den Verweisen ab, die Sie unter *Projekt/Verweise hinzufügen…* eingebunden haben:



Programmieren der OLE-Automation

Das Grundprinzip besteht darin, dass Sie in C# eine Instanz der gewünschten Klasse erzeugen. Mit diesem Objekt können Sie dann wie mit jedem anderen Objekt arbeiten.

HINWEIS: Wer bereits mit älteren C#-Versionen (< 4.0) gearbeitet hat, wird sich sicher noch an die grauenhafte Arbeit mit dem Erzeugen von Objekten und der Übergabe von optionalen Parametern erinnern. Vergessen Sie dies alles, seit C# 4.0 können Sie auf derartige Handstände verzichten.

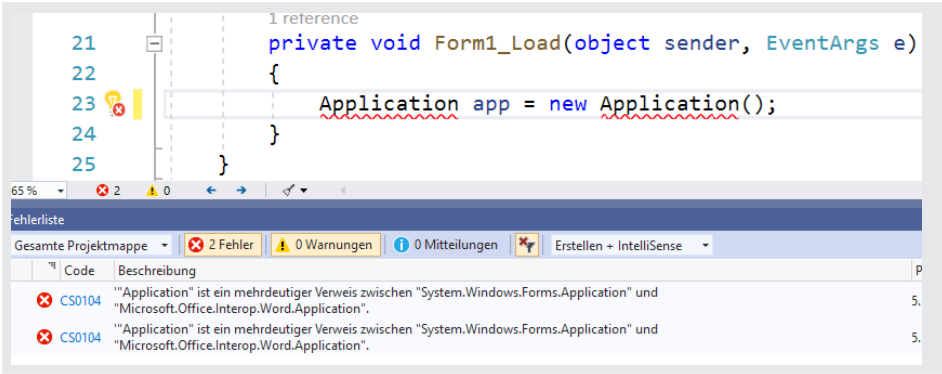
Voraussetzung für das Erstellen einer Instanz ist ein Verweis auf die entsprechende Klasse. Um neue Verweise zu erstellen, müssen Sie unter **Projekt | Verweise hinzufügen...**

- die gewünschte COM-Klassenbibliothek (z. B. *Microsoft Word 15.0 Object Library*) auswählen
- oder alternativ eine der vorhandenen PIAs (*Primary Interop Assemblies*) nutzen.

Im vorliegenden Fall nutzen wir die PIA *Microsoft.Office.Interop.Word* (siehe obige Abbildung).

HINWEIS: Binden Sie den neuen Namespace (*using*) direkt ein, kann es zu Überschneidungen der COM-Objektnamen mit den C#-Objekten kommen.

Beispiel 5.29: Hier ist nicht eindeutig, welches *Application* gemeint ist.



Besser Sie verwenden in diesem Fall einen Aliasnamen für die Assembly, wie es das folgende Beispiel zeigt:

Beispiel 5.30: Verwendung Aliasnamen

C#

Erstellen einer Objektvariablen *myWord* als Instanz des *Word.Application*-Objekts:

```
using Word = Microsoft.Office.Interop.Word;
```

```
namespace WindowsFormsApplication3
{
...
     var word = new Word.Application();
...
```

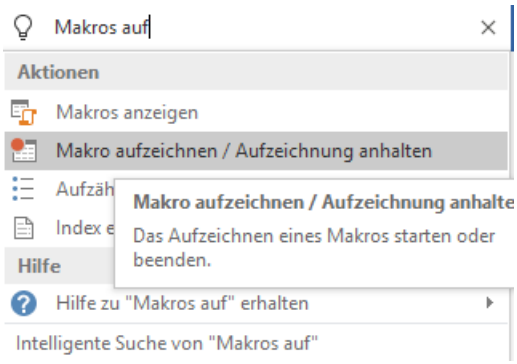
5.5.2 Drucken mit Microsoft Word

Verwenden Sie Word für die Druckausgabe, können Sie zwei verschiedene Varianten einsetzen:

- Sie entwerfen die komplette Seite mit Word und fügen an den relevanten Stellen sogenannte Platzhalter (Formularfelder) ein. Diese werden später aus dem C#-Programm heraus gezielt aufgerufen und mit neuen Inhalten gefüllt. Der Vorteil dieser Variante: Das Word-Dokument kann quasi wie eine Vorlage genutzt werden, der Aufwand ist minimal. Nachteil: Sie müssen die Datei zur Laufzeit in den Word-Editor laden.
- Der komplette Report bzw. das komplette Word-Dokument wird zur Laufzeit aus C# heraus generiert. Vorteil: Das Erstellen von Listen ist mit dieser Variante wesentlich einfacher als mit vorgefertigten Dokumenten. Nachteil: jede Menge Quellcode.

Der Nachteil der zweiten Variante kann jedoch schnell wieder wettgemacht werden, lassen Sie einfach Word für sich arbeiten.

Was ist gemeint? Öffnen Sie Word und geben Sie im Suchfenster einfach Makros aufzeichnen ein.



Alle weiteren Aktionen, die Sie durchführen (Text eingeben, formatieren, Tabellen erstellen etc.) werden durch den Makrorekorder aufgezeichnet. Sie müssen nur noch das aufgezeichnete Makro in Ihr C#-Programm einfügen und „geringfügig“ anpassen.

Beispiel 5.31: Transformieren eines aufgezeichneten Word-Makros in C#

C#

Sie erstellen bei eingeschaltetem Makrorekorder ein neues Dokument und geben Sie eine 16 Punkt große Überschrift ein. Das Resultat ist folgender Bandwurm (Word-VBA):

```
Sub Makro1()
  Documents.Add Template:=»Normal», NewTemplate:=False, DocumentType:=0 :=False
    Selection.Font.Size = 16
    Selection.Font.Bold = wdToggle
    Selection.Font.BoldBi = wdToggle
    Selection.TypeText Text:="Überschrift"
End Sub
```

Aus diesem Quellcode-Haufen filtern wir uns erst einmal die relevanten Daten heraus:

```
Sub Makro1()
  Documents.Add
  Selection.Font.Size = 16
  Selection.TypeText Text:="Überschrift"
End Sub
```

Das sieht doch schon viel freundlicher aus, das Resultat beider Makros ist dasselbe. Kopieren Sie nun den Quellcode in Ihre C#-Anwendung. Erster Schritt ist jetzt das Erzeugen einer Word-Instanz bzw. einer *Word.Application*-Instanz:

```
using Word = Microsoft.Office.Interop.Word;
...
private void Button1_Click(object sender, EventArgs e)
{
   var word = new Word.Application();
```

Word muss extra einblendet werden:

```
   word.Visible = true;
```

Und hier kommen die eigentlichen Anweisungen:

```
   word.Documents.Add();
   word.Selection.Font.Size = 16;
   word.Selection.TypeText("Überschrift");
}
```

Der ehemalige Makrocode ist fett hervorgehoben.

HINWEIS: Das obige C#-Listing ist erst ab C# 4.0 lauffähig. In älteren Versionen müssen Sie sich damit abfinden, dass Sie immer alle Parameter an die jeweiligen Methoden übergeben müssen. Dies erfordert zum Teil einen beträchtlichen Mehraufwand und ein intensives Studium der VBA-Hilfe.

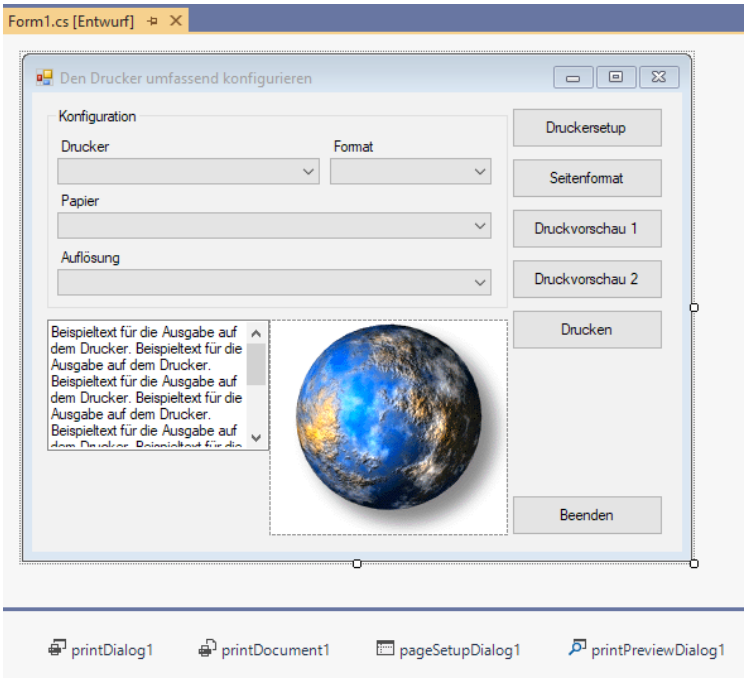
5.6 Praxisbeispiele

5.6.1 Den Drucker umfassend konfigurieren

Das Ziel dieses Beispiels ist eine umfassende Darstellung des Zusammenspiels der einzelnen Druckerkomponenten sowie deren Konfiguration per Code bzw. per Dialogbox. Insgesamt zehn Beispielseiten verdeutlichen die verschiedenen Möglichkeiten der Gestaltung des Druckbilds.

Oberfläche (Hauptformular Form1)

Entwerfen Sie eine Oberfläche entsprechend folgender Abbildung:

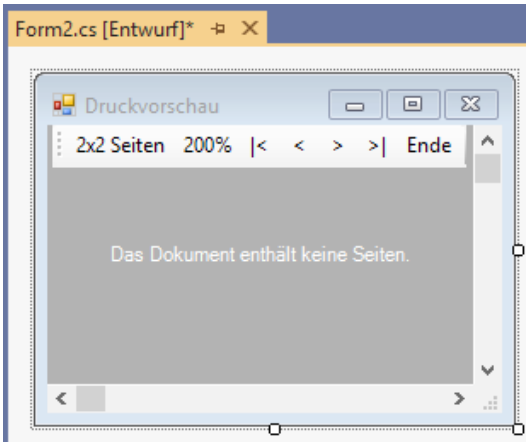


Verknüpfen Sie die vier nicht sichtbaren Komponenten (siehe unterer Bildrand) über die *Documents*-Eigenschaft mit dem *printDocument1*.

Sowohl die *TextBox* als auch die *PictureBox* dienen uns lediglich als Container für einen zu druckenden Text bzw. eine zu druckende Grafik.

Oberfläche (Druckvorschau Form2)

Mit der folgenden Oberfläche wollen wir keinen Schönheitspreis gewinnen, es geht lediglich um die Darstellung des Grundprinzips. Welche Komponenten Sie für die Oberflächengestaltung nutzen, bleibt Ihrer Fantasie überlassen. Wichtig ist vor allem das *PrintPreview*-Control:



Quelltext (Form1)

```
using System.Drawing.Printing;

public partial class Form1 : Form
{
```

Eine globale Variable erleichtert uns die Anzeige bzw. den Druck der richtigen Seite:

```
    private int page;
```

Die folgende Routine aktualisiert die *ComboBox*en nach Änderungen über die Standarddialoge:

```
    private void Aktualisieren()
    {
```

Der aktuelle Drucker:

```
        comboBox1.Text = printDocument1.PrinterSettings.PrinterName;
```

Die verschiedenen Papierformate:

```
        comboBox2.Items.Clear();
        foreach (PaperSize ps in printDocument1.PrinterSettings.PaperSizes)
        {
           comboBox2.Items.Add(ps);
        }
        comboBox2.Text = printDocument1.DefaultPageSettings.PaperSize.ToString();
```

Die Seitenausrichtung:

```
        if (printDocument1.DefaultPageSettings.Landscape)
        {
           comboBox3.SelectedIndex = 0;
         }
         else
```

```
        {
         comboBox3.SelectedIndex = 1;
        }
```

Die Druckauflösung:

```
        comboBox4.Items.Clear();
        foreach (PrinterResolution res in
                 printDocument1.PrinterSettings.PrinterResolutions)
        {
           comboBox4.Items.Add(res);
        }
        comboBox4.Text =
                 printDocument1.DefaultPageSettings.PrinterResolution.ToString();
    }
```

Beim Programmstart füllen wir zunächst *comboBox1* mit den Namen der verfügbaren Drucker und aktualisieren die Anzeige:

```
    private void Form1_Load(object sender, EventArgs e)
    {
       foreach(String s in PrinterSettings.InstalledPrinters)
       {
              comboBox1.Items.Add(s);
        }
        Aktualisieren();
    }
```

Die Anzeige des Standard-Druckerdialogs:

```
    private void Button2_Click(object sender, EventArgs e)
    {
       printDialog1.ShowDialog();
       Aktualisieren();
    }
```

Das Einrichten der Seite (Fehler bei der Umrechnung beachten!):

```
    private void Button3_Click(object sender, EventArgs e)
    {
        pageSetupDialog1.PageSettings.Margins.Left =
                 (int)(pageSetupDialog1.PageSettings.Margins.Left * 2.54);
        pageSetupDialog1.PageSettings.Margins.Top =
                 (int)(pageSetupDialog1.PageSettings.Margins.Top * 2.54);
        pageSetupDialog1.PageSettings.Margins.Right =
                 (int)(pageSetupDialog1.PageSettings.Margins.Right * 2.54);
        pageSetupDialog1.PageSettings.Margins.Bottom =
                 (int)(pageSetupDialog1.PageSettings.Margins.Bottom * 2.54);
        pageSetupDialog1.ShowDialog();
        Aktualisieren();
    }
```

Start des Druckvorgangs bzw. der Druckvorschau:

```
    private void printDocument1_BeginPrint(object sender,
                                           System.Drawing.Printing.PrintEventArgs e)
    {
       page = 1;
```

```
        printDocument1.DocumentName = "Mein erstes Testdokument";
    }
```

Das eigentliche Drucken der Seiten passiert wie immer im *PrintPage*-Event unseres *PrintDocument*-Objekts:

```
    private void printDocument1_PrintPage(object sender,
                                System.Drawing.Printing.PrintPageEventArgs e)
    {
```

Eine Zufallszahl für optische Spielereien:

```
        Random rnd = new Random();
```

Einen *Pen* definieren:

```
        Pen p = new Pen(System.Drawing.Color.Black, 1);
```

Eine Variable für den einfacheren Zugriff auf das *Graphics*-Objekt:

```
        Graphics g = e.Graphics;
```

Die aktuell zu druckende Seite:

```
        int printPage = 0;
```

Umschalten in Millimeter:

```
        g.PageUnit = GraphicsUnit.Millimeter;
```

Berücksichtigung des Druckbereichs:

```
        switch (e.PageSettings.PrinterSettings.PrintRange)
        {
            case PrintRange.SomePages:
                 printPage = page + e.PageSettings.PrinterSettings.FromPage - 1;
                 break;
            case PrintRange.AllPages:
                 printPage = page;
                 break;
        }
```

Drucken der jeweiligen Seite (1 bis 10):

```
        switch (printPage)
        {
            case 1:
```

Ein paar Rechtecke (10 × 10 cm):

```
                g.FillRectangle(new SolidBrush(Color.Blue), 30, 30, 100, 100);
                g.FillRectangle(new SolidBrush(Color.Green), 40, 40, 100, 100);
                g.FillRectangle(new SolidBrush(Color.Yellow), 50, 50, 100, 100);
                g.FillRectangle(new SolidBrush(Color.Cyan), 60, 60, 100, 100);
                g.FillRectangle(new SolidBrush(Color.Red), 70, 70, 100, 100);
                break;
             case 2:
```

Einige Linien auf Seite 2:

```
                g.DrawLine(new Pen(Color.Black, 10), 50, 100, 150, 200);
                g.DrawLine(new Pen(Color.Black, 10), 50, 200, 150, 100);
                break;
            case 3:
```

Ausgabe der Grafik in Originalgröße:

```
                g.DrawString("Grafik 100%", new Font("Arial", 10, FontStyle.Bold,
                             GraphicsUnit.Millimeter), Brushes.Black, 70, 50);
                g.DrawImage(pictureBox1.Image, 50, 100);
                break;
            case 4:
```

Skalieren der Grafik auf 10 cm Breite:

```
                g.DrawString("Grafik 10cm breit", new Font("Arial", 10,
                             FontStyle.Bold, GraphicsUnit.Millimeter),
                             Brushes.Black, 70, 50);
                g.DrawImage(pictureBox1.Image, 50, 100, 100,
                         pictureBox1.Image.Height * 100 %
pictureBox1.Image.Width);
                g.DrawRectangle(new Pen(Color.Black, 0.1f), 50, 100, 100,
                           pictureBox1.Image.Height * 100 %
                           pictureBox1.Image.Width);
                break;
             case 5:
```

Anzeige der Seitenränder:

```
                g.DrawString("Seitenränder", new Font("Arial", 10, FontStyle.Bold,
                             GraphicsUnit.Millimeter), Brushes.Black, 70, 50);
                g.PageUnit = GraphicsUnit.Display;
                g.DrawRectangle(new Pen(Color.Black), e.MarginBounds);
                g.PageUnit = GraphicsUnit.Millimeter;
                break;
            case 6:
```

Ausgabe von Text (linksbündig):

```
                RectangleF rect = new RectangleF();
                rect = (RectangleF)e.MarginBounds;
                g.PageUnit = GraphicsUnit.Display;
                g.DrawString(TextBox1.Text, new Font("Arial", 10, FontStyle.Bold,
                             GraphicsUnit.Millimeter), Brushes.Black, rect);
                g.PageUnit = GraphicsUnit.Millimeter;
                break;
            case 7:
```

Ausgabe von Text (zentriert):

```
               RectangleF rect1 = new RectangleF();
               StringFormat format = new StringFormat();
               format.Alignment = StringAlignment.Center;
               rect1 = (RectangleF)e.MarginBounds;
               g.PageUnit = GraphicsUnit.Display;
```

```
            g.DrawString(TextBox1.Text, new Font("Arial", 10, FontStyle.Bold,
                  GraphicsUnit.Millimeter), Brushes.Black, rect1, format);
            g.PageUnit = GraphicsUnit.Millimeter;
            break;
        case 8:
```

Ausgabe von zufälligen Linien über den gesamten Blattbereich:

```
            g.DrawString("Zufallslinien ohne Clipping", new Font("Arial", 10,
              FontStyle.Bold, GraphicsUnit.Millimeter), Brushes.White, 70, 50);
             g.PageUnit = GraphicsUnit.Display;
             for (int i = 0; i <= 500; i++)
             {
                        g.DrawLine(p, 0, 0, rnd.Next(e.PageBounds.Width),
                                            rnd.Next(e.PageBounds.Height));
             }
             break;
        case 9:
```

HINWEIS: Vergleichen Sie den Ausdruck mit der Druckvorschau, werden Sie feststellen, dass die Druckvorschau die physikalischen Seitenränder nicht berücksichtigt.

Berücksichtigung der Seitenränder bei der Druckausgabe:

```
            g.DrawString("Zufallslinien mit Clipping", new Font("Arial", 10,
                FontStyle.Bold, GraphicsUnit.Millimeter), Brushes.White, 70, 50);
            g.PageUnit = GraphicsUnit.Display;
            g.SetClip(e.MarginBounds);
            for (int i = 0; i <=500; i++)
            {
                g.DrawLine(p, 0, 0, rnd.Next(e.PageBounds.Width),
                                    rnd.Next(e.PageBounds.Height));
            }
            break;
        case 10:
```

Berücksichtigung der Seitenränder sowie Verschieben des Offsets bei der Druckausgabe:

```
             g.PageUnit = GraphicsUnit.Display;
             g.SetClip(e.MarginBounds);
             g.TranslateTransform(e.MarginBounds.Left, e.MarginBounds.Top);
             for (int i = 0; i <= 500; i++)
             {
                       g.DrawLine(p, 0, 0, rnd.Next(e.PageBounds.Width),
                                           rnd.Next(e.PageBounds.Height));
             }
             break;
    }
```

Seitennummer einblenden:

```
    g.DrawString($"Seite: {printPage}", new Font("Arial", 10, FontStyle.Bold,
                  GraphicsUnit.Millimeter), Brushes.Red, 10, 10);
```

Vorbereiten der nächsten Seite:

```
        page++;
```

Berücksichtigung des Druckbereichs:

```
        switch (e.PageSettings.PrinterSettings.PrintRange)
        {
            case PrintRange.SomePages:
                e.HasMorePages = (printpage < e.PageSettings.PrinterSettings.ToPage);
                break;
            case PrintRange.AllPages:
                e.HasMorePages = (page < 12);
                break;
        }
    }
```

Aktuellen Drucker wechseln:

```
    private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
    {
       printDocument1.PrinterSettings.PrinterName = comboBox1.Text;
       Aktualisieren();
    }
```

Seitenausrichtung ändern:

```
    private void comboBox3_SelectedIndexChanged(object sender, EventArgs e)
    {
        printDocument1.DefaultPageSettings.Landscape =
             (comboBox3.SelectedIndex == 0);
        Aktualisieren();
    }
```

Papierformat ändern:

```
    private void comboBox2_SelectedIndexChanged(object sender, EventArgs e)
    {
        printDocument1.DefaultPageSettings.PaperSize =
printDocument1.PrinterSettings.PaperSizes[comboBox2.SelectedIndex];
    }
```

Druckauflösung ändern:

```
    private void comboBox4_SelectedIndexChanged(object sender, EventArgs e)
    {
        printDocument1.DefaultPageSettings.PrinterResolution =

printDocument1.PrinterSettings.PrinterResolutions[comboBox4.SelectedIndex];
    }
```

Druckvorschau anzeigen (Vollbild):

```
    private void Button1_Click(object sender, EventArgs e)
    {
         printPreviewDialog1.WindowState = FormWindowState.Maximized;
```

```
        printPreviewDialog1.ShowDialog();
    }
```

Die eigene Druckvorschau anzeigen:

```
    private void Button4_Click(object sender, EventArgs e)
    {
        Form2 f2 = new Form2();
        f2.printPreviewControl1.Document = printDocument1;
        f2.ShowDialog();
    }
```

Den Druckvorgang starten:

```
    private void Button5_Click(object sender, EventArgs e)
    {
        printDocument1.Print();
    }
}
```

Quelltext (Form2)

Im Formular *Form2* geht es im Wesentlichen nur um die Konfiguration der *PrintPreviewControl*-Komponente.

Die Navigation zwischen den Seiten:

```
public partial class Form2 : Form
{
    ...
```

Nächste Seite:

```
        private void toolStripButton4_Click(object sender, EventArgs e)
        {
            printPreviewControl1.StartPage++;
        }
```

Vorhergehende Seite:

```
        private void toolStripButton3_Click(object sender, EventArgs e)
        {
            if (printPreviewControl1.StartPage > 0)
            {
                printPreviewControl1.StartPage--;
            }
        }
```

Erste Seite:

```
        private void toolStripButton6_Click(object sender, EventArgs e)
        {
            printPreviewControl1.StartPage = 0;
        }
```

Letzte Seite (setzen Sie einfach einen Wert, der groß genug ist):

```
        private void toolStripButton7_Click(object sender, EventArgs e)
        {
            printPreviewControl1.StartPage = 999;
        }
```

Seite auf 200 Prozent skalieren:

```
        private void toolStripButton2_Click(object sender, EventArgs e)
        {
            printPreviewControl1.AutoZoom = false;
            printPreviewControl1.Zoom = 200;
        }
```

Vier Seiten gleichzeitig anzeigen (eingepasst in die Komponente):

```
        private void toolStripButton1_Click(object sender, EventArgs e)
        {
            printPreviewControl1.Columns = 2;
            printPreviewControl1.Rows = 2;
            printPreviewControl1.AutoZoom = true;
        }
}
```

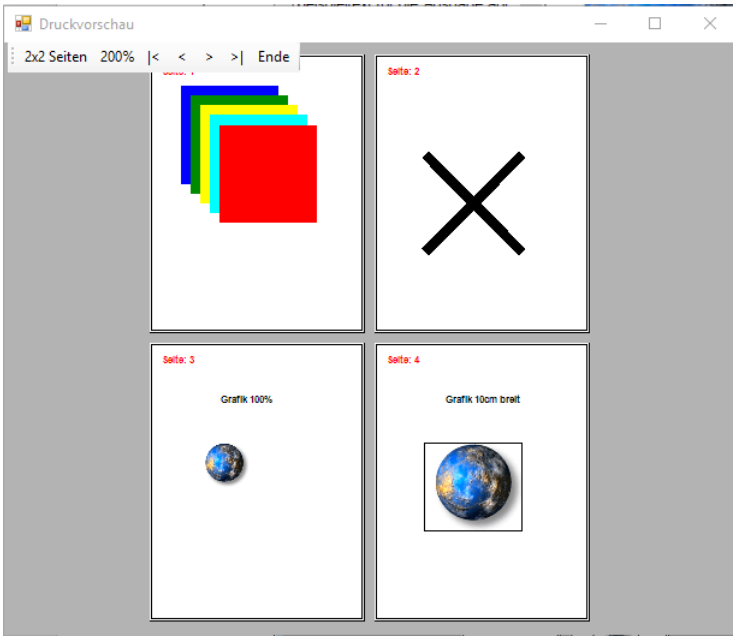
Test

Nach dem Programmstart sollten alle Druckerparameter korrekt in den *ComboBoxen* angezeigt werden.

HINWEIS: Testen Sie, was passiert, wenn Sie Änderungen in den *ComboBoxen* bzw. mithilfe der Druckerdialoge vornehmen.

Nun haben Sie die Möglichkeit, sich die zehn verschiedenen Druckseiten in einer der beiden Druckvorschauen zu betrachten oder zu Papier zu bringen:

Als Beispiel hier unsere „selbst gebastelte“ Druckvorschau in Aktion:

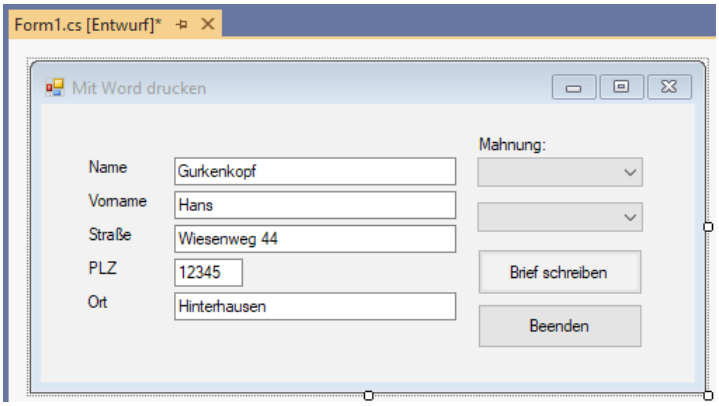


5.6.2 Druckausgabe mit Word

Eines der „dankbarsten Opfer“ für COM-Automation ist nach wie vor Word für Windows. Unser Beispiel zeigt Ihnen, wie Sie aus einem C#-Programm heraus ein neues Word-Dokument erstellen, Kopf- und Fußzeilen einfügen und Daten übertragen. (Das Beispiel lässt sich problemlos so anpassen, dass die Daten statt aus den Eingabefeldern gleich aus einer Datenbank kommen.)

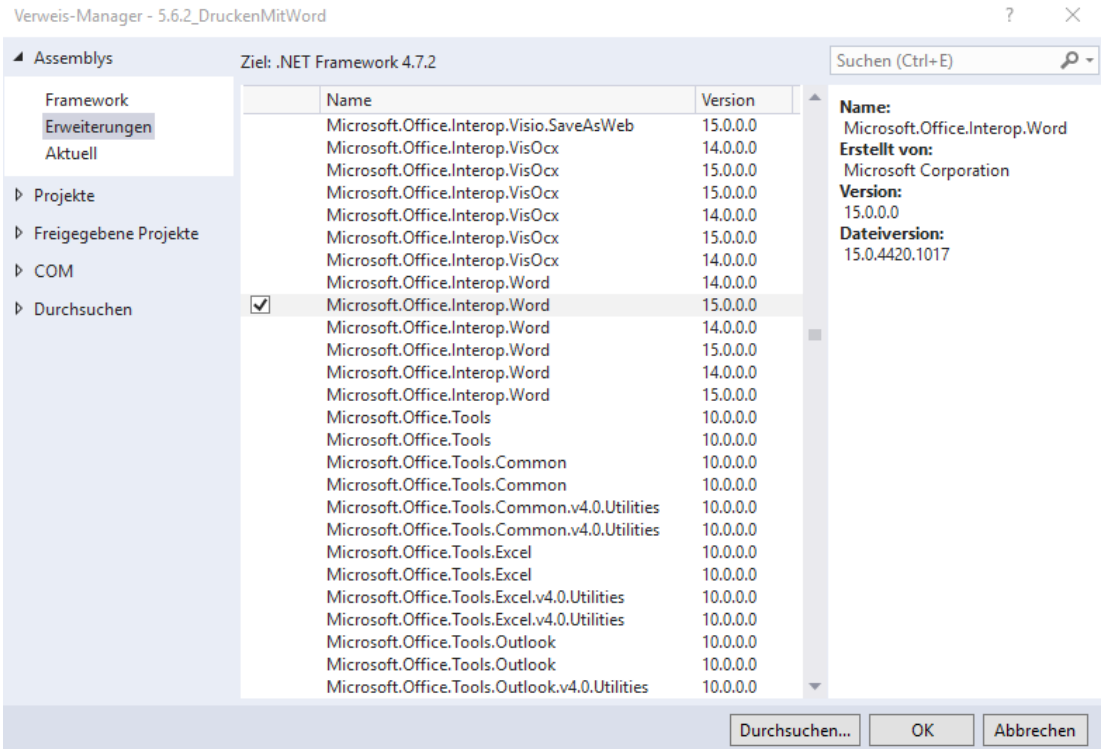
Oberfläche

Den Grundaufbau können Sie der folgenden Abbildung entnehmen:



In der oberen *ComboBox* finden sich drei Einträge: „1. Mahnung“ … „3. Mahnung“, die Sie im Eigenschaftenfenster über die *Items*-Auflistung hinzufügen. In der unteren *ComboBox* wird zwischen „Herr“ und „Frau“ unterschieden.

Damit Sie problemlos die Word-Objekte und -Konstanten verwenden können, müssen Sie noch einen Verweis auf die *Microsoft.Office.Interop.Word*-Assembly (eine *PIA = Primary Interop Assembly*) einrichten (**Projekt | Verweis hinzufügen…**).



Quelltext

```
using Word = Microsoft.Office.Interop.Word;

public partial class Form1 : Form
{
```

Es geht los:

```
        private void Button1_Click(object sender, EventArgs e)
        {
```

Grundlage für die Verbindung zu Word ist eine implizit typisierte lokale Variable vom Typ *ApplicationClass*:

```
            var wordapp = new Word.Application();
```

Der Ablauf ist mit wenigen Worten erklärt: Nach der Initialisierung der Variablen können Sie alle Methoden des *Application*-Objekts verwenden. Bevor Sie lange in der Word-Dokumentation herumstochern, ist es sinnvoller, ein Word-Makro aufzuzeichnen und dieses entsprechend zu modifizieren. Zum einen haben Sie gleich die korrekte Syntax, zum anderen sparen Sie sich jede Menge Arbeit.

Bei Problemen kneifen wir an dieser Stelle:

```
          if (wordapp == null)
          {
              MessageBox.Show("Konnte keine Verbindung zu Word herstellen!");
              return;
          }
```

Word sichtbar machen (standardmäßig wird Word nicht angezeigt):

```
          wordapp.Visible = true;
```

Ein neues Dokument erzeugen:

```
          wordapp.Documents.Add();
          if (wordapp.ActiveWindow.View.SplitSpecial != 0)
          {
              wordapp.ActiveWindow.Panes[2].Close();
          }
          if (((int) wordapp.ActiveWindow.ActivePane.View.Type == 1)|
                        ((int) wordapp.ActiveWindow.ActivePane.View.Type == 2)|
                        ((int)wordapp.ActiveWindow.ActivePane.View.Type == 5))
          {
              wordapp.ActiveWindow.ActivePane.View.Type =
                                    Word.WdViewType.wdPrintView;
          }
```

Kopfzeile erzeugen:

```
           wordapp.ActiveWindow.ActivePane.View.SeekView =
                   Word.WdSeekView.wdSeekCurrentPageHeader;
           wordapp.Selection.Font.Name = "Times New Roman";
           wordapp.Selection.Font.Size = 12;
           wordapp.Selection.Font.Bold = 1;
           wordapp.Selection.ParagraphFormat.Alignment =
                   Word.WdParagraphAlignment.wdAlignParagraphCenter;
           wordapp.Selection.TypeText(
             "Kohlenhandel Brikett-GmbH & Co.-KG. - Holzweg 16 - 54633
Steinhausen");
```

Fußzeile erzeugen:

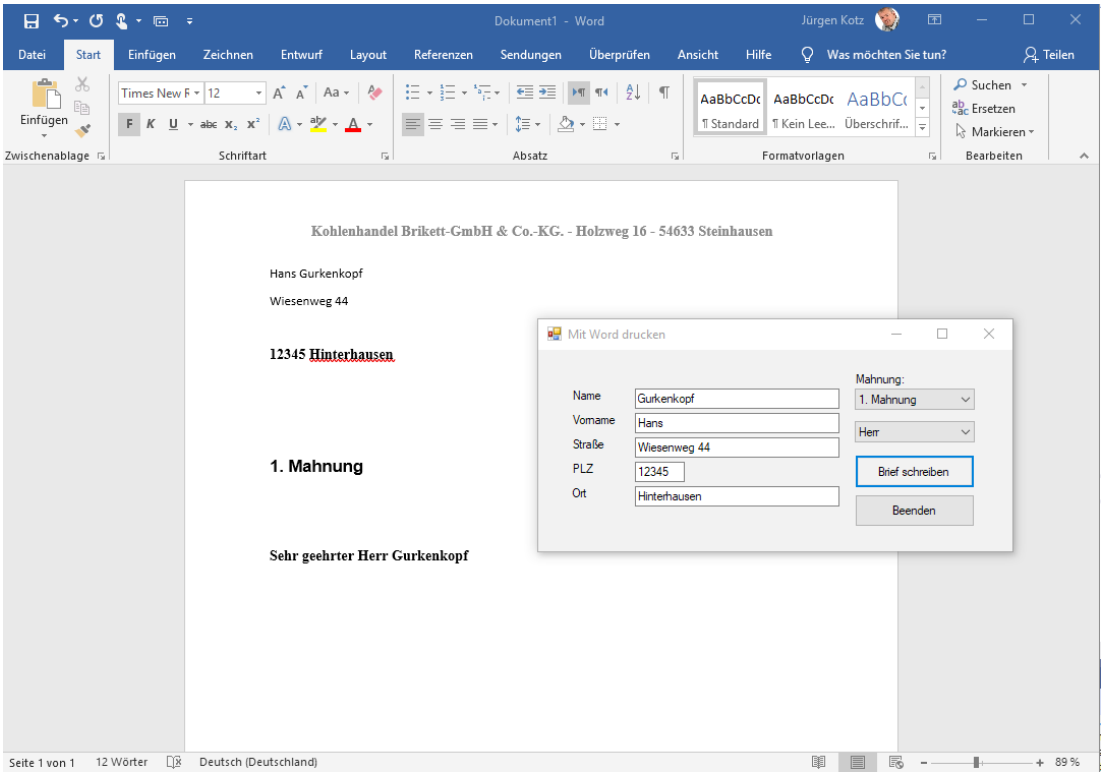
```
           wordapp.ActiveWindow.ActivePane.View.SeekView =
                  Word.WdSeekView.wdSeekCurrentPageFooter;
           wordapp.Selection.TypeText(
             "Bankverbindung: Stadtsparkasse Steinhausen BLZ 123456789 KtoNr. " +
                 "782972393243");
```

In den Textteil wechseln und die Adresse eintragen:

```
            wordapp.ActiveWindow.ActivePane.View.SeekView =
                    Word.WdSeekView.wdSeekMainDocument;
            wordapp.Selection.TypeText(textBox2.Text + " " + textBox1.Text);
            wordapp.Selection.TypeParagraph();
            wordapp.Selection.TypeText(textBox3.Text);
            wordapp.Selection.TypeParagraph();
            wordapp.Selection.Font.Name = "Times New Roman";
            wordapp.Selection.Font.Size = 12;
            wordapp.Selection.Font.Bold = 1;
            wordapp.Selection.TypeText(textBox4.Text + " " + textBox5.Text);
            wordapp.Selection.TypeParagraph();
            wordapp.Selection.TypeParagraph();
            wordapp.Selection.Font.Name = "Arial";
            wordapp.Selection.Font.Size = 14;
            wordapp.Selection.Font.Bold = 1;
            wordapp.Selection.TypeText(comboBox1.Text);
            wordapp.Selection.TypeParagraph();
            wordapp.Selection.Font.Name = "Times New Roman";
            wordapp.Selection.Font.Size = 12;
            wordapp.Selection.Font.Bold = 1;

            if (comboBox2.SelectedIndex == 0)
            {
                wordapp.Selection.TypeText("Sehr geehrter Herr " + textBox1.Text);
            }
            else
            {
                wordapp.Selection.TypeText("Sehr geehrte Frau " + textBox1.Text);
            }
        }
    }
```

Test

Starten Sie das Programm, füllen Sie die Maske aus und übertragen Sie die Daten in ein Word-Dokument!



Anmerkung ab .NET 4.0

Vielleicht hat mancher Leser bei Verwendung der mittlerweile berühmt berüchtigten PIA[2]s schlechte Erfahrungen gesammelt. So erscheint schnell mal eine Fehlermeldung, wenn auf dem Ziel-PC nicht die entsprechenden Assemblies installiert sind.

Hintergrund derartiger „Problemchen“ ist die Tatsache, dass beim Kompilieren eines Projekts mit eingebundenen PIAs nur Verweise auf eben diese Assemblies eingebunden werden.

Die Assemblies müssen also gegebenenfalls mitgegeben werden, was den Umfang Ihres Projekts jedoch schnell aufblähen kann.

Seit .NET 4 besteht die Möglichkeit, die verwendeten Interop-Typen in Ihre Assembly einzubetten. Damit entfällt auch die Weitergabe bzw. Installation der PIAs auf den Ziel-PCs.

HINWEIS: Stellen Sie alte Projekte um, müssen Sie auch die Zielplattform auf .NET 4 anpassen, andernfalls steht Ihnen dieses Feature nicht zur Verfügung!

[2] *Primary Interop Assembly*

Zum Einbetten genügt es, wenn Sie die Eigenschaft „Interop-Typen einbetten“ der jeweiligen PIA auf *true* setzen (dies ist bei neuen Projekten standardmäßig der Fall).

Wer jetzt befürchtet, dass sich die Anwendung massiv aufbläht, liegt falsch, es werden wirklich **nur** die unmittelbar benötigten Typen in die Assembly aufgenommen.

6 Windows Forms-Datenbindung

Sie haben im Buch in den Kapiteln 18 bereits die Grundlagen von ADO.NET kennengelernt und wissen, wie man Datenbanken abfragen und aktualisieren kann. Um Ein- und Ausgaben zu realisieren, hatten wir dort bereits mit einfacher Datenbindung gearbeitet (meist unter Verwendung des *DataGridView*). An dieser Stelle wollen wir uns dieser Thematik im Detail widmen.

6.1 Prinzipielle Möglichkeiten

Datenbindung ist ganz allgemein die Verknüpfung zwischen einer Steuerelementeigenschaft und einer Datenquelle. In Abhängigkeit von der Beantwortung der beiden Fragen

- *„Will ich die Datenbindung manuell oder mit Drag & Drop-Assistentenunterstützung programmieren?“* oder
- *„Sollen komplette Listen bzw. Tabelleninhalte oder nur einzelne Felder angebunden werden?“*

… kann man das Gebiet der Datenbindung grob in vier Bereiche aufteilen:

- Manuelle Datenbindung an einfache Datenfelder
- Manuelle Datenbindung an Listen/Tabelleninhalte
- Entwurfszeit-Datenbindung an ein typisiertes DataSet
- Drag&Drop-Datenbindung

Wer wenig Code schreiben will, wird weitestgehend die Hilfe der Assistenten in Anspruch nehmen. Der solide Handwerker, der lieber etwas mehr Code schreibt und dafür aber die volle Kontrolle über sein Programm behält, wird die manuelle Datenbindung bevorzugen.

6.2 Manuelle Bindung an einfache Datenfelder

Bestimmte Eigenschaften vieler Windows Forms-Controls lassen sich an eine Datenquelle binden. Damit ändert der Wert in der Datenquelle den Wert der gebundenen Eigenschaft und umgekehrt.

Beispiel 6.1: Ein DataSet *ds* enthält die Tabelle „Employees“. Eine *TextBox* soll an das Feld „LastName“ angebunden werden.

C#

Fügen Sie von der Toolbox eine *BindingSource*-Komponente zum Formular hinzu.

```
bindingSource1.DataSource = ds;
bindingSource1.DataMember = "Employees";
```

Die *Text*-Eigenschaft der *TextBox* wird angebunden:

```
textBox1.DataBindings.Add("Text", bindingSource1, "LastName");
```

Um die Datensätze weiterblättern zu können, brauchen Sie nur noch eine *BindingNavigator*-Komponente hinzuzufügen, deren *BindingSource*-Eigenschaft Sie auf *bindingSource1* setzen.

6.2.1 BindingSource erzeugen

Die mit .NET 2.0 eingeführte *BindingSource* löste die veralteten (aber natürlich nach wie vor unterstützten) Klassen *BindingManagerBase* bzw. *CurrencyManager* ab. Eine *BindingSource* kapselt die Datenquelle des Formulars, sie schiebt sich quasi als zusätzliche Schicht zwischen Datenquelle und Anzeigecontrols. Mittels *DataSource*- bzw. *DataMember*-Eigenschaft wird eine *BindingSource* mit der Datenquelle verbunden.

Beispiel 6.2: Verschiedene Varianten zum Erzeugen einer *BindingSource* und ihrer Verbindung mit der Tabelle „Employees“ eines *DataSet*-Objekts *ds*

C#

```
BindingSource bs = new BindingSource();
bs.DataSource = ds;
bs.DataMember = "Employees";
```

oder

```
BindingSource bs = new BindingSource(ds, "Employees");
```

oder

```
DataTable dt = ds.Tables["Employees"];
BindingSource bs = new BindingSource();
bs.DataSource = dt;
```

oder

```
DataView dv = ds.Tables["Employees"].DefaultView;
BindingSource bs = new BindingSource();
bs.DataSource = dv;
```

6.2.2 Binding-Objekt

Ein *Binding*-Objekt ermöglicht die einfache Bindung zwischen dem Wert einer Objekteigenschaft und dem Wert einer Steuerelementeigenschaft. Bei der Instanziierung sind drei Parameter zu übergeben:

- die zu bindende Eigenschaft des Controls (z. B. *Text*),
- die Datenquelle, an die zu binden ist (*BindingSource*, *DataSet*, *DataTable*, *DataView*),
- das Feld innerhalb der Datenquelle, das angebunden werden soll (z. B. *Vorname*).

Beispiel 6.3: Die Steuerelementeigenschaft *Text* wird an die Eigenschaft *Geburtsdatum* der Personal-Tabelle gebunden.

C#

```
BindingSource bs = new BindingSource(ds, "Employees");
Binding b1 = new Binding("Text", bs, "BirthDate");
```

6.2.3 DataBindings-Collection

Die Datenanbindung für einfache Steuerelemente, wie z. B. *Label* oder *TextBox*, wird durch Hinzufügen von *Binding*-Objekten zur *DataBindings*-Auflistung des Steuerelements komplettiert. Der *Add*-Methode sind entweder ein komplettes *Binding*-Objekt oder aber dessen drei Argumente zu übergeben.

Beispiel 6.4: (Fortsetzung)

C#

Das im Beispiel 6.3 erzeugte *Binding*-Objekt wird zur *DataBindings*-Collection einer *TextBox* hinzugefügt:

```
textBox1.DataBindings.Add(b1);
```

Eine Überladung der *Add*-Methode, die ohne explizit erzeugtes *Binding*-Objekt auskommt:

```
textBox1.DataBindings.Add("Text", bs, "BirthDate");
```

Bemerkungen

- Mit der *Control*-Eigenschaft können Sie das Steuerelement abrufen, zu dem die *DataBindings*-Collection gehört.

- Nachdem die Steuerelemente angebunden sind, werden lediglich die Werte der ersten Zeile der *DataTable* angezeigt, Möglichkeiten zum Navigieren bzw. Blättern sind noch nicht vorhanden.

6.3 Manuelle Bindung an Listen und Tabellen

Bei dieser komplexeren Form der Datenbindung wollen wir Steuerelemente, die mehrere Werte anzeigen können, an eine Liste von Werten binden. Die dafür am häufigsten verwendeten Steuerelemente sind *DataGridView*, *ComboBox* oder *ListBox*.

6.3.1 DataGridView

Das *DataGridView* ist ein sehr leistungsfähiges Datengitter-Steuerelement.

Beispiel 6.5: Anzeige der *Personal*-Tabelle im *DataGridView*

C#

```
dataGridView1.DataSource = ds;
dataGridView1.DataMember = "Employees";

oder

BindingSource bs = new BindingSource(ds, "Employees");
dataGridView1.DataSource = bs;
```

6.3.2 Datenbindung von ComboBox und ListBox

Häufig werden *ComboBox* und *ListBox* zum Implementieren sogenannter „Nachschlagefunktionalität“ bei *DataTable*s (oder *DataView*s) eingesetzt, zwischen denen eine Master-Detail-Relation besteht. Um die *ComboBox*/*ListBox* mit der Master-Tabelle zu verknüpfen, muss zunächst die *SelectedValue*-Eigenschaft an den in der Mastertabelle enthaltenen Fremdschlüssel angebunden werden. Anschließend werden den *DataSource*-, *DisplayMember*- und *ValueMember*-Eigenschaften die entsprechenden Spalten der Detailtabelle zugewiesen.

Beispiel 6.6: Datenbindung von *ComboBox*

C#

Die Tabellen *Orders* und *Employees* der *Nordwind*-Datenbank sind durch eine Master-Detail-Beziehung verknüpft. In der *ComboBox* soll der zur aktuellen Bestellung gehörige LastName aus der *Employees*-Tabelle angezeigt werden.

Verbinden der *ComboBox* mit der Mastertabelle:

```
BindingSource bindingSourceBest = new BindingSource();
bindingSourceBest.DataSource = ds.Tables["Orders"];
comboBox1.DataBindings.Add(
        "SelectedValue", bindingSourceBest, "EmployeeId");
```

Anbinden der Detaildaten an die *ComboBox*:

```
BindingSource bindingSourcePers = new BindingSource();
bindingSourcePers.DataSource = ds.Tables["Employees"];

comboBox1.DataSource = bindingSourcePers;
comboBox1.DisplayMember = "LastName";
comboBox1.ValueMember = "EmployeeId";
```

6.4 Navigations- und Bearbeitungsfunktionen

Für das Durchblättern der Datensätze sowie für Editieren, Hinzufügen und Löschen haben Sie hauptsächlich zwei Möglichkeiten:

- Sie können die verschiedenen Methoden der *BindingSource* verwenden oder
- Sie verwenden einen *BindingNavigator*, der die Methodenaufrufe kapselt.

6.4.1 Navigieren zwischen den Datensätzen

So wie das gute alte Recordset-Objekt aus den Zeiten vor .NET hat auch die *BindingSource* die Methoden *MoveNext*, *MovePrevious*, *MoveFirst* und *MoveLast.*

Beispiel 6.7: Bewegen zum ersten Datensatz

C#

```
BindingSource bs = new BindingSource(ds, "Employees");
private void Button1_Click(object sender, EventArgs e)
{
    bs.MoveFirst();
}
```

6.4.2 Hinzufügen und Löschen

Dafür bietet die *BindingSource* die Methoden *Add*, *AddNew*, *Remove*, *RemoveAt*, *RemoveCurrent* und *RemoveFilter*.

Beispiel 6.8: Ein neuer Datensatz wird hinzugefügt.

C#

```
bs.AddNew();
```

Der aktuelle Datensatz wird gelöscht:

```
bs.RemoveCurrent();
```

6.4.3 Aktualisieren und Abbrechen

Mit der *EndEdit*- bzw. *CancelEdit*-Methode der *BindingSource* kann der aktuelle Editiervorgang beendet bzw. abgebrochen werden.

Beispiel 6.9: Geänderte Daten vom *DataTable*-Objekt *dt* in die Datenbank übertragen

C#

```
bs.EndEdit();
adap.Update(ds, "Employees");
```

HINWEIS: Wenn Sie die *EndEdit*-Methode nicht aufrufen, werden die geänderten Daten erst beim Weiterblättern in die *DataTable* übernommen.

6.4.4 Verwendung des BindingNavigators

Ein *BindingNavigator* eignet sich nur für die Zusammenarbeit mit einer *BindingSource*.

Beispiel 6.10: Ein *BindingNavigator* wird mit einem *BindingSource*-Objekt *bs* verknüpft.

C#

```
bindingNavigator1.BindingSource = bs;
```

Der *BindingNavigator* bietet alle Funktionen zum Weiterblättern sowie zum Hinzufügen und zum Löschen – mit Ausnahme der „Speichern"- und der „Abbrechen"-Schaltfläche, die Sie selbst hinzufügen und implementieren müssen.

Beispiel 6.11: Ein *BindingNavigator*, dem Sie zwei Schaltflächen hinzugefügt haben, wird für das Speichern eines *DataTable*-Objekts im DataSet *ds* und für das Abbrechen der aktuellen Operation „nachgerüstet".

C#

Speichern:

```
private void ToolStripButton1_Click(object sender, EventArgs e)
{
    bs.EndEdit();
    adap.Update(ds, "Employees");
}
```

Abbrechen:

```
private void ToolStripButton2_Click(object sender, EventArgs e)
{
    bs.CancelEdit();
}
```

Ergebnis

6.5 Die Anzeigedaten formatieren

Zum Formatieren der Inhalte manuell gebundener Steuerelemente ist etwas zusätzlicher Aufwand erforderlich. Die *Binding*-Objekte müssen separat erzeugt und mit Event-Handlern für das *Format*- und für das *Parse*-Event nachgerüstet werden.

Beispiel 6.12: Die Anzeige des Geburtsdatums wird formatiert.

C#

```
Binding b1 = new Binding("Text", bs, "BirthDate");
```

Aufruf der Formatierungsmethoden (Implementierung siehe unten):

```
b1.Format += new ConvertEventHandler(DateToDateString);
b1.Parse +=  new ConvertEventHandler(DateStringToDate);
textBox1.DataBindings.Add(b1);
```

Datenquelle → Anzeige:

```
private void DateToDateString(object sender, ConvertEventArgs e)
{
  try
  {
     e.Value = Convert.ToDateTime(e.Value).ToString("d.M.yyyy");
```

```
  }
  catch{}
}
```

Anzeige → Datenquelle:

```
private void DateStringToDate(object sender, ConvertEventArgs e)
{
   e.Value = Convert.ToDateTime(e.Value);
}
```

7 Erweiterte Grafikausgabe

Wie es die Kapitelüberschrift schon verrät, werden wir Sie im Folgenden mit den **erweiterten** Möglichkeiten der Grafikausgabe unter .NET traktieren. Grundkenntnisse der Grafikprogrammierung, siehe Kapitel 4, sind für die Lektüre dieses Kapitels unerlässlich.

Die wichtigsten Themen im Überblick:

- die GDI+-Transformationen
- Low-Level-Grafikzugriff
- erweiterte Techniken (Alpha-Blending, Animationen …)
- 3D-Grafik unter .NET
- die Verwendung von GDI-Funktionen unter .NET

7.1 Transformieren mit der Matrix-Klasse

Die bereits in Abschnitt 4.3 über die Seitenkoordinaten vorgestellten Transformationen lassen sich auch mithilfe einer Transformationsmatrix realisieren. Die Mathematiker unter den Lesern werden das zu schätzen wissen. Dass es nicht bei den beschriebenen Transformationen bleiben muss und dass vieles auch einfacher realisierbar ist, zeigt der folgende Abschnitt.

7.1.1 Übersicht

Ausgangspunkt aller Überlegungen ist zunächst die Klasse *Matrix*, die vom Namespace *System.Drawing.Drawing2D* bereitgestellt wird. Hierbei handelt es sich intern um eine 3 × 3 Matrix, auch wenn Sie nur Zugriff auf die beiden ersten Spalten haben (die anderen Werte sind konstant).

$$\begin{vmatrix} m11 & m12 & 0 \\ m21 & m22 & 0 \\ dx & dy & 1 \end{vmatrix}$$

Das Füllen der Matrix können Sie bereits mithilfe des Konstruktors bewerkstelligen, übergeben Sie einfach die Werte *m11* bis *dy*.

Beispiel 7.1: Erzeugen einer neuen Matrix

C#

```
Matrix matrix = new Matrix(1, 0,
                           0, 1,
                           0, 0);
```

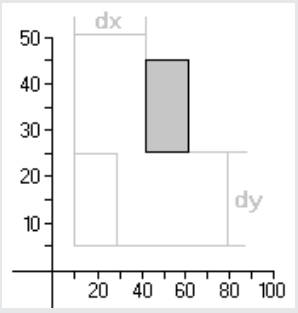
HINWEIS: Zur besseren Übersicht habe ich im Listing noch Zeilenumbrüche hinzugefügt.

Alternativ können Sie über die *Elements*-Auflistung auf die einzelnen Elemente der Matrix zugreifen, es handelt sich jedoch **nicht** um ein zweidimensionales Array.

Die Elemente *dx* und *dy* stehen über die Eigenschaften *OffsetX* und *OffsetY* zur Verfügung.

7.1.2 Translation

Eine Verschiebung bzw. Translation realisieren Sie durch eine Änderung der *dx*-, *dy*-Werte.



$$x' = x + dx$$
$$y' = y + dy$$

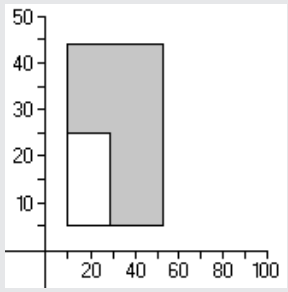
... als Transformationsmatrix:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ dx & dy & 1 \end{vmatrix}$$

Alternativ können Sie die Methode *Translate* auf eine initialisierte Matrix anwenden. Übergeben Sie dazu die Werte für *dx* und *dy*.

7.1.3 Skalierung

Eine Größenänderung bzw. Skalierung erreichen Sie über die Werte *m11* bzw. *m22*:



$$x' = x \cdot m11$$
$$y' = y \cdot m22$$

... als Transformationsmatrix:

$$\begin{vmatrix} m11 & 0 & 0 \\ 0 & m22 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Gleiches erreichen Sie über die Methode *Scale*.

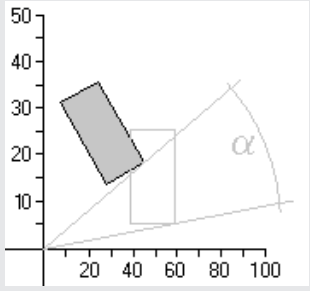
Beispiel 7.2: Verwendung von *Scale*

C#

```
Matrix matrix = new Matrix(1, 0, 0, 1, 0, 0);
matrix.Scale(2, 0);
```

7.1.4 Rotation

Auch an das Drehen des Koordinatensystems haben die .NET-Entwickler gedacht, allerdings genügt nicht das einfache Einsetzen des gewünschten Drehwinkels, sondern Sie müssen schon etwas rechnen:



$$x' = (x \cdot \cos\alpha) - (y \cdot \sin\alpha)$$
$$y' = (y \cdot \sin\alpha) - (y \cdot \cos\alpha)$$

... als Transformationsmatrix:

$$\begin{vmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Sie vermuten es sicher schon, mithilfe der Methode *Rotate* bietet sich ein wesentlich einfacherer Weg zum Drehen des Koordinatensystems.

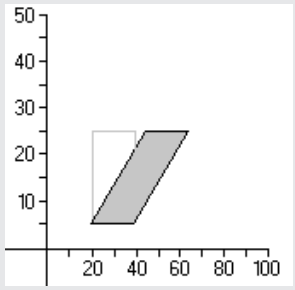
Beispiel 7.3: Verwendung von *Rotate*

C#

```
Matrix matrix = new Matrix(1, 0, 0, 1, 0, 0);
matrix.Rotate(45);      // Gradangabe
```

7.1.5 Scherung

Eine Scherung in x- bzw. y-Richtung erreichen Sie über die folgende Transformationsmatrix:



$$x' = x + (m12 \cdot y)$$
$$y' = y + (m21 \cdot x)$$

... als Transformationsmatrix:

$$\begin{vmatrix} 1 & m12 & 0 \\ m21 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Beispiel 7.4: Auch hier sind Sie mit der Methode *Shear* besser bedient.

C#

```
Matrix matrix = new Matrix(1, 0, 0, 1, 0, 0);
matrix.Shear(0.1f, 0);
```

7.1.6 Zuweisen der Matrix

Die Bearbeitung der Transformationsmatrix ist kein Selbstzweck, über die Eigenschaft *Transform* können Sie die mit obigen Methoden bearbeitete Transformationsmatrix einem *Graphics*-Objekt zuweisen. Nachfolgende Zeichenoperationen werden nun mit dieser Matrix verarbeitet.

Beispiel 7.5: Verschieben (20,10) und Skalieren (2,2) der Ausgabe

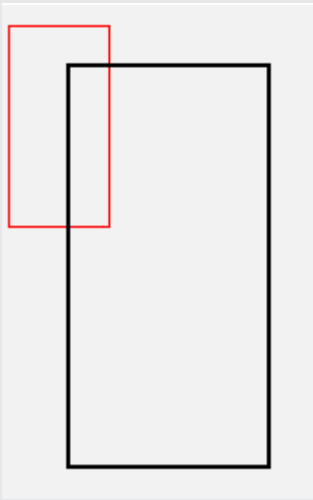
C#

```
Graphics g = CreateGraphics();
Matrix matrix = new Matrix();
matrix.Translate(20, 10);
matrix.Scale(2, 2);
g.DrawRectangle(Pens.Red, 10, 10, 50, 100);
g.Transform = matrix;
g.DrawRectangle(Pens.Black, 10, 10, 50, 100);
```

oder bei direktem Bearbeiten der Matrix:

```
Graphics g = CreateGraphics();
Matrix matrix = new Matrix(2, 0, 0, 2, 20, 10);
g.DrawRectangle(Pens.Red, 10, 10, 50, 100);
g.Transform = matrix;
g.DrawRectangle(Pens.Black, 10, 10, 50, 100);
```

Ergebnis



7.2 Low-Level-Grafikmanipulationen

Möchten Sie Grafiken nicht nur anzeigen, sondern auch verändern, bieten sich auf den ersten Blick die *GetPixel-/SetPixel*-Methoden an. Diese ermöglichen den Zugriff auf die einzelnen Bildpunkte einer **Bitmap**[1], es ist demnach kein Problem, zum Beispiel eine Farbe durch eine andere auszutauschen.

Doch wer bereits erste Schritte auf diesem Gebiet unternommen hat, wird schnell enttäuscht sein. Das Auslesen und Setzen der einzelnen Punkte verbraucht bei komplexeren Aufgaben so viel Zeit, dass dies wohl kaum einem Anwender zumutbar ist, es sei denn, man möchte ihn ärgern.

Beispiel 7.6: Drehen einer Bitmap[2] (in *picturebox1.Image*) um 90°

C#

```
private void Button1_Click(object sender, EventArgs e)
{
   int x;
   int y;
   Bitmap bitmap1;
   Bitmap bitmap2;
   Color color;
```

[1] Dies ist wichtig, Sie müssen eine Pixelgrafik in das *Image* geladen haben.

[2] Ja, ja, es gibt dafür eine extra Methode, an dieser Stelle geht es aber nur um das Grundprinzip!

```
   bitmap1 = (Bitmap)pictureBox1.Image;
   bitmap2 = new Bitmap(bitmap1.Height, b1.Width);
   for(x = 0; x < bitmap1.Width; x++)
      for (y = 0; y < bitmap1.Height; y++)
      {
        color = bitmap1.GetPixel(x, y);
        bitmap2.SetPixel(bitmap1.Height - y - 1, x, color);
      }
    pictureBox1.Image = bitmap2;
    pictureBox1.Refresh();
}
```

Als Alternative bietet sich dem begeisterten GDI-Programmierer die Arbeit mit den *Device Independent Bitmaps* (DIB) an. Doch die Verwendung der entsprechenden Funktionen erfordert nicht nur eine genaue Kenntnis des GDI, sondern auch unnötigen Schreibaufwand, muss doch die DIB wieder in das Bitmap-Format zurückkonvertiert werden. Vom zusätzlichen Speicherverbrauch wollen wir an dieser Stelle gar nicht erst sprechen.

Die Zauberworte für unsere Probleme heißen *LockBits* und *Scan0*. Dabei handelt es sich um Eigenschaften des *Bitmap*-Objekts. Mit *LockBits* wird eine Bitmap im Arbeitsspeicher für uns gesperrt, gleichzeitig können wir die Bitmap in einen für uns günstigen Datentyp umwandeln. Der eigentliche Clou ist *Scan0*, ein Pointer auf das erste Bitmap-Byte.

HINWEIS: Auch wenn es auf den ersten Blick so scheinen mag, *Scan0* ist kein direkter Ersatz für *GetPixel* und *SetPixel*. Wie die Bitmap-Daten aufgebaut sind, wie viele Spalten und Zeilen es gibt, bleibt unberücksichtigt. Ein Fehler bei der Arbeit mit diesem Pointer führt meist zu einem Programmabsturz.

7.2.1 Worauf zeigt Scan0?

Die allgemein Antwort lautet: auf das erste Byte der Bitmap. Wie die folgenden Daten aufgebaut sind und aus wie vielen Bytes sie bestehen, wird durch das Bitmap-Format bestimmt. Dieses können Sie mit *LockBits* direkt angeben:

Syntax:

```
BitmapData LockBits(Rectangle rect, ImageLockMode flags, PixelFormat
format);
```

Übergeben werden der gewünschte Ausschnitt der Bitmap (im Zweifel alles), der Lockmode (meist *ReadWrite*) und das gewünschte Bitmap-Format.

C# unterstützt unter anderem folgende wichtige Bitmap-Formate:

Format	Beschreibung
Format1bppIndexed	Schwarz-Weiß-Bilder, bei denen jedes Pixel durch ein Bit dargestellt wird
Format4bppIndexed	Bilder mit 16 Farben, ein Byte stellt somit die Informationen für zwei benachbarte Pixel zur Verfügung.

Format8bppIndexed	Bilder dieses Typs können 256 Farben darstellen. Damit entspricht ein Byte auch einem Pixel. Doch freuen Sie sich nicht zu früh, es handelt sich nicht um den direkten Farbwert, sondern nur um den Index in einer getrennt gespeicherten Farbpalette.
Format-16bppRgb555, Format16bppRgb565	Bei diesem Format werden mit jeweils 5 bzw. 6 Bit für die drei Grundfarben die Farbwerte dargestellt bzw. abgespeichert. Das interne Format: ▪ pf15bit:0rrrrrgggggbbbbb ▪ pf16bit: rrrrrgggggbbbbb Dass die Arbeit mit derart verschachtelten Daten nicht unbedingt einfach ist, dürfte schnell ersichtlich sein.
Format24bppRgb	Das Wunschformat jedes Grafikprogrammierers: Jeder Punkt wird mit drei Bytes (RGB) zu je 8 Bit dargestellt. Der Zugriff auf derartige Bitmaps ist relativ problemlos realisierbar, beachten Sie jedoch, dass die Bildzeilen immer auf Vielfache von vier aufgerundet werden.
Format32bppArgb	Noch etwas schneller lassen sich 32-Bit-Bilder bearbeiten. Dies wird durch die bessere Speicherausrichtung (4 Byte) erreicht. Das vierte Byte hat normalerweise keine Bedeutung, kann aber für Transparenzinformationen genutzt werden. Der zusätzlich benötigte Speicher spielt heute wohl kaum noch eine Rolle.

7.2.2 Anzahl der Spalten bestimmen

Eigentlich müsste die Frage anders gestellt werden, da meist nicht die Anzahl der Bildpunkte, sondern die Anzahl der nötigen Bytes von Bedeutung ist. Die Pixelanzahl können Sie mit *Bitmap.Width* bestimmen, die Byte-Anzahl berechnet sich aus den jeweiligen Bildformaten, wie sie im vorhergehenden Abschnitt vorgestellt wurden.

32-Bit-Bild, Bitmap.Width = 300

*4 * Bitmap.Width = 1200 Bytes/Zeile*

Doch Vorsicht: Ein 24-Bit-Bild mit einer Breite von 299 Pixeln hat nicht etwa

*3 * Bitmap.Width = 897 Bytes/Zeile*

sondern

*((3 * Bitmap.Width + 3) DIV 4) * 4 = 900 Bytes/Zeile*

HINWEIS: Die Bitmaps werden in jeder Zeile auf das Vielfache von 4 Bytes aufgefüllt!

Aus diesem Grund finden Sie auch eine zusätzliche Eigenschaft *Stride*, die uns die tatsächliche Länge einer Bitmap-Zeile in Pixeln liefert.

Beispiel 7.7: Bestimmen der Anzahl von Füll-Bytes (24-Bit-Bitmap)

C#

```
using System.Drawing.Imaging;
...


    Bitmap bmp = (Bitmap)pictureBox1.Image;
    BitmapData bmpData = bmp.LockBits(new Rectangle(0, 0,
        bmp.Width, bmp.Height),
        ImageLockMode.ReadWrite, PixelFormat.Format24bppRgb);
    int lineoffs = bmpData.Stride - bmp.Width * 3;
    bmp.UnlockBits(bmpData);
```

HINWEIS: Dieser Offset muss nach der Abarbeitung einer Zeile zum Pointer hinzuaddiert werden, um wieder auf das erste Byte der folgenden Zeile zu zeigen.

7.2.3 Anzahl der Zeilen bestimmen

Diese Antwort ist schnell gegeben. Über die Eigenschaft *Bitmap.Height* steht Ihnen direkt der Wert zur Verfügung.

7.2.4 Zugriff im Detail (erster Versuch)

Folgende Reihenfolge müssen Sie beim direkten Zugriff auf die einzelnen Bitmap-Bytes beachten:

- Bitmap mit *LockBits* sperren,
- mit *Scan0* einen Pointer auf das erste Byte ermitteln,
- über *Marshal.Read...* die gewünschten Bytes/Integers etc. lesen,
- über *Marshal.Write...* die gewünschten Bytes/Integers schreiben,
- die Bitmap mit *UnlockBits* freigeben und
- eventuell die zugehörige *PictureBox* mit *Refresh* aktualisieren.

Den wohl wichtigsten Punkt dürfen wir natürlich auch nicht vergessen:

HINWEIS: Auch wenn die Konstanten *Format24bppRgb* oder *Format32bppArgb* heißen, lassen Sie sich nicht ins Boxhorn jagen! Die Bytes liegen immer in der Reihenfolge Blau-Grün-Rot bzw. Blau-Grün-Rot-Alpha im Speicher!

Beispiel 7.8: Alle Pixel der Grafik sollen auf Schwarz gesetzt werden (24-Bit-Bitmap).

C#

```
using System.Drawing.Imaging;
using System.Runtime.InteropServices;
...

    private void Button3_Click(object sender, EventArgs e)
    {
      Bitmap bmp = pictureBox1.Image as Bitmap;
```

Sperren der Bitmap:

```
      BitmapData bmpData = bmp.LockBits(new Rectangle(0, 0,
                           bmp.Width, bmp.Height),
                    ImageLockMode.ReadWrite,
                           PixelFormat.Format24bppRgb);
```

Pointer ermitteln:

```
      IntPtr ptr = bmpData.Scan0;
      Int offset = 0;
      for (int y=0; y < bmp.Height -1; y++ )
      {
        for (int x=0; x < bmp.Width *3 -1; x++ )
          {
             byte p = 0;
```

Schreiben in den gewünschten Speicherbereich (alle Farbwerte = 0):

```
             Marshal.WriteByte(ptr, offset, p);
```

Offset für *Marshal.Write* setzen:

```
             offset += 1;
          }
      }
```

Freigabe der Bitmap und *PictureBox* aktualisieren:

```
      bmp.UnlockBits(bmpData);
      pictureBox1.Refresh();
    }
```

7.2.5 Zugriff im Detail (zweiter Versuch)

Nach unserem ersten Versuch wollen wir es jetzt performanter machen. Dazu bietet sich in C# die Verwendung von „unsicherem“ Code an. Hier können wir mit Pointern schalten und walten, wie wir wollen, und natürlich auch reichlich Fehler produzieren.

HINWEIS: Gekennzeichnet werden unsichere Codeabschnitte mit dem *unsafe*-Bezeichner. Prinzipiell wird jedoch von der Verwendung von unsafe-Code abgeraten.

Beispiel 7.9: Wir versuchen uns mit dem gleichen Beispiel wie im vorherigen Abschnitt.

C#

```
            Bitmap bmp = pictureBox1.Image as Bitmap;


        BitmapData bmpData = bmp.LockBits(new Rectangle(0, 0,
                                bmp.Width, bmp.Height),
                        ImageLockMode.ReadWrite, PixelFormat.Format24bppRgb);
        unsafe
        {
```

Hier ermitteln wir einen Byte-Pointer auf das erste Byte der Bitmap:

```
            byte* ptr = (byte*)bmpData.Scan0;
            int lineoffs = bmpData.Stride - bmp.Width * 3;
```

Für alle Zeilen und Spalten:

```
            for (int y = 0; y < bmp.Height - 1; y++)
            {
                for (int x = 0; x < bmp.Width * 3 - 1; x++)
                {
```

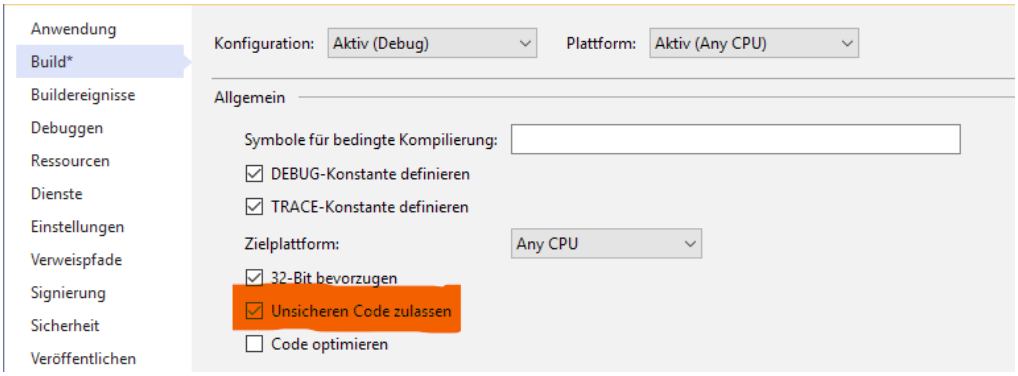
Wert setzen und Pointer inkrementieren:

```
                    ptr[0] = 0;
                    ptr++;
                }
```

Am Zeilenende den Offset addieren:

```
                ptr += lineoffs;
            }
        }
        bmp.UnlockBits(bmpData);
        pictureBox1.Refresh();
```

HINWEIS: Damit Sie das Beispiel kompilieren können, muss in den Projektoptionen auch das Kompilieren von unsicherem Code erlaubt werden.



Weiter optimieren

Ein Blick auf unseren Code zeigt, dass wir eigentlich viel zu viele Berechnungen in den Schleifen ausführen (x, y berechnen, Offset addieren). Diesem Missstand wollen wir nun abhelfen. Um die unselige Offset-Rechnerei zu vermeiden, erzeugen wir einfach eine 32-Bit-Grafik. Damit entfällt auch die Notwendigkeit, zwischen Zeilen und Spalten zu unterscheiden, wir können die geschachtelten Schleifen auflösen und der Pointer kann über den gesamten Speicherbereich iterieren:

```
Bitmap bmp = pictureBox1.Image as Bitmap;
   BitmapData bmpData = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height),
                        ImageLockMode.ReadWrite, PixelFormat.Format32bppRgb);
unsafe
{
   byte* ptr = (byte*)bmpData.Scan0;
   int size = bmp.Height * bmp.Width * 4;  // 4 Bytes pro Pixel
   for (int i = 0; i < size; i++)
       ptr[i] = 0;
}
bmp.UnlockBits(bmpData);
pictureBox1.Refresh();
```

Damit wird uns dieser Lösungsansatz auch für die weiteren Beispiele als Vorlage dienen.

7.2.6 Invertieren

Eine der einfachsten Operationen ist das Invertieren einer Bitmap, die einzelnen RGB-Werte brauchen nur negiert zu werden, das heißt, der Farbwert ist von 255 abzuziehen.

Beispiel 7.10: Eine Bitmap wird invertiert.

C#

```
public void Invert(Bitmap bmp)
{
                BitmapData bmpData = bmp.LockBits(new Rectangle(0, 0,
                bmp.Width, bmp.Height),
                ImageLockMode.ReadWrite, PixelFormat.Format32bppRgb);
    unsafe
    {
        byte* ptr = (byte*)bmpData.Scan0;
        int size = bmp.Height * bmp.Width * 4;
        for (int i = 0; i < size; i++)
        {
```

HINWEIS: Wir verwenden *xor 255,* das ist noch mal etwas schneller als die Subtraktion.

```
            ptr[i] = (byte)(ptr[i]^255);
        //    oder auch
        //    ptr[i] = (byte)(255 - ptr[i]);
        }
```

```
            }
            bmp.UnlockBits(bmpData);
        }
```

Ergebnis

7.2.7 In Graustufen umwandeln

Beim Umwandeln einer Farbgrafik in ein Graustufenbild werden die einzelnen Farben entsprechend ihrer Leuchtkraft bewertet und daraus ein Graustufenwert (8 Bit) berechnet. Dieser Wert wird nachfolgend allen drei Farbkanälen zugewiesen.

Beispiel 7.11: Eine farbige Bitmap wird in ein Graustufenbild transformiert.

C#

```
        public void Grey(Bitmap bmp)
        {
            BitmapData bmpData = bmp.LockBits(new Rectangle(0, 0, bmp.Width,
bmp.Height),
                                    ImageLockMode.ReadWrite,
PixelFormat.Format32bppRgb);
            unsafe
            {
                byte* ptr = (byte*)bmpData.Scan0;
                int size = bmp.Height * bmp.Width;
                for (int i = 0; i < size; i++)
                {
```

Farbwerte auslesen:

```
                    byte blau = ptr[0];
                    byte grün = ptr[1];
                    byte rot = ptr[2];
```

Grauwert berechnen:

```
                    byte grau = (byte)((77 * blau + 151 * grün + 28 * rot) / 256);
```

Grauwert in die RGB-Bytes eintragen:

```
                    ptr[0] = ptr[1] = ptr[2] = grau;
```

Und Sprung auf das nächste Pixel:

```
                    ptr += 4;
                }
            }
            bmp.UnlockBits(bmpData);
        }
```

Alternativ können Sie auch diese Anweisung verwenden:

```
ptr[0] = ptr[1] = ptr[2] = (byte)((77 * ptr[0] + 151 * ptr[1] + 28 * ptr[2]) / 256);
```

7.2.8 Heller/Dunkler

Um ein Bild aufzuhellen oder dunkler zu machen, genügt es, dass zu jedem Wert eine Konstante addiert wird. Um Werteüberläufe zu verhindern, müssten wir entweder bei jedem Wert abfragen, ob das Berechnungsergebnis den Wertebereich (255) überschreitet, oder wir legen gleich ein Array an, in dem für jeden der möglichen 256 Werte der neue Wert gespeichert ist. Insbesondere bei großen Bildern können Sie so wertvolle Sekunden sparen, da nur noch der Wert aus dem Array ausgelesen werden muss (LUT = Look-Up-Table).

Beispiel 7.12: Die Helligkeit einer Bitmap wird geändert.

C#

Normieren auf den Bereich 0 ... 255:

```
private byte Normiere(int value)
      {
           if (value < 0)
           {
               return 0;
           }
           if (value > 255)
           {
               return 255;
           }
           return (byte)value;
       }
```

Die Konvertierungsfunktion (*value* = Änderung der Helligkeit):

```
        public void Brightness(Bitmap bmp, short value)
        {
```

Zunächst die LUT berechnen:

```
            byte[] ar = new byte[256];
            for (int i = 0; i < 256; i++)
            {
                ar[i] = normiere(i + value);
            }
            BitmapData bmpData = bmp.LockBits(new Rectangle(0, 0,
                       bmp.Width, bmp.Height),
                                  ImageLockMode.ReadWrite,
PixelFormat.Format32bppRgb);
            unsafe
            {
                byte* ptr = (byte*)bmpData.Scan0;
                int size = bmp.Height * bmp.Width;
                for (int i = 0; i < size; i++)
                {
```

Hier lesen wir einfach die Werte aus der LUT aus:

```
                    ptr[0] = ar[ptr[0]];
                    ptr[1] = ar[ptr[1]];
                    ptr[2] = ar[ptr[2]];
                    ptr += 4;
                }
            }
            bmp.UnlockBits(bmpData);
        }
```

Ergebnis

Übergeben Sie der Funktion einen positiven oder negativen Wert, um das Bild aufzuhellen oder abzudunkeln.

7.2.9 Kontrast

Um den Kontrast eines Bilds zu erhöhen, normieren wir zunächst die Farbwerte, indem wir diese in einen Integerwert umwandeln und 128 abziehen. Den resultierenden Wert multiplizieren wir mit einem konstanten Faktor, nachfolgend wird die Normierung durch Addition von 128 wieder aufgehoben. Da wir die Gleitkomma-Operationen nicht für jeden Pixel ausführen möchten (Performance!), verwenden wir wieder ein Array (LUT), in welchem wir die Farbwerte vorberechnen.

Beispiel 7.13: Kontrast einer Bitmap verändern

C#

```
        public void Contrast(Bitmap bmp, float value)
        {
            byte[] ar = new byte[256];
            value = 1 + value / 100;
```

Zunächst die LUT berechnen:

```
            for (int i = 0; i < 256; i++)
            {
                ar[i] = Normiere((int)((i - 128) * value) + 128);
            }

            BitmapData bmpData = bmp.LockBits(new Rectangle(0, 0,
                                 bmp.Width, bmp.Height),
                                 ImageLockMode.ReadWrite,
PixelFormat.Format32bppRgb);
            unsafe
            {
```

```
                byte* ptr = (byte*) bmpData.Scan0;
                int size = bmp.Height * bmp.Width;
```

Werte aus der LUT auslesen und zuweisen:

```
                for (int i = 0; i < size; i++)
                {
                    ptr[0] = ar[ptr[0]]; ptr[1] = ar[ptr[1]]; ptr[2] = ar[ptr[2]];
                    ptr += 4;
                }
            }
            bmp.UnlockBits(bmpData);
        }
```

Ergebnis

7.3 Fortgeschrittene Techniken

Im Folgenden möchten wir noch auf einige Techniken eingehen, mit denen Sie die Ausgabeergebnisse bezüglich Geschwindigkeit und/oder Qualität verbessern können.

7.3.1 Flackerfrei dank Double Buffering

Sicher haben Sie auch schon vor dem Problem gestanden, dass Sie umfangreiche Grafiken ausgeben wollten, die Darstellung auf dem Bildschirm aber während dieser Zeit unerträglich flackert. Ein Beispiel zeigt die Problematik und im Anschluss auch den Lösungsansatz.

Beispiel 7.14: Ausgabe einiger Linien

C#

```
double von = Environment.TickCount;
Graphics g = CreateGraphics();
for (int i = 0; i < 800; i++)
{
    g.DrawLine(Pens.Red, 0, i, 400, i);
    g.DrawLine(Pens.Blue, 0, 0, 400, i);
    g.DrawLine(Pens.Green, 400, 0, 0, i);
}
g.Dispose();
double bis = Environment.TickCount;
label1.Text = ((bis - von) / 1000).ToString() + " s";
```

Nach unerträglichen 2 Sekunden und viel Flackerei ist die Grafik endlich aufgebaut. Zwischenzeitlich sind auch Zeichenoperationen zu sehen, die wieder überdeckt werden.

Eine Puffer-Bitmap erzeugen

Mithilfe einer zusätzlichen Speicher-Bitmap können wir die Ausgaben zunächst unabhängig von der Oberfläche realisieren und nachfolgend ausgeben.

Beispiel 7.15: Fortführung des obigen Beispiels

C#

```
double von = Environment.TickCount;
```

Bitmap erzeugen (entsprechend der Fenstergröße):

```
Bitmap bmp = new Bitmap(ClientRectangle.Width,
                        ClientRectangle.Height);
```

Grafikausgaben in der Bitmap vornehmen:

```
Graphics g = Graphics.FromImage(bmp);
for (int i = 0; i < 800; i++)
{
```

```
                g.DrawLine(Pens.Red, 0, i, 400, i);
                g.DrawLine(Pens.Blue, 0, 0, 400, i);
                g.DrawLine(Pens.Green, 400, 0, 0, i);
            }
            g.Dispose();
```

Die Bitmap im Fenster wiedergeben:

```
            g = CreateGraphics();
            g.DrawImage(bmp, 0, 0);
            g.Dispose();
            bmp.Dispose();
            double bis = Environment.TickCount;
            label1.Text = ((bis - von) / 1000).ToString() + " s";
```

Mit 0,031 Sekunden Zeitbedarf können wir schon eine wesentliche Verbesserung feststellen, zusätzlich ist die Ausgabe flackerfrei und – last but not least – können wir die Bitmap mit jeder Paint-Operation ausgeben, ohne sie erneut aufbauen zu müssen.

Und was ist mit der PictureBox?

Hier haben wir es mit einem Sonderfall zu tun, die *PictureBox* beherrscht bereits „ab Werk" Double Buffering. Doch Achtung:

HINWEIS: Sie müssen mit der **enthaltenen** Grafik arbeiten, nicht mit der Bildschirmdarstellung.

Falsch:

```
Graphics g = pictureBox1.CreateGraphics();
...
```

Richtig:

```
Graphics g = Graphics.FromImage(pictureBox1.Image);
...
g.Dispose();
pictureBox1.Invalidate();
```

HINWEIS: Wichtig ist das *Invalidate*, sonst passiert auf dem Bildschirm nichts, bis die Grafik – zum Beispiel nach einem Verdecken – neu gezeichnet werden muss.

Sollten Sie keine Grafik in die *PictureBox* geladen haben, erzeugen Sie einfach eine entsprechende Grafik:

```
Bitmap bmp = new Bitmap(ClientRectangle.Width, ClientRectangle.Height);
pictureBox1.Image = bmp;
```

7.3.2 Transparenz realisieren

Sollen Bilder ein- bzw. ausgeblendet werden, müssen Sie das Rad nicht neu erfinden. Hier hilft Ihnen GDI+ mit seiner bereits eingebauten Fähigkeit, die Transparenz mithilfe des Alpha-Kanals zu steuern.

Beispiel 7.16: Ein Bild mit einer Transparenz zwischen 0 und 100% darstellen

C#

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

Das Flackern unterbinden:

```
            SetStyle(ControlStyles.UserPaint, true);
            SetStyle(ControlStyles.AllPaintingInWmPaint, true);
            SetStyle(ControlStyles.DoubleBuffer, true);
        }
```

Bild zeichnen:

```
        protected override void OnPaint(PaintEventArgs e)
        {
            base.OnPaint(e);
```

Der Bitmap werden neue Attribute (eine *ColorMatrix*) zugewiesen:

```
            ImageAttributes attr = new ImageAttributes();
            ColorMatrix matrix = new ColorMatrix();
```

Hier wird die Transparenz festgelegt:

```
            matrix.Matrix33 = trackBar1.Value / 100f;
            attr.SetColorMatrix(matrix);
            Bitmap bmp = pictureBox1.Image as Bitmap;
```

Leider ist die erforderliche Überladung der *DrawImage*-Methode etwas umfangreich:

```
            e.Graphics.DrawImage(bmp, new Rectangle(0, 0, bmp.Width, bmp.Height), 0, 0,
                                 bmp.Width, bmp.Height, GraphicsUnit.Pixel, attr);
        }
```

Zu guter Letzt müssen wir noch die Bildaktualisierung erzwingen:

```
        private void trackBar1_Scroll(object sender, EventArgs e)
        {
           Invalidate();
        }
```

7.4 Praxisbeispiele

7.4.1 Die Transformationsmatrix verstehen

Wem die Ausführungen in Abschnitt 7.1 zu theoretisch waren, hier ist ein praktisches Beispiel zum Experimentieren.

Über sechs *TextBox*en erhalten Sie die Möglichkeit, die Matrix-Elemente *m11* bis *dy* zur Laufzeit frei zu verändern. Einige weitere, bereits vorkonfigurierte Beispiele können Sie über eine *ComboBox* abrufen.

Oberfläche

Siehe Laufzeitansicht am Schluss des Beispiels. In die *ComboBox* tragen Sie folgende Werte ein:

```
-
Reset
Rotate(15)
Rotate(45)
Shear(0.1,0)
Shear(0,0.1)
Translate(5,0)
Translate(0,5)
Scale(2,0)
Scale(0,2)
```

Quelltext

Binden Sie zunächst den Namespace *System.Drawing.Drawing2D* ein.

Das Button-Klick-Ereignis:

```
private void Button1_Click(object sender, EventArgs e)
   Matrix matrix = new Matrix(Convert.ToSingle(textBox1.Text),
                              Convert.ToSingle(textBox2.Text),
                              Convert.ToSingle(textBox3.Text),
                             Convert.ToSingle(textBox4.Text),
                              Convert.ToSingle(textBox5.Text),
                              Convert.ToSingle(textBox6.Text));
```

Wie Sie sehen, füllen wir die Matrix gleich beim Erstellen. Auf eine Fehlerprüfung haben wir verzichtet.

```
   Graphics g = CreateGraphics();
```

Ausgabe löschen und zunächst ein Rechteck **ohne** Transformation zeichnen:

```
   g.Clear(BackColor);
   g.DrawRectangle(Pens.Red, 10, 10, 50, 100);
```

Transformationsmatrix anwenden:

```
   g.Transform = matrix;
```

Die eigentlichen Zeichenfunktionen realisieren:

```
   g.DrawLine(Pens.Red, 0, 0, 200, 0);
   g.DrawLine(Pens.Red, 0, 0, 0, 200);
   g.DrawRectangle(Pens.Black, 10, 10, 50, 100);
```

Die über die *ComboBox* abrufbaren Beispiele:

```
private void ComboBox1_SelectedIndexChanged(object sender, EventArgs e)
{
    Matrix matrix;
```

Erster Eintrag (ohne Funktion):

```
    if (comboBox1.SelectedIndex == 0)
   {
        return;
    }
```

Zweiter Eintrag (Zurücksetzen der Matrix):

```
    if (comboBox1.SelectedIndex == 1)
    {
        matrix = new Matrix(1, 0, 0, 1, 0, 0);
    }
```

Auslesen der *TextBox*en:

```
else
{
    matrix = new Matrix(Convert.ToSingle(textBox1.Text),
                        Convert.ToSingle(textBox2.Text),
                        Convert.ToSingle(textBox3.Text),
                        Convert.ToSingle(textBox4.Text),
                        Convert.ToSingle(textBox5.Text),
                        Convert.ToSingle(textBox6.Text));
}
Graphics g = CreateGraphics();
g.Clear(BackColor);
g.DrawRectangle(Pens.Red, 10, 10, 50, 100);
```

Direktes Verändern der Matrix durch Methodenaufrufe:

```
switch (comboBox1.SelectedIndex)
{
  case 2:   // Rot 15
      matrix.Rotate(15);
      break;
  case 3:   // Rot 45
      matrix.Rotate(45);
      break;
  case 4:
      matrix.Shear(0.1f, 0);
      break;
  case 5:
      matrix.Shear(0, 0.1f);
      break;
  case 6:
      matrix.Translate(5, 0);
      break;
  case 7:
      matrix.Translate(0, 5);
      break;
  case 8:
      matrix.Scale(2, 0);
      break;
  case 9:
      matrix.Scale(0, 2);
      break;
  default:
        break;
}
```

Die neuen Werte in die *TextBox*en eintragen:

```
textBox1.Text = matrix.Elements[0].ToString();
textBox2.Text = matrix.Elements[1].ToString();
textBox3.Text = matrix.Elements[2].ToString();
textBox4.Text = matrix.Elements[3].ToString();
textBox5.Text = matrix.Elements[4].ToString();
textBox6.Text = matrix.Elements[5].ToString();
```

Transformationsmatrix zuordnen:

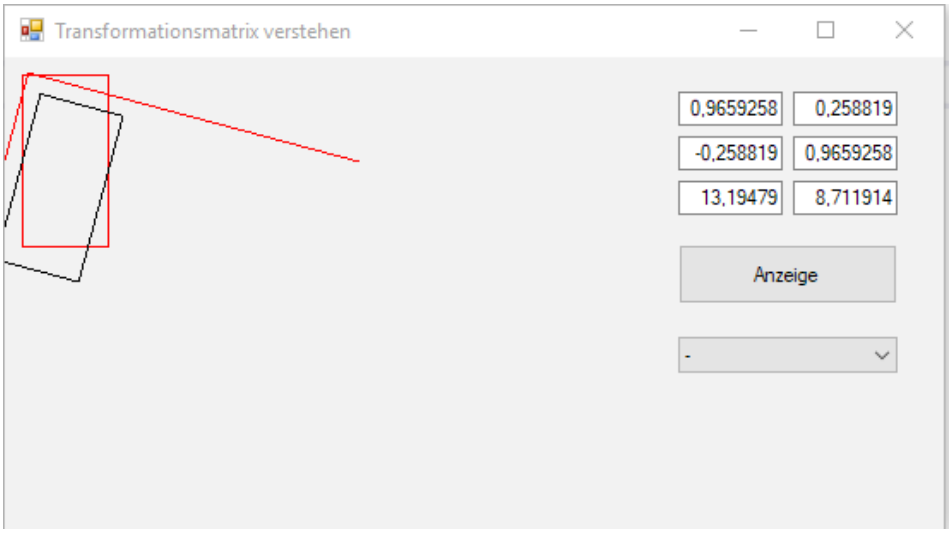
```
    g.Transform = matrix;
```

Grafikausgabe:

```
    g.DrawLine(Pens.Red, 0, 0, 200, 0);
    g.DrawLine(Pens.Red, 0, 0, 0, 200);
    g.DrawRectangle(Pens.Black, 10, 10, 50, 100);
    comboBox1.SelectedIndex = 0;
}
```

Test

Starten Sie die Anwendung und versuchen Sie sich zunächst an Skalierungen und Translationen (siehe Abschnitt 7.1). Wer zunächst einen Überblick bekommen möchte, setzt die Matrix über die *ComboBox* zurück und wählt dann eine Transformation per *ComboBox*. Die Änderung der Werte in den *TextBox*en dürften schnell für mehr Klarheit sorgen:

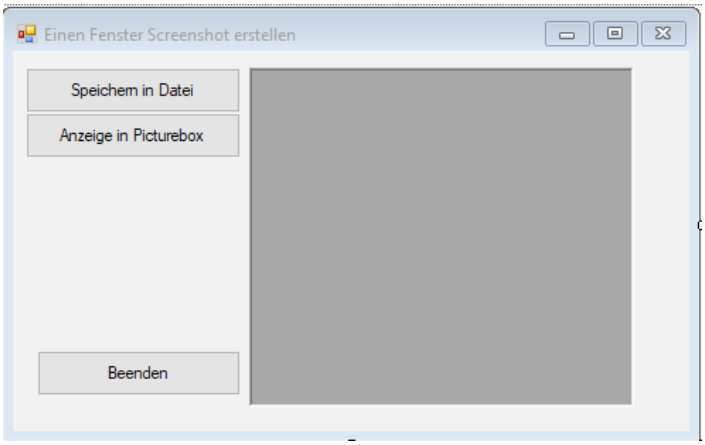


7.4.2 Einen Fenster-Screenshot erzeugen

Geht es darum, einen Screenshot vom aktuellen Fenster zu erzeugen, kommen Sie um ein wenig GDI-Programmierung nicht herum.

Oberfläche

Erstellen Sie zunächst eine Oberfläche entsprechend folgender Abbildung (*PictureBox*, drei *Button*s):



Quelltext

```
using System.Runtime.InteropServices;

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

Binden Sie nachfolgende Konstante sowie die GDI-Funktion *BitBlt* ein:

```
    private const int SRCCOPY = 0xCC0020;

    [DllImport("gdi32.dll")]
    private  static extern int  BitBlt(IntPtr  hDestDC,int  x,int  y,int  nWidth,
                                       int  nHeight, IntPtr  hSrcDC,int  xSrc,
                                       int  ySrc,int  dwRop);
    ...
```

Die Routine zum Speichern des Screenshots:

```
    private void Button1_Click(object sender, EventArgs e)
    {
```

Erzeugen einer neuen Bitmap mit den Maßen und der Farbtiefe des aktuellen Fensters:

```
        Graphics g1 = CreateGraphics();
        Image img = new Bitmap(ClientRectangle.Width,
                               ClientRectangle.Height, g1);
        Graphics g2 = Graphics.FromImage(img);
```

Kopieren der Fenster-Bitmap in die eigene Bitmap:

```
        IntPtr dc1 = g1.GetHdc();
        IntPtr dc2 = g2.GetHdc();
        BitBlt(dc2, 0, 0, ClientRectangle.Width,
               ClientRectangle.Height, dc1, 0, 0, 13369376);
        g1.ReleaseHdc(dc1);
        g2.ReleaseHdc(dc2);
```

Speichern der Daten im PNG-Format:

```
        img.Save("Form1.png", System.Drawing.Imaging.ImageFormat.Png);
        MessageBox.Show("Fenster-Screenshot gesichert", "Info");
    }
```

Ähnlich gestaltet sich die Routine zur Anzeige in der *PictureBox*:

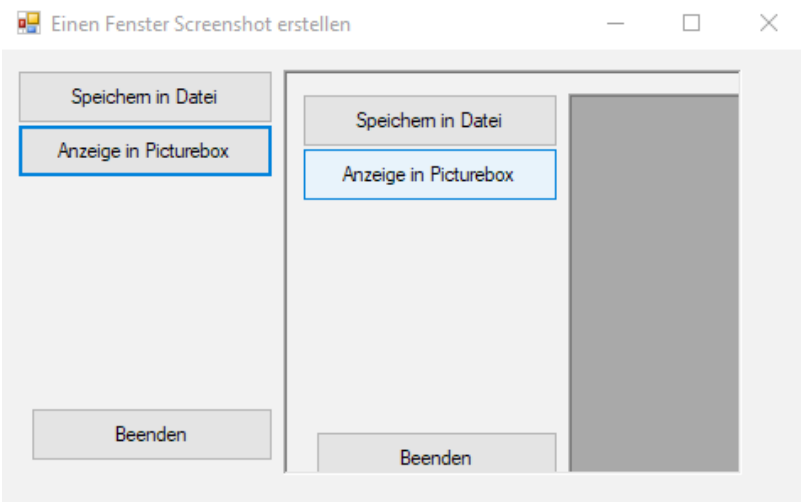
```
    private void Button2_Click(object sender, EventArgs e)
    {
       Graphics g1 = CreateGraphics();
       Image img = new Bitmap(ClientRectangle.Width,
                              ClientRectangle.Height, g1);
       Graphics g2 = Graphics.FromImage(img);

       IntPtr dc1 = g1.GetHdc();
       IntPtr dc2 = g2.GetHdc();
       BitBlt(dc2, 0, 0, ClientRectangle.Width, ClientRectangle.Height,
              dc1, 0, 0, 13369376);
       g1.ReleaseHdc(dc1);
       g2.ReleaseHdc(dc2);
```

Wir weisen das *Image* der *PictureBox* zu:

```
        pictureBox1.Image = img;
    }
    ...
}
```

Test

Nach dem Programmstart und dem Klick auf den Button „Anzeige in Picturebox“ sollte sich Ihnen der folgende Anblick bieten:



Die erzeugte Datei *Form1.png* findet sich im Anwendungsverzeichnis.

8 Ressourcen/ Lokalisierung

Über Ressourcen können Sie externe Informationen in Ihr Programm aufnehmen. Das betrifft Texte, Grafiken und andere Elemente, die sich nicht ohne Weiteres per Code darstellen lassen. Das .NET-Framework kennt prinzipiell zwei Typen von Ressourcen:

- Manifestressourcen und
- typisierte Ressourcen.

Mit beiden Ressourcentypen und ihren Ableitungen (streng typisierte Ressourcen) werden wir uns in diesem Kapitel auseinandersetzen.

Eine besondere Bedeutung haben typisierte Ressourcen vor allem im Zusammenhang mit der Lokalisierung von Anwendungen (siehe Abschnitt 8.4).

8.1 Manifestressourcen

Manifestressourcen können nahezu beliebige Dateien sein, die zur Entwurfszeit (als Ergebnis des Build-Prozesses) in die Assembly integriert werden. Eine Assembly kann mehrere Manifestressourcen enthalten. „Manifestressource“ heißt es deshalb, weil die Namen der Ressourcen im Manifest der Assembly abgelegt sind. Zur Laufzeit kann jede Manifestressource über ihren Namen als Stream ausgelesen werden.

8.1.1 Erstellen von Manifestressourcen

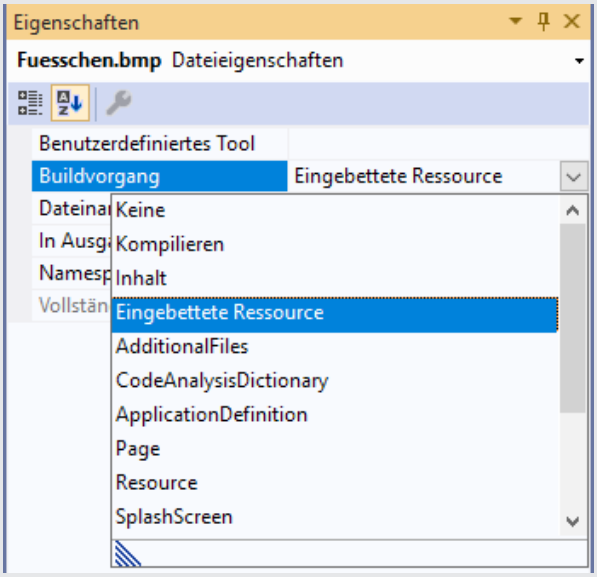
Normalerweise verwendet man zum Anlegen einer Manifestressource den Assembly Linker (*al.exe*). Besitzer von Visual Studio können aber auf dieses Tool locker verzichten, da sie die entsprechenden Dateien lediglich zum Projekt hinzufügen und die *Buildvorgang*-Eigenschaft auf *Eingebettete Ressource* setzen müssen.

Beispiel 8.1: Manifestressource erstellen

C#

Um die Bilddatei *Fuesschen.bmp* als Manifestressource anzulegen, wählen Sie im Kontextmenü des Projektmappen-Explorers **Hinzufügen | Vorhandenes Element...** (oder Hauptmenü **Projekt | Vorhandenes Element hinzufügen...**) und selektieren die Datei *Bild1.jpg*. Im Kontextmenü von *Bild1.jpg* klicken Sie auf **Eigenschaften**:

Im Eigenschaftendialog stellen Sie die Eigenschaft *Buildvorgang* auf *Eingebettete Ressource*:



Nach dem Kompilieren der Anwendung befindet sich die Bilddatei in der erzeugten Assembly, Sie brauchen also keine weitere Datei mitzugeben.

8.1.2 Zugriff auf Manifestressourcen

Der übliche Weg zu eingebetteten Ressourcen führt über die Methoden der *Assembly*-Klasse. So kann man die *GetManifestResourceNames*- bzw. *GetManifestResourceStream*-Methode verwenden, um alle eingebetteten Ressourcen einer bestimmten Assembly aufzulisten bzw. zu laden. Bevor wir aber zu Einzelheiten kommen, wollen wir erläutern, nach welchem Muster die Namensvergabe erfolgt.

Namensgebung eingebetteter Ressourcen

Allgemein entspricht der Name einer eingebetteten Ressource folgendem Muster:

Syntax:

```
<DefaultNamespace>.<Unterverzeichnisse>.Dateiname
```

Der *DefaultNamespace* wird in den Projekteigenschaften festgelegt und ist meist identisch mit dem Namen der Assembly (z. B. *WindowsApplication1*). Falls sich – wie in unserem Fall – die Ressource im Rootverzeichnis des Projekts befindet, entfallen die *Unterverzeichnisse* und der Name ist *<DefaultNamespace>.Dateiname*. Mehrere Unterverzeichnisse sind nicht durch Schrägstrich, sondern durch Punkte voneinander zu trennen. Es ist möglich, dass Sie spezielle Verzeichnisse innerhalb Ihres Projekts erzeugen, um dort die Ressource(n) abzulegen, z. B. ein Verzeichnis *\Bilder*. Dann wäre der Name der Ressource z. B. *WindowsApplication1.Bilder.Bild1.jpg*.

HINWEIS: Ist der Code einmal generiert, so können Sie zwar den Namespace nachträglich ändern, nicht aber die Unterverzeichnisse, die zur Benennung einer eingebetteten Ressource benutzt wurden!

Auflisten aller eingebetteten Ressourcen

Für diesen Zweck kommt die *GetManifestResourceNames*-Methode der *Assembly*-Klasse zur Anwendung.

Beispiel 8.2: Auflisten aller eingebetteten Ressourcen

C#

Es wird die aktuelle Assembly geladen. Die Namen aller darin enthaltenen Ressourcen werden in einer *ListBox* angezeigt.

```
using System.Reflection;
using System.IO;
...

Assembly ass = Assembly.GetExecutingAssembly();string [] resNames =
ass.GetManifestResourceNames();
listBox1.Items.Clear();
if( resNames.Length > 0 )
{
   listBox1.BeginUpdate();
   foreach(string resName in resNames)
   {
     listBox1.Items.Add(resName);
   }
   listBox1.EndUpdate();
}
```

Die Inhalte eingebetteter Ressourcen auslesen

Hierfür verwenden Sie die *GetManifesResourceStream*-Methode der *Assembly*-Klasse:

Beispiel 8.3: Eine in der eigenen Assembly eingebettete Bildressource wird angezeigt.

C#

```
...
using System.Reflection;
using System.IO;
...
Assembly ass = Assembly.GetExecutingAssembly();
if (pictureBox1.Image != null)
{
   pictureBox1.Image.Dispose();
}
Stream stream = ass.GetManifestResourceStream(
                " _8._1_ManifestRessource.Fuesschen.bmp");
pictureBox1.Image = new Bitmap(stream);
```

Die *GetManifestResourceStream*-Methode hat zwei Überladungen. Die erste erwartet den Namen der Ressource, die zweite erwartet stattdessen einen Typ und einen String. Das vereinfacht die Namensgewinnung für die Ressource: Intern wird der Namespace des Typs genommen und mit dem String wird der Name vervollständigt.

Einige Klassen des .NET Framework verwenden dieses Verfahren ebenfalls, aber anstatt *Stream*-Objekte zurückzugeben, erzeugen sie damit ein bestimmtes Objekt. So hat die *Bitmap*-Klasse einen Konstruktor, der eine eingebettete Ressource in ein *Bitmap*-Objekt laden kann.

Beispiel 8.4: Eine deutlich kürzere Version des obigen Beispiels

C#

```
if (pictureBox1.Image != null)
{
      pictureBox1.Image.Dispose();
}
pictureBox1.Image = new Bitmap(typeof(Form1), "Fuesschen.bmp");
```

Dem *Bitmap*-Konstruktor wird auf diese Weise mitgeteilt, eine Ressource zu finden, deren *Namespace* dem von *Form1* entspricht (weil *Form1* im Rootverzeichnis des Projekts liegt).

Eine weitere Variante der zweiten Zeile:

```
pictureBox1.Image = new Bitmap(GetType(), "Fuesschen.bmp");
```

8.2 Typisierte Ressourcen

Typisierte Ressourcen bauen auf einfachen Manifestressourcen auf. Es handelt sich hierbei um Zusammenstellungen von Schlüssel-Wert-Paaren, wobei der Schlüssel ein eindeutiger String und der Wert ein beliebiges Objekt ist. Im Gegensatz zu den Manifestressourcen sind diese Ressourcen typisiert und wesentlich effizienter, da sie nicht streambasiert arbeiten. Die Bereitstellung erfolgt in der Regel als Datei mit der Extension *.resources*.

8.2.1 Erzeugen von .resources-Dateien

Dazu benötigen Sie einen *ResourceWriter* aus dem Namespace *System.Resources*.

Beispiel 8.5: Erzeugen einer *.resources*-Datei

C#

Die Methode *createResources* erzeugt eine Ressourcendatei, die Texte für beliebige Meldungen enthält.

```
        void CreateResources()
        {
            ResourceWriter rw = new
ResourceWriter("Messages.resources");
```

Die String-Ressourcen als Schlüssel-Wert-Paar hinzufügen:

```
            rw.AddResource("1", "Wahrscheinlich haben Sie recht.");
            rw.AddResource("2", "Ja, natürlich.");
            rw.AddResource("3", "Versuchen Sie es später noch einmal!");
            rw.AddResource("4", "Bitte keine Ablenkungsmanöver!");
            rw.AddResource("5", "Leider nein.");
            rw.AddResource("6", "Wie ich sehe - ja!");
```

Die Datei *Messages.resources* wird erzeugt und geschlossen:

```
            rw.Generate();
            rw.Close();
        }
```

Nach Aufruf von *createResources()* werden Sie im Unterverzeichnis *\bin\Debug* die Datei *Messages.resources* vorfinden.

8.2.2 Hinzufügen der .resources-Datei zum Projekt

Die erzeugte Ressourcendatei *Messages.resources* fügen Sie – genauso wie oben für eine Manifestressource beschrieben – dem Projekt hinzu. Wählen Sie also den Menüpunkt **Projekt | Vorhandenes Element hinzufügen...** und setzen Sie die *Buildvorgang*-Eigenschaft auf *Eingebettete Ressource*. Alternativ können Sie auch das *Hinzufügen*-Kontextmenü im Projektmappen-Explorer verwenden.

HINWEIS: Da die Datei *Messages.resources* nach dem Kompilieren in der Assembly eingebettet ist, können Sie sie auch aus dem Verzeichnis *\bin\Debug* löschen.

8.2.3 Zugriff auf die Inhalte von .resources-Dateien

Für den Zugriff spielt die Klasse *ResourceManager* (Namespace *System.Resources*) eine wichtige Rolle. Sie benötigt zur Instanziierung zwei Parameter: einen Verweis auf die *resources*-Datei, die als Manifestressource in der Assembly abgelegt ist, sowie einen Verweis auf die Assembly selbst.

Der *ResourceManager* stellt für den Zugriff auf Ressourcenelemente die Methoden *GetString()*, *GetObject()* und *GetStream()* bereit. Anzugeben ist der Name des Ressourcenelements unter Beachtung der Groß-/Kleinschreibung. Die Bedeutung der Groß-/Kleinschreibung kann aber mittels *IgnoreCase*-Eigenschaft deaktiviert werden.

Beispiel 8.6: Zugriff auf die Inhalte von .resources-Dateien

C#

Der Zugriff auf die im Projekt *ResourcesTest* eingebettete Datei *Messages.resources* erfolgt über die *GetString*-Methode des *ResourceManager*, wobei der Methode der Schlüssel als Parameter übergeben wird. Um einen einfachen Test zu ermöglichen, erzeugen wir den Schlüssel mit einem Zufallsgenerator, der zugehörige Wert wird in einem *Label* angezeigt.

```
using System.Resources;
using System.Reflection;
...
ResourceManager rm =  new ResourceManager(
        "8._2_TypisierteRessourcen.Messages",
Assembly.GetExecutingAssembly());
Random z = new Random();
string num = z.Next(1, 6).ToString();
label1.Text = rm.GetString(num);
```

Bemerkungen zum Zugriff auf .resources-Dateien

- Die Extension *resources* ist **kein** Bestandteil des Namens, der dem Konstruktor als Argument übergeben wird.
- Das Objekt für die Assembly, in der sich der laufende Code befindet, erhält man über die Methode *System.Reflection.Assembly.GetExecutingAssembly()*.
- Die *GetString()*-Methode des *ResourceManager* ist eine typsichere Alternative zur *GetObject()*-Methode.

8.2.4 ResourceManager einer .resources-Datei erzeugen

Wollen Sie die Ressourcen nachträglich manipulieren, so ist die gezeigte Vorgehensweise nicht geeignet, da die *.resources*-Datei unmittelbar nach ihrem Erzeugen erst per Hand in das Projekt „eingebettet“ werden muss und nach dem Kompilieren – wie jede andere Manifestressource auch – nicht nachträglich manipuliert werden kann.

Ein Ausweg bietet sich, wenn Sie die Datei nicht in die Assembly einbetten, sondern als separate Datei mitführen. Dann können Sie einen passenden *ResourceManager* mithilfe der statischen Methode *CreateFilebasedResourceManager()* erzeugen.

Beispiel 8.7: Erzeugen eines *ResourceManager* aus der Datei *Messages.resources*.

C#

```
ResourceManager rm = ResourceManager.CreateFileBasedResourceManager(
                                    "Messages", Application.StartupPath, null);
```

In diesem Fall befindet sich die Datei *Messages.resources* im selben Verzeichnis wie die Assembly.

Bemerkungen

- Das Pendant zum *ResourceWriter* ist der *ResourceReader*, mit dem sich – als Alternative zum gezielten Zugriff per *ResourceManager* – komplette Ressourcendateien verarbeiten lassen.
- Bequemer als per Code können Ressourcen auch mit dem in Visual Studio integrierten Ressourceneditor angelegt werden (Menü **Projekt | Neues Element hinzufügen... | Ressourcendatei**) .

8.2.5 Was sind .resx-Dateien?

Ressourcen liegen in *resources*-Dateien in einem binären Format vor. Anstatt, wie oben gezeigt, mittels *RessourceWriter* lassen sie sich aber auch aus **.resx*-Dateien erzeugen.

HINWEIS: *resx*-Dateien sind XML-Dokumente!

Zur Umwandlung einer binären *resources*- in eine XML-basierte *resx*-Datei können Sie das Tool *resgen.exe* (*Resource File Generator*) einsetzen.

Visual Studio bietet einen komfortablen Editor für *resx*-Dateien. Der Aufruf erfolgt über den Menüpunkt **Projekt | Neues Element hinzufügen... | Ressourcendatei** oder einfach durch Doppelklick auf eine vorhandene *resx*-Datei im Projektmappen-Explorer.

HINWEIS: *resx*-Ressourcendateien haben besondere Bedeutung im Zusammenhang mit der Lokalisierung von Anwendungen. Dieses Thema wird in Abschnitt 8.4 behandelt.

8.3 Streng typisierte Ressourcen

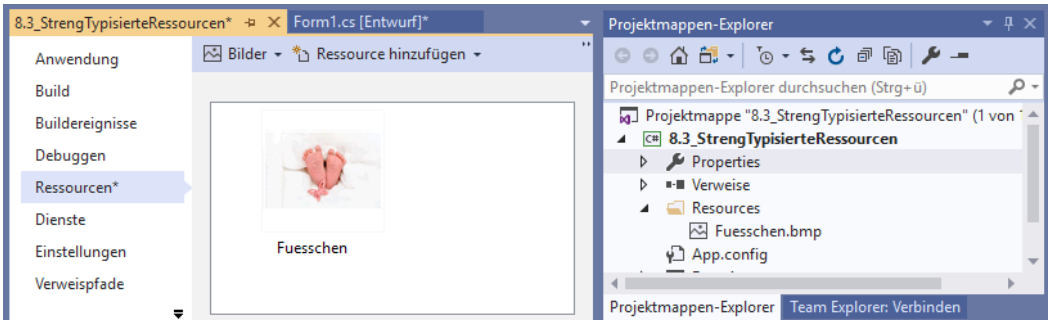
Beim Arbeiten mit dem *ResourceManager*-Objekt kann es häufig zu Fehlern kommen, da die Namen der Ressourcenelemente als Zeichenketten vorliegen und Tippfehler dazu führen, dass das Element nicht gefunden wird. Aus diesem Grund werden im .NET-Framework auch streng typisierte Ressourcen (*strongly-typed resources*) unterstützt.

So bietet das Tool *resgen.exe* die Option, eine Wrapper-Klasse für eine beliebige Ressourcendatei zu generieren, sodass der Programmierer mit *early binding* auf die Ressourcennamen zugreifen kann. Laufzeitfehler aufgrund falscher Ressourcennamen lassen sich so vermeiden.

8.3.1 Erzeugen streng typisierter Ressourcen

Visual Studio stellt einen Designer bereit, der unter anderem auch das automatische Generieren von Wrapper-Klassen für Ressourcen übernimmt. Die Bedienung ist sehr einfach:

Über das Menü **Projekt | ...-Eigenschaften...** (oder durch Doppelklick auf den *Properties*-Eintrag im Projektmappen-Explorer) öffnen Sie die Seite „Ressourcen“ des Projektdesigners.



Im Ergebnis wurde innerhalb des Projektverzeichnisses ein neues Unterverzeichnis namens */Resources* angelegt, in dem die Bitmap gespeichert ist.

8.3.2 Verwenden streng typisierter Ressourcen

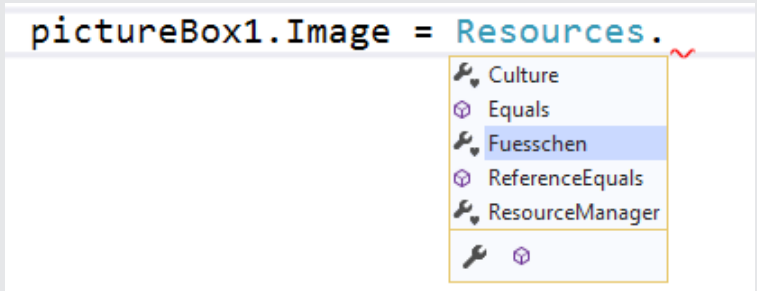
Die Verwendung der streng typisierten Ressourcen ist recht simpel, der Zugriff erfolgt über das *Properties.Resources*-Objekt.

Beispiel 8.8: Anzeige einer Bitmap

C#

```
pictureBox1.Image = Resources.Fuesschen;
```

Da auch eine Unterstützung per IntelliSense erfolgt, wird die Fehleranfälligkeit deutlich verringert:



8.3.3 Streng typisierte Ressourcen per Reflection auslesen

Für den Zugriff auf Ressourcen wie Grafiken, Videos, Sound etc. bietet sich die *GetManifestResourceStream*-Methode an.

Beispiel 8.9: Alle Ressourcen ermitteln

C#

```
...
using System.Reflection;
using System.IO;
using System.Resources;
using System.Collections;
```

Assembly laden:

```
            Assembly assembly = Assembly.GetExecutingAssembly();
            foreach (string s in assembly.GetManifestResourceNames())
            {
```

Hier bestimmen wir zunächst die einzelnen Ressource-Streams:

```
                listBox1.Items.Add(s);
                if (s.ToLower().EndsWith(".resources"))
```

Wenn in diesem *Stream* weitere Ressourcen enthalten sind:

```
                {
                  Stream stream = myass.GetManifestResourceStream(s);
                  using(ResourceReader reader = new ResourceReader(stream))
                  {
```

```
                    IDictionaryEnumerator id = reader.GetEnumerator();
                    while (id.MoveNext())
                    {
```

ID.key bezeichnet die gleiche Ressource, die Sie auch mit *Properties.Resources.xyz* auslesen können:

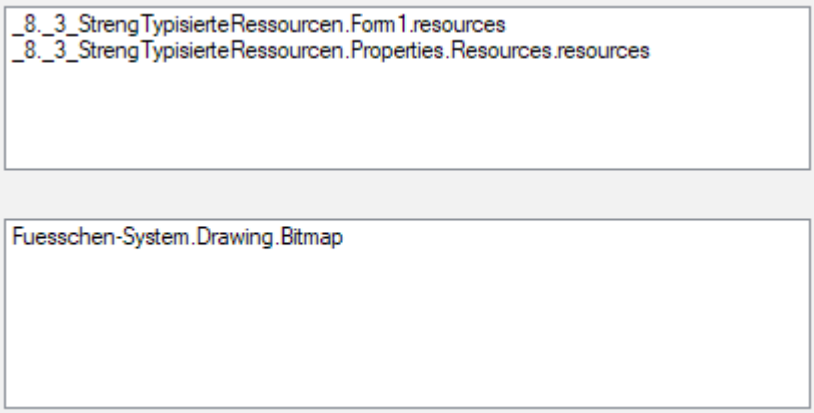
```
                        listBox2.Items.Add($"{id.Key}-{id.Value}");
```

Über *id.Value* können wir direkt auf die einzelnen Bitmaps zugreifen:

```
                        if (id.Value is Bitmap)
                        {
                            Bitmap bmp = (Bitmap)(id.Value as Bitmap)
                                                .Clone();
                            bmp.Save(id.Key + ".bmp");
                        }
                    }
                }
            }
        }
```

Ergebnis

Die Anzeige in *listBox1* und *listBox2*:



HINWEIS: Nach dem Ausführen des obigen Beispiels (siehe Buch-Beispieldaten) werden alle Bitmaps aus der Assembly extrahiert.

8.4 Anwendungen lokalisieren

Das .NET Framework ermöglicht die komfortable Lokalisierung von Anwendungen. Texte und andere sprachabhängige Informationen befinden sich nicht mehr im eigentlichen Quellcode, sondern werden in eigenen Assemblies (sogenannte Satelliten-Assemblies) bereitgestellt, die parallel zur eigentlichen Assembly existieren, darüber hinaus jedoch keinen Code beinhalten.

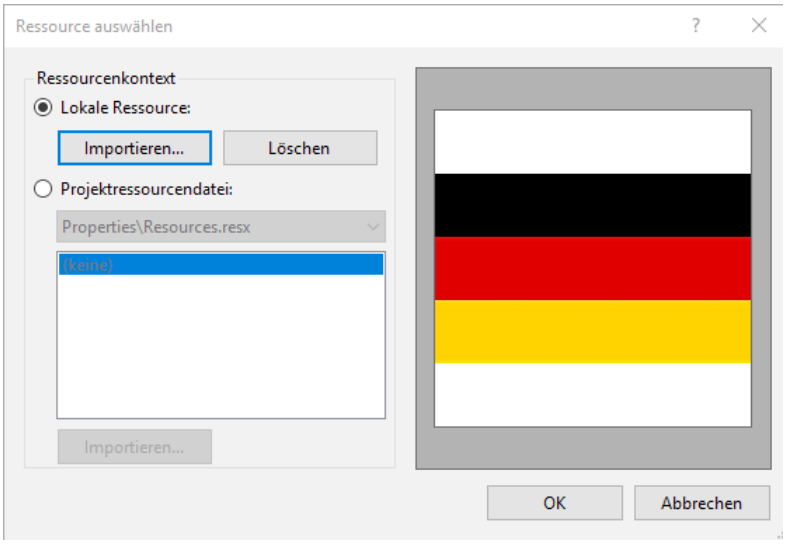
8.4.1 Localizable und Language

Visual Studio kann automatisch die Ressourcendateien für die zu lokalisierenden Elemente der Benutzerschnittstelle, wie z. B. Beschriftungen der Steuerelemente eines Formulars, erzeugen. Von Bedeutung sind dabei die Formulareigenschaften *Localizable* und *Language.*

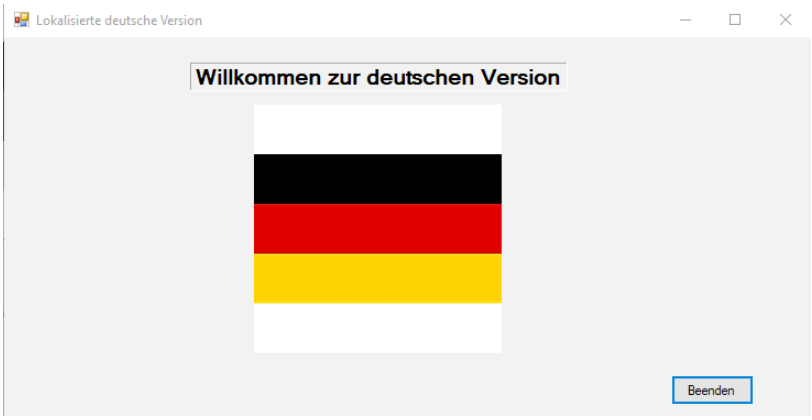
HINWEIS: *Localizable* und *Language* sind „künstliche" Formulareigenschaften. Sie sorgen lediglich dafür, dass der Designer bei der Code-Generierung die zu lokalisierenden Informationen in der richtigen *.resx*-Datei ablegt.

8.4.2 Beispiel „Landesfahnen"

In einer neuen Windows-Anwendung setzen wir die *Localizable*-Eigenschaft von *Form1* auf *True*, die *Language*-Eigenschaft verbleibt zunächst auf ihrem Standardwert (*Default*). Auf *Form1* befinden sich ein *Label*, eine *PictureBox* und ein *Button*. Der *Image*-Eigenschaft der *PictureBox* weisen wir per Dialog eine Bitmap mit der deutschen Flagge zu:



Die Beschriftung erfolgt in deutscher Sprache, sodass die Standardversion etwa so aussieht:

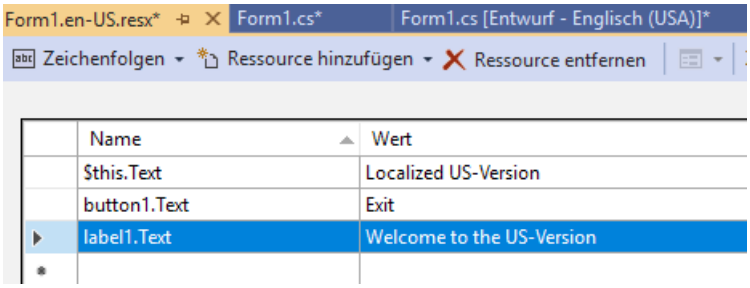


Nun stellen wir im Eigenschaftenfenster von *Form1* die *Language*-Eigenschaft auf *Deutsch* ein. Dazu selektieren wir unser Land aus einer schier endlos langen Liste, in der (fast) jedes Land der Erde vertreten ist.

Wählen Sie im Anschluss *Englisch (USA)* als neue *Language* und gestalten Sie auf analoge Weise die US-Version von *Form1*:

Ein Blick in den Projektmappen-Explorer zeigt, dass die Ressourcendateien *Form1.de-DE.resx* und *Form1.en-US.resx*, welche die sprachabhängigen Informationen kapseln, hinzugekommen sind.

Wie z. B. ein Doppelklick auf *Form1.en-US.resx* zeigt, sind die Text-Ressourcen als Schlüssel-Wert-Paar hinterlegt:



Nach dem Kompilieren werden wir zunächst nur die deutsche Version des Programms zu Gesicht bekommen, es sei denn, wir ändern die Spracheinstellung des aktuellen Threads. Dazu fügen wir die fett gedruckten Programmzeilen hinzu:

HINWEIS: Nach wie vor haben wir es mit einem einzigen Formular (*Form1*) zu tun, es ist deshalb völlig egal, von welcher Formularansicht aus wir das Quellcodefenster öffnen.

```
using System.Globalization;
using System.Threading;
...
        public Form1()
        {
```

Wichtig ist, dass die Einstellung der *CurrentUICulture* **vor** dem Aufruf von *InitializeComponent()* erfolgt:

```
            // Thread.CurrentThread.CurrentUICulture = new CultureInfo("de-DE");
             Thread.CurrentThread.CurrentUICulture = new CultureInfo("en-US");
             InitializeComponent();
        }
    }
}
```

Bemerkungen

- Neben der Hauptassembly wurden zwei neue Ordner (*de* und *en-US*) erzeugt, die jeweils eine Satelliten-Assembly **.resources.dll* enthalten (Satelliten-Assemblies bestehen nur aus Ressourcen).
- Die Original-Bilddateien der deutschen und der amerikanischen Flagge sind nur für den Entwurf des Projekts, nicht aber für die Weitergabe der kompilierten Anwendung erforderlich, da auch die kompletten Bildressourcen in den Satelliten-Assemblies eingelagert sind.

- Die Lokalisierung von .NET-Anwendungen beschränkt sich nicht nur auf das Anlegen von Text-Ressourcen. Nahezu jede Eigenschaft, wie z. B. die Höhe oder Breite eines jeden Formulars oder Steuerelements, lässt sich landesspezifisch anpassen.
- Zwar ist es auch möglich, Grafiken und andere Dateien direkt als Ressourcen in eine Assembly einzubetten oder mit der Assembly zu verlinken, allerdings unterstützen diese Ressourcen nicht die Lokalisierung, denn dazu müssen Grafiken etc. in eine *.resx*-Datei integriert werden.
- Die Spracheinstellung des aktuellen Threads richtet sich beim Programmstart nach den Spracheinstellungen von Windows. Zur Laufzeit können Sie die Sprache wechseln, z. B. mit:

```
Thread.CurrentThread.CurrentUICulture =
          new System.Globalization.CultureInfo("en-GB");
```

- Die aktuelle Sprache ermitteln Sie mit

```
System.Threading.Thread.CurrentThread.CurrentUICulture.Name;
```

oder

```
System.Threading.Thread.CurrentThread.CurrentUICulture.NativName;
```

Name liefert den englischen Sprachnamen, *NativName* den Namen der Sprache in der aktuellen Sprache.

HINWEIS: Für Werte aus der Systemsteuerung wie Datum/Uhrzeit oder Währung ändern Sie wie gerade gezeigt zusätzlich die Eigenschaft CurrentCulture des CurrentThread-Objektes.

9 Komponenten-entwicklung

Die komponentenbasierte Entwicklung gehört mit zu den Grundpfeilern der .NET-Philosophie. Visual Studio bietet vielfältige Features, die dem Programmierer die Entwicklung eigener Steuerelemente (Komponenten) erleichtern. Was Sie dabei erleben, ist OOP pur, und wir setzen für die Lektüre dieses Kapitels voraus, dass Sie einigermaßen sattelfest in Begriffen wie Klassen, Vererbung, Konstruktor usw. sind.

9.1 Überblick

Bevor Sie sich auf die Komponentenprogrammierung stürzen, sollten Sie sich gut überlegen, welchen Komponententyp Sie wirklich benötigen, bietet Ihnen doch Visual Studio gleich ein ganzes Arsenal von Möglichkeiten an:

Typ	Bemerkung/Verwendung
Benutzer-steuerelement	Sie möchten ein oder mehrere Controls in einem Container zusammenfassen und mit einer neuen Schnittstelle/Funktionalität ausstatten. Ableitung von *UserControl*.
Benutzerdefiniertes Steuerelement	Sie wollen ein neues Control erstellen oder ein vorhandenes um zusätzliche Funktionen erweitern. Ableitung vom Urtyp *Control* bzw. von vorhandenen Steuerelementen.
Geerbtes Benutzer-steuerelement	Sie möchten ein Benutzersteuerelement aus Ihrem oder aus anderen Projekten um weitere Funktionen/Controls erweitern.
Komponentenklasse	Sie wollen ganz weit unten anfangen und sich sowohl um Schnittstelle als auch Funktionalität komplett selbst kümmern. Ableitung von *Component*.

Unter ähnlichen und leicht verwechselbaren Namen (dem Übersetzer sei Dank!) verbergen sich teilweise grundverschiedene Ansätze. Im Folgenden wollen wir Ihnen deshalb zunächst die Grundkonzepte und Unterschiede an kleineren Beispielen vorstellen, bevor wir die

Gemeinsamkeiten bei der Definition von Eigenschaften und Methoden bzw. beim Auslösen von Ereignissen besprechen.

Im Anschluss beschäftigen wir uns noch mit einigen interessanten Fragen im Zusammenhang mit der Komponentenentwicklung.

9.2 Benutzersteuerelement

Hierbei handelt es sich quasi um einen Container für beliebige Steuerelemente, die damit zu einer Einheit verschmolzen werden können. Die Entwicklung erfolgt (wie bei einer normalen Windows-Anwendung) rein visuell, Sie platzieren die konstituierenden Steuerelemente im Container und legen deren Eigenschaften fest. In einem weiteren Schritt können Sie Ihrem Benutzersteuerelement ein eigenes Interface mit Eigenschaften, Methoden und Ereignissen geben.

Damit dürften sich auch schon die Vor- und Nachteile dieses Typs klar herausstellen:

- Die Entwicklung von Benutzersteuerelementen ist, dank visueller Unterstützung, recht einfach und schnell möglich.
- Die Programmierlogik zwischen den enthaltenen Controls kann einfach realisiert werden und ist in der Gesamtkomponente gekapselt.
- Eigenschaften und Methoden der enthaltenen Komponenten können sicher vor dem Anwender des Benutzersteuerelements ausgeblendet werden.
- Nachteilig ist der teilweise erhebliche Aufwand für das Erstellen einer sinnvollen Programmierschnittstelle (Eigenschaften/Methoden/Ereignisse).

Damit dürfte sich dieser Komponententyp hauptsächlich für Routine-Programmieraufgaben anbieten, bei denen eine öfter wiederkehrende Logik in einer Komponente gekapselt werden soll.

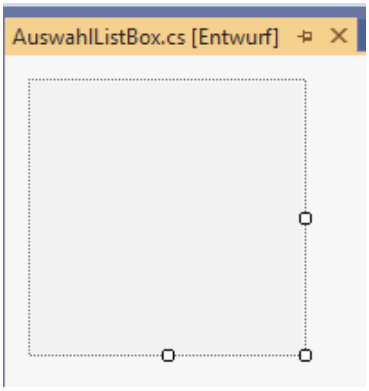
9.2.1 Entwickeln einer Auswahl-ListBox

HINWEIS: Um Ihnen den Test der neuen Komponenten möglichst einfach zu machen, integrieren wir die Komponente in ein normales Windows-Projekt. Im Normalfall werden Sie die Komponente sicher in einer „Windows-Steuerelemente-Bibliothek“ unterbringen, da nur so eine einfache Wiederverwendbarkeit gegeben ist.

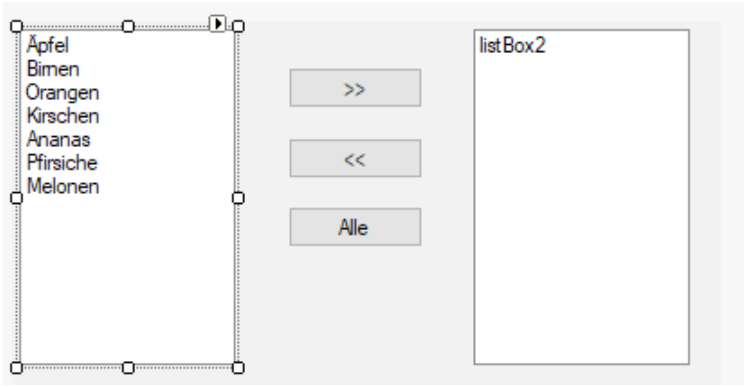
Erstellen Sie zunächst eine neue Windows Forms-Anwendung und fügen Sie über den Menüpunkt **Projekt | Benutzersteuerelement hinzufügen** ein neues Benutzersteuerelement unter dem Namen *AuswahlListBox.cs* hinzu.

Oberflächendesign

Im Designer finden Sie jetzt bereits den „Container“ für die zu platzierenden Steuerelemente vor:



Die Optik und das Handling dürften Ihnen vom normalen Formularentwurf her bereits bekannt vorkommen, platzieren Sie einfach zwei *ListBox*en und drei *Button*s innerhalb des obigen Steuerelements:



Der linken *ListBox* (*listBox1*) fügen Sie im Eigenschafteneditor einige Einträge (über die *Items*-Eigenschaft) hinzu.

Implementieren der Programmlogik

Jetzt noch schnell etwas Code hinzufügen und fertig ist die neue Komponente.

Löschen der bisherigen Einträge und Kopieren **aller** Einträge in *listBox2*:

```
        private void Button3_Click(object sender, EventArgs e)
        {
            listBox2.Items.Clear();
            listBox2.Items.AddRange(listBox1.Items);
        }
```

Verschieben eines Eintrags von *listBox2* in *listBox1*:

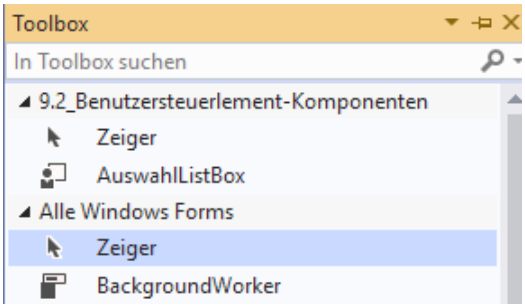
```
private void Button2_Click(object sender, EventArgs e)
{
    try
    {
        listBox1.Items.Add(listBox2.Items[listBox2.SelectedIndex]);
        listBox2.Items.RemoveAt(listBox2.SelectedIndex);
    }
    catch (Exception)
    {
        MessageBox.Show("Keine Auswahl getroffen!");
    }
}
```

Verschieben eines Eintrags von *listBox1* in *listBox2*:

```
private void Button1_Click(object sender, EventArgs e)
{
    try
    {
        listBox2.Items.Add(listBox1.Items[listBox1.SelectedIndex]);
        listBox1.Items.RemoveAt(listBox1.SelectedIndex);
    }
    catch (Exception)
    {
        MessageBox.Show("Keine Auswahl getroffen!");
    }
}
```

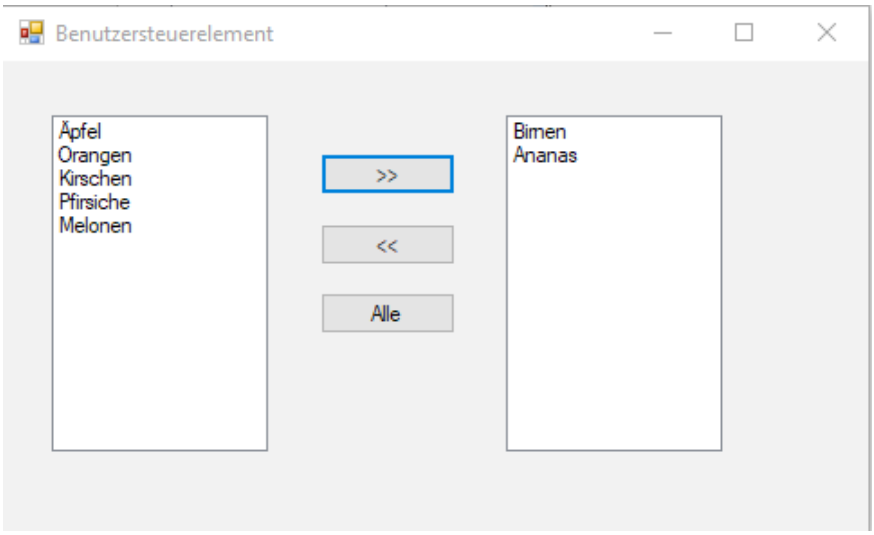
9.2.2 Komponente verwenden

Zunächst kompilieren Sie die aktuelle Anwendung. Ein Blick in den Werkzeugkasten sollte im Erfolgsfall bereits die neue Komponente zeigen:



Mit dem bereits im Projekt vorhandenen Formular können wir uns jetzt an einen ersten Test wagen. Fügen Sie die neue Komponente ein und starten Sie das Programm. Bereits jetzt verfügt Ihre Komponente über jede Menge Funktionalität. Was fehlt, ist jedoch eine Interaktion mit dem eigentlichen Programm. Zu diesem Zweck können Sie weitere Eigenschaften

einführen, um zum Beispiel die Einträge in *listBox2* abzufragen oder die Einträge von *listBox1* vorzudefinieren. Doch dazu später mehr.



9.3 Benutzerdefiniertes Steuerelement

Ein benutzerdefiniertes Steuerelement verwenden Sie, wenn Sie von vorhandenen Controls (z. B. *TextBox*, *Label*, *Timer* etc.) bzw. vom Urtyp *Control* eine neue Komponente **ableiten** wollen. Ihr Steuerelement erbt zunächst alle Eigenschaften, Ereignisse und Methoden des Vorgängers.

Sie können darauf aufbauend

- eigene Eigenschaften,
- Methoden und
- Ereignisse implementieren sowie
- Methoden und Ereignisse ausblenden.

HINWEIS: Die äußeren Abmessungen der Komponente sind zunächst durch den Vorfahren bestimmt. Um ein Zeichnen der Komponente mittels *Paint* brauchen Sie sich nicht zu kümmern, solange Sie nicht *Control* als Vorfahren verwenden.

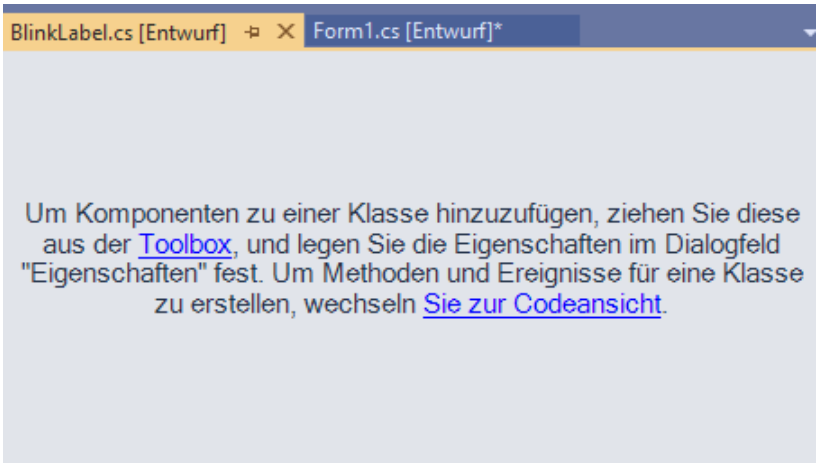
9.3.1 Entwickeln eines BlinkLabels

HINWEIS: Um den Test der neuen Komponenten möglichst einfach zu gestalten, integrieren wir die Komponente in ein normales Windows-Projekt. Im Normalfall werden Sie die Komponente in einer „Windows-Steuerelemente-Bibliothek“ unterbringen, da nur so eine einfache Wiederverwendbarkeit gegeben ist.

Wie beim Benutzersteuerelement erstellen wir zunächst eine neue Windows Forms-Anwendung und fügen über den Menüpunkt **Projekt | Neues Element hinzufügen** ein neues *Benutzerdefiniertes Steuerelement* unter dem Namen *BlinkLabel.cs* hinzu.

Oberflächendesign

Zum Projekt wurde automatisch die folgende Ansicht hinzugefügt:



Nicht sehr viel Ähnlichkeit mit einem *Label*, werden Sie sicher bemerken, aber hier handelt es sich lediglich um einen Dummy, der für alle Klassen bzw. Vorfahren gleich aussieht.

Festlegen des Typs des Vorfahrens

Ein Blick in die Liste der Eigenschaften zeigt bereits jetzt jede Menge Properties. Doch ach, bisher wird *System.Windows.Forms.Control* als Klassentyp angezeigt, was auch richtig ist, da wir noch keinen eigenen Vorfahrtyp bestimmt haben. Das wollen wir nun nachholen, indem wir in die Code-Ansicht wechseln (Doppelklick auf den Dummy).

Hier suchen Sie die Klassendeklaration

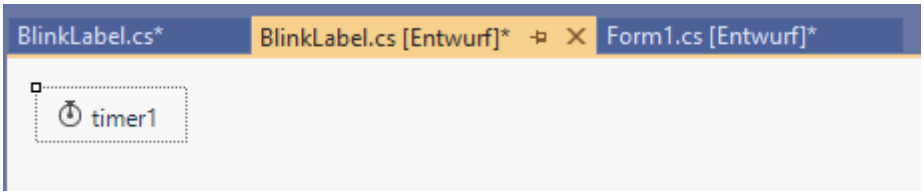
```
using System;
...

namespace _9._3_BenutzerdefiniertesSteuerelement
{
```

```
    public partial class BlinkLabel : Control
    {
...
```

… und ersetzen sie durch die folgende Zeile:

```
...
    public partial class BlinkLabel : Label
...
```

Bei einem Blick in die Eigenschaftenliste werden Sie alle Label-spezifischen Eigenschaften und Ereignisse vorfinden. Wechseln Sie nun wieder zurück zum Dummy und fügen Sie einen *Timer* ein. Das Ganze sieht im Moment zwar etwas merkwürdig aus, aber es funktioniert.



HINWEIS: Sie können den *Timer* natürlich auch per Code erzeugen, was sicher eleganter ist, doch wir bohren diesmal das Brett an der dünnsten Stelle.

Implementieren der Programmlogik

Nach einem Doppelklick auf den *Timer* und dem Wechsel in die Codeansicht dürfte sich Ihnen der folgende Anblick bieten:

```
using System;
...
namespace Beispiel1
{
    public partial class BlinkLabel : Label
    {
        public BlinkLabel()
        {
            InitializeComponent();
            timer1.Interval = 500;
            timer1.Start();
        }

        protected override void OnPaint(PaintEventArgs pe)
        {
            base.OnPaint(pe);
        }

        private void Timer1_Tick(object sender, EventArgs e)
        {
            this.Visible = !this.Visible;
```

```
        }
    }
}
```

Fügen Sie lediglich die fett hervorgehobenen Codezeilen ein, um das Steuerelement über den Konstruktor zu initialisieren und mittels *Timer* ein- und auszublenden. Das war es auch schon und wir können uns nun um das Einbinden des Steuerelements kümmern.

Kompilieren Sie jedoch zunächst die Anwendung, um das neue Steuerelement in die Toolbox aufzunehmen.

9.3.2 Verwenden der Komponente

Öffnen Sie das zum Projekt gehörende Formular *Form1* und platzieren Sie das *BlinkLabel* auf dem Formular, um sich von der Funktionstüchtigkeit zu überzeugen.

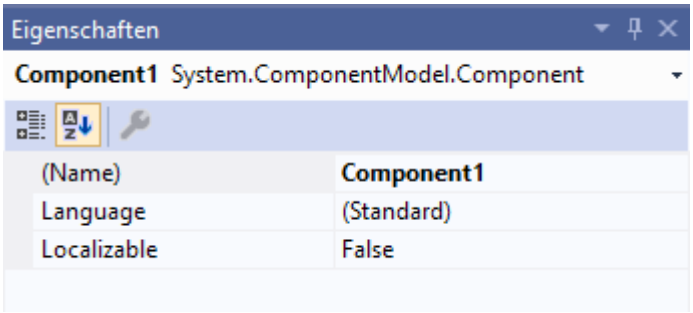
Bereits im Entwurfsmodus beginnt das Label nervend zu blinken!

HINWEIS: Natürlich handelt es sich hier nur um ein ziemlich simples Steuerelement ohne weitere Eigenschaften. Haben Sie sich aber durch dieses Kapitel durchgekämpft, sollte es Ihnen möglich sein, zum Beispiel die Blinkfrequenz über eine eigene Eigenschaft festzulegen oder weitere Ereignisse an die Komponente zu binden.

9.4 Komponentenklasse

Möchten Sie ganz unten anfangen und lediglich die rudimentärsten Funktionen übernehmen, verwenden Sie eine Komponentenklasse.

Die vorgegebene Basisklasse *System.ComponentModel.Component* können Sie übernehmen. Bei einem Blick auf die Eigenschaftenliste werden Sie aber feststellen, dass es sich um einen absoluten Basistyp handelt, der erst mit Leben befüllt werden muss:



Dieser Steuerelementetyp eignet sich zum Beispiel für den Entwurf nicht sichtbarer Komponenten (Multimedia-Timer, Datenbank-Objekte etc.).

Doch was unterscheidet eine Komponentenklasse eigentlich von einer ganz trivialen Klasse?

- Die Fähigkeit, im Designer angezeigt zu werden und Eigenschaften/Ereignisse im Eigenschaftenfenster zu präsentieren, dürfte schnell erkannt sein.
- Ein Blick in den Quellcode zeigt uns jedoch auch, dass die Komponente in der Lage ist, weitere Komponenten aufzunehmen (*IContainer*):

```
namespace _9._3_BenutzerdefiniertesSteuerelement
{
    partial class Component1
    {

        public Component1()
        {
            InitializeComponent();
        }

        public Component1(IContainer container)
        {
            container.Add(this);
            InitializeComponent();
        }

    }
```

Damit können Sie auch hier per Drag&Drop zum Beispiel einen *Timer* oder ein Data Control einfügen, dessen Eigenschaften konfigurieren und Ereignisse programmieren.

Ob Sie sich nun für das Erstellen einer Klasse oder einer Komponentenklasse entscheiden, hängt nur davon ab, wie viel Komfort Sie dem Endanwender bieten wollen.

HINWEIS: Auf ein eigenes Beispiel verzichten wir an dieser Stelle, da der prinzipielle Ablauf der Vorgehensweise beim *Custom Control* entspricht.

9.5 Eigenschaften

Im Folgenden wollen wir uns mit den verschiedenen Varianten und Optionen von Eigenschaften näher befassen.

HINWEIS: Die Ausführungen lassen sich auf alle der eingangs genannten drei Steuerelementetypen anwenden, auch wenn wir uns in den folgenden Beispielen auf benutzerdefinierte Steuerelemente beschränken werden.

9.5.1 Einfache Eigenschaften

Unter dieser Rubrik wollen wir Eigenschaften verstehen, die auf einfachen Basistypen (zum Beispiel *String* oder *Integer*) basieren. Im Eigenschaftenfenster haben Sie die Möglichkeit, den Wert zu editieren (nur wenn die Eigenschaft dies auch zulässt).

Folgende Steuerungsmöglichkeiten und Optionen für die Eingabe bestehen:

- nur Lesezugriff,
- Schreib-/Lesezugriff,
- Schreibzugriff,
- Ausblenden im Eigenschaftenfenster,
- Wertebereichsbeschränkung und Fehlerprüfung,
- Hinzufügen von Beschreibungen,
- Default-Werte,
- Einfügen in Kategorien.

9.5.2 Schreib-/Lesezugriff (Get/Set)

Hierbei dürfte es sich um die wohl am häufigsten verwendete Variante bei Eigenschaften handeln. Der Nutzer des Steuerelements hat die Möglichkeit, sowohl Eigenschaftswerte zu lesen als auch zu ändern. Dazu müssen Sie als Entwickler sowohl die *get*- als auch die *set*-Option implementieren.

HINWEIS: Schreib-/Lesezugriff (mit Zugriffsmethoden)

Beispiel 9.1: Definition einer Property

C#

Die beiden Zugriffsmethoden:

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

9.5.3 Nur-Lese-Eigenschaft (ReadOnly)

Möchten Sie dem Anwender lediglich einen Lesezugriff auf den Eigenschaftswert gestatten, lassen Sie bei der Deklaration der Eigenschaft einfach die *set*-Option weg.

Beispiel 9.2: Nur-Lese-Eigenschaft

C#

Die Zugriffsmethode:

```
public int MeineReadOnlyEigenschaft { get; }
```

Ergebnis

Bei Verwendung der Komponente wird die Eigenschaft im Eigenschaftenfenster in grauer Schrift angezeigt, da sie schreibgeschützt ist.

9.5.4 Nur-Schreib-Zugriff (WriteOnly)

Bei dieser Variante wird die *get*-Option weggelassen.

```
private int meineWriteOnlyEigenschaft;
public int MeineWriteOnlyEigenschaft
{
    set { meineWriteOnlyEigenschaft = value; }
}
```

Allerdings sollten Sie für derartige Anwendungsfälle besser eine Methode verwenden, da dies den Sinn der Operation besser verdeutlicht.

HINWEIS: Die Eigenschaft wird sinnigerweise nicht im Eigenschaftenfenster angezeigt, welcher Wert sollte auch dargestellt werden?

9.5.5 Hinzufügen von Beschreibungen

Mit dem Attribut *Description* steuern Sie den Inhalt des Beschreibungsfelds im Eigenschaftenfenster.

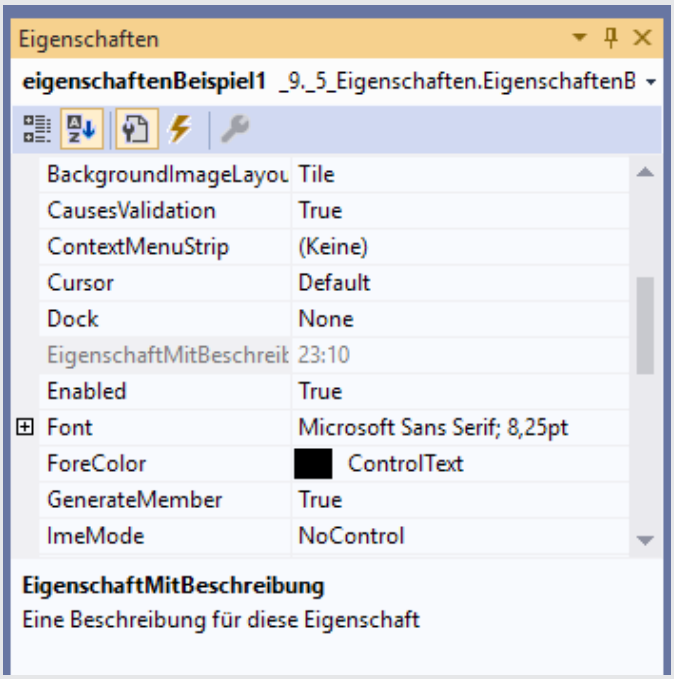
Beispiel 9.3: Eine Beschreibung festlegen

C#

```
[Description("Eine Beschreibung für diese Eigenschaft")]
public string EigenschaftMitBeschreibung
{
    get { return DateTime.Now.ToShortTimeString(); }
}
```

Ergebnis

Die Eigenschaft im Eigenschaftenfenster:



9.5.6 Ausblenden im Eigenschaftenfenster

Nicht in jedem Fall möchte man, dass eine Eigenschaft schon zur Entwurfszeit im Eigenschaftenfenster angezeigt wird. Über das Attribut *Browsable* haben Sie die Möglichkeit, die Sichtbarkeit der Eigenschaft zu steuern.

Beispiel 9.4: Einfügen eines Attributs

C#

```
[Browsable(false)]
 public int NichtSichtbareEigenschaft { get; set; }
```

9.5.7 Einfügen in Kategorien

Auch hier dient ein Attribut (*Category*) dazu, den Eigenschaften weitere Informationen für den Eigenschafteneditor mit auf den Weg zu geben.

Beispiel 9.5: Einfügen von *MeineIntegerEigenschaft* in die Kategorie „Nutzlose Eigenschaften“

C#

```
[Category("Nutzlose Eigenschaften")]
public int MeineIntegerEigenschaft { get; set; }
```

Ergebnis

Das Resultat im Eigenschaftenfenster:

HINWEIS: Dieses Attribut wirkt sich natürlich nur aus, wenn die Eigenschaften auch in Kategorien angezeigt werden.

9.5.8 Default-Wert einstellen

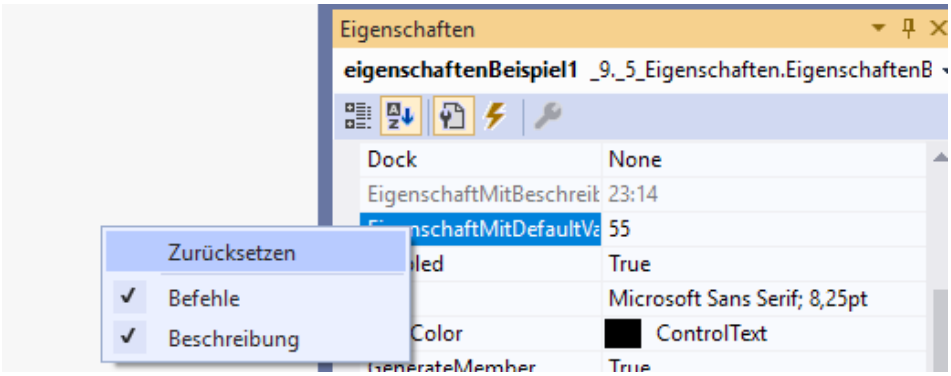
Mithilfe des Attributs *DefaultValue* können Sie dem Anwender die Möglichkeit geben, über Reset bzw. Zurücksetzen der Eigenschaft einen vorgegebenen Wert zuzuweisen.

Beispiel 9.6: Standardwert zuweisen

C#

```
[DefaultValue(55)]
public int EigenschaftMitDefaultValue { get; set; }
```

Im Eigenschaftenfenster erreichen Sie die gewünschte Funktion über das Kontextmenü:



9.5.9 Standardeigenschaft (Indexer)

Da Programmierer so wenig wie möglich schreiben wollen, wurden für den Zugriff auf Array-Eigenschaften die Indexer erfunden.

Der Vorteil: Sie können auf die Angabe eines Eigenschaftennamens verzichten und direkt mit dem Objekt arbeiten.

Beispiel 9.7: Standardeigenschaft definieren

C#

```
    public partial class EigenschaftenBeispiel : Control
    {
...
        private string[] mydata = {"rot","gelb","grün"};

        public string this[int index]
        {
            get { return mydata[index]; }
        }
```

Verwenden können Sie die Klasse später wie folgt:

```
        private void button1_Click(object sender, EventArgs e)
        {
            Text = eigenschaftenBeispiel1[0];
        }
```

HINWEIS: Sie können pro Klasse nur einen Indexer definieren.

9.5.10 Wertebereichsbeschränkung und Fehlerprüfung

Hierbei handelt es sich um eine der wohl komplexesten Aufgaben des Programmierers. Es geht darum, Fehleingaben des Anwenders zu verhindern bzw. die Eingabe auf gewünschte Werte zu beschränken. Dazu stehen dem Entwickler innerhalb der *set*-Zugriffsmethode alle Möglichkeiten offen.

Beispiel 9.8: Wertebereichsbeschränkung und Fehlerprüfung

C#

Eigenschaft, die nur Integerwerte zwischen 50 und 200 akzeptiert und gegebenenfalls eine entsprechende Anpassung vornimmt

```
        private int eigenschaftMitBereichsbeschraenkungeineVar;
        public int EigenschaftMitBereichsbeschränkung
        {
            get { return eigenschaftMitBereichsbeschraenkung ; }
            set
            {
                if (value < 50)
                {
                    eigenschaftMitBereichsbeschraenkung  = 50;
                }
                else
                {
                    if (value > 200)
                    {
                        eigenschaftMitBereichsbeschraenkung  = 200;
                    }
                    else
                    {
                        eigenschaftMitBereichsbeschraenkung  = value;
                    }
                }
            }
        }
```

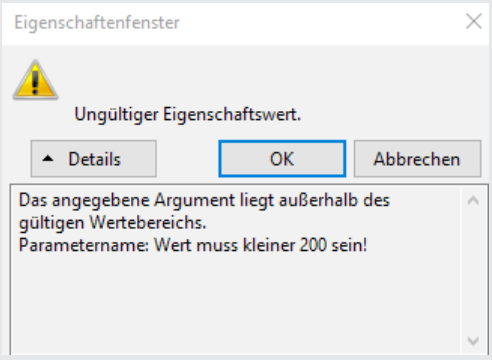
Alternativ können Sie auch den Anwender mit Fehlermeldungen zupflastern:

```
        public int EigenschaftMitFehlerprüfung
        {
            get { return eigenschaftMitBereichsbeschraenkung ; }
            set
            {
                if (value < 50)
                {
                    throw new ArgumentOutOfRangeException("Wert muss größer 50
sein!");
                }
                else
                {
                    if (value > 200)
                    {
                        throw new ArgumentOutOfRangeException(
                              "Wert muss kleiner 200 sein!");
                    }
```

```
                else
                {
                    eigenschaftMitBereichsbeschraenkung  = value;
                }
            }
        }
    }
```

Ergebnis

Fehlermeldung bei Angabe eines falschen Werts:

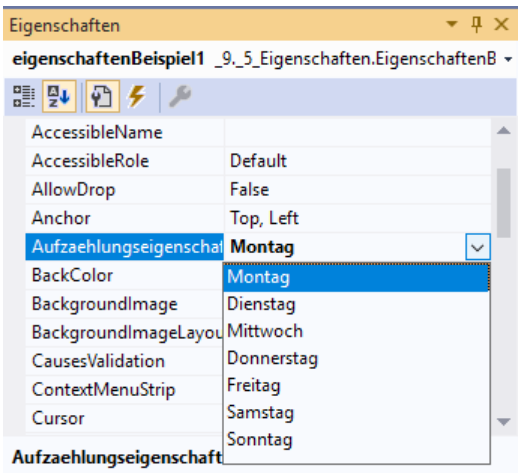


Eine weitere Möglichkeit zur Einschränkung bieten die sogenannten Aufzählungstypen (Enumerationen), die im folgenden Abschnitt besprochen werden.

9.5.11 Eigenschaften von Aufzählungstypen

Ist der Wertebereich einer Eigenschaft zur Entwicklungszeit bereits bekannt, können Sie die Fehlermöglichkeiten durch Verwendung eines Aufzählungstyps einschränken.

Im Eigenschaftenfenster wird in diesem Fall eine Liste der zulässigen Werte angezeigt:



Der Vorteil für den Endanwender liegt auf der Hand: Statt irgendwelcher numerischer Werte erscheinen aussagefähige Beschriftungen. Auf die Verwendung der Hilfefunktion kann deshalb fast immer verzichtet werden.

Für Sie als Entwickler bedeutet ein Aufzählungstyp ein wenig mehr Arbeit, da Sie zuerst einen entsprechenden Typ deklarieren müssen. Die eigentliche Umsetzung in der Komponentendefinition unterscheidet sich nicht von der einer einfachen Eigenschaft.

Beispiel 9.9: Aufzählungseigenschaft deklarieren

C#

Typ deklarieren:

```
public enum Wochentag
 {
     Montag = 0,
     Dienstag = 1,
     Mittwoch = 2,
     Donnerstag = 3,
     Freitag = 4,
     Samstag = 5,
     Sonntag = 6
 }
```

Die Eigenschaft:

```
public Wochentag Aufzaehlungseigenschaft {get; set; }
```

Beispiel 9.10: Verwendung der Eigenschaft *Aufzählungseigenschaft*

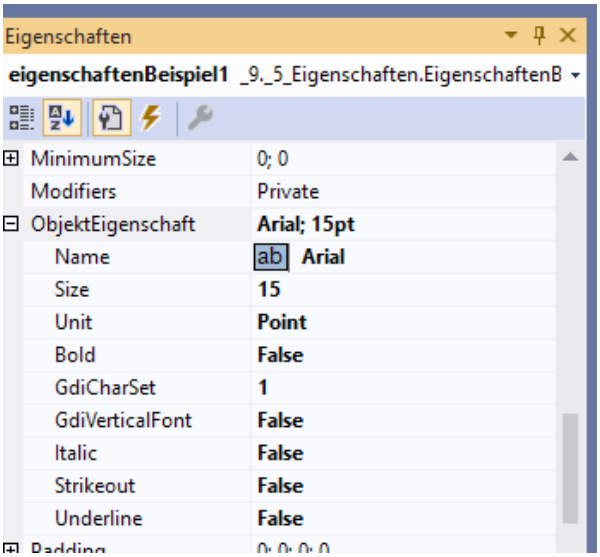
C#

```
...
eigenschaftenBeispiel1.Aufzaehlungseigenschaft =
                          Wochentag.Donnerstag;
```

9.5.12 Standard-Objekteigenschaften

Während Sie beim vorhergehenden Eigenschaftstyp lediglich einzelne Optionen festlegen können, bietet eine Objekeigenschaft wesentlich mehr. Das wohl prominenteste Beispiel dürfte die Eigenschaft *Font* sein, die wiederum über eigene Eigenschaften verfügt.

Im Eigenschaftenfenster können Sie entweder einen Eigenschafteneditor verwenden oder Sie expandieren den Eintrag und legen die Objekteigenschaften einzeln fest:



Beispiel 9.11: Implementieren eines *Font*-Objekts in unserer Beispielkomponente

C#

```
        public Font ObjektEigenschaft { get ; set; } = new Font("Arial",15);
```

9.5.13 Eigene Objekteigenschaften

Komplizierter wird die ganze Geschichte, wenn Sie ein neues Objekt erstellen wollen. In diesem Fall genügt es nicht, wenn Sie einfach ein Objekt in Ihre Komponente integrieren, wie es das folgende Beispiel zeigt.

Beispiel 9.12: Unsere Komponente hat ein Objekt mit drei Eigenschaften vom Typ *Integer.*

C#

```
using System.ComponentModel;
...
    public class Testklasse : ExpandableObjectConverter
    {
        public int Wert1 { get; set; }
        public int Wert2 { get; set; }
        public int Wert3 { get; set; }
    }
```

Die Eigenschaft:

```
...

        public Testklasse Objekteigenschaft2 { get; set; }
```

Ergebnis

Ein Test der Komponente in der IDE zeigt folgendes Ergebnis:

Das Resultat dürfte sicher nicht ganz Ihren Erwartungen entsprechen, haben wir doch einen Fehler in unserem Beispiel, wie er gern gemacht wird:

Beachten Sie bitte, dass die Objektvariable nicht initialisiert ist.

Deshalb:

```
public Testklasse Objekteigenschaft2 { get; set; } = new Testklasse();
```

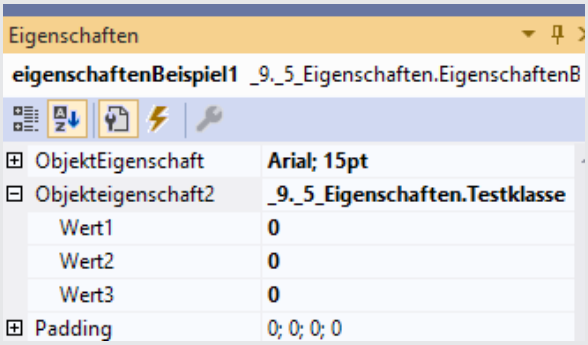
Ein erneuter Blick in das Eigenschaftenfenster zeigt keine Veränderung, aber die Komponente ist zumindest zur Laufzeit schon voll funktionstüchtig:

```
eigenschaftenBeispiel1.Objekteigenschaft2.Wert2 = 5;
Text = eigenschaftenBeispiel1.Objekteigenschaft2.Wert2.ToString();
```

Nach viel Sucherei in der Microsoft-Dokumentation und in diversen Foren stellt sich heraus, dass wir nur mit zusätzlichen Attributen, die die Anzeige im Eigenschaftenfenster steuern, weiterkommen:

```
[TypeConverterAttribute(typeof(System.ComponentModel.ExpandableObjectConverter))]
    public class TestKlasse
    {
```

Das Ergebnis im Eigenschaftenfenster:



HINWEIS: Wem die Anzeige von „TestKlasse“ nicht gefällt, der kann sich einen eigenen *TypeConverter* von *ExpandableObjectConverter* ableiten und die Methoden *CanConvertFrom*, *ConvertFrom*, *ConvertTo*, *GetCreateInstanceSupported* sowie *CreateInstance* überschreiben.

Damit dürften wir die wichtigsten Varianten von Eigenschaften berücksichtigt haben. Auf alle Möglichkeiten (Eigenschafteneditor etc.) können wir aus Platzgründen leider nicht eingehen, die gezeigten Beispiele dürften jedoch manche Unklarheit beseitigt haben.

9.6 Methoden

Hinter den Methoden von Steuerelementen verbirgt sich nichts anderes als normale Funktionen, die jedoch fest an die jeweilige Klasse gekoppelt sind. Aus der Realisierung ergibt sich auch das Einsatzgebiet: Methoden sollten, im Gegensatz zu Eigenschaften, Aktionen auslösen, die teilweise mit Rückgabewerten (Funktionen) verbunden sind. Methoden bieten sich auch dann an, wenn es darum geht, mehrere Eigenschaften gleichzeitig zu beeinflussen.

HINWEIS: Da Definition und Verwendung von Methoden zum Handwerkszeug des C#-Programmierers gehört, möchten wir im Weiteren nur noch auf einige spezielle Themen eingehen.

9.6.1 Konstruktor

Das Besondere: Diese Methode wird automatisch beim Erzeugen einer Klasseninstanz ausgeführt. Damit ist dies auch der ideale Ansatzpunkt, um

- alle internen Variablen unseres Steuerelements zu initialisieren,
- das Aussehen des Steuerelements anzupassen
- und gegebenenfalls Ereignishandler zuzuweisen.

HINWEIS: Der Konstruktor trägt immer den Namen der Klasse.

Eine Besonderheit gilt es noch zu beachten: Erzeugen Sie eine neue Komponentenklasse, legt C# automatisch zwei überladene Konstruktoren an, wie der folgende Quellcodeausschnitt zeigt:

```
namespace _9._6_Methoden
{
    public partial class Component1 : Component
    {
        public Component1()
        {
            InitializeComponent();
        }

        public Component1(IContainer container)
        {
            container.Add(this);
            InitializeComponent();
        }
    }
}
```

Welcher Konstruktor wird nun eigentlich ausgeführt?

Die Antwort: Wenn die Komponente einem Container (*Form/Panel* etc.) zugeordnet wird, nutzt die IDE nicht den einfachen, sondern den Konstruktor mit Parameter, um die Komponente in die *IContainer*-Auflistung einzufügen.

```
    partial class Form1
    {
        private System.ComponentModel.IContainer components = null;
...
        private void InitializeComponent()
        {
            this.component11 = new _9._6_Methoden.Component1(this.components);
        }
...
    }
```

Andernfalls wird der Standardkonstruktor verwendet:

```
    partial class Form1
    {
        private void InitializeComponent()
        {
...
            this.component11 = new _9._6_Methoden.Component1();
...
        }
    }
```

Der Vorteil: Wird ein Formular erzeugt und klinkt sich die Komponente in die *IContainer*-Auflistung *components* ein, wird auch beim Aufruf von *Form.Dispose* die *Dispose*-Methode des Controls aufgerufen. Dazu überschreibt das Formular seine geerbte *Dispose*-Methode:

```
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }
```

Ressourcen werden so definiert freigegeben, andernfalls müsste nach dem Löschen des Formulars Ihre Komponente irgendwann vom Garbage Collector entsorgt werden.

Beispiel 9.13: Formular öffnen und wieder freigeben

C#

```
        Form2 f2 = new Form2();
        f2.ShowDialog();
        f2.Dispose();  // --> Hier wird auch Component.Dispose() aufgerufen
```

9.6.2 Class-Konstruktor

Noch ein Konstruktor? Ja! Wer schon einmal mit dem NET-Reflector in einer NET-Assembly herumgestöbert hat, wird sicher auch die Methode *.cctor* gefunden haben. Hierbei handelt es sich um den **Class-Konstruktor**, der beim ersten Zugriff auf die Klasse ausgeführt wird. Der Einwand, das tut der normale Konstruktor auch, kann so nicht stehen bleiben. Was passiert beispielsweise, wenn Sie eine statische Methode aufrufen? In diesem Fall wird vorher automatisch der Class-Konstruktor abgearbeitet (nach dem Laden der Klasse, vor dem Zugriff auf die Member). Der eigentliche Konstruktor ist zu diesem Zeitpunkt noch gar nicht in Aktion getreten.

Womit auch gleich das Anwendungsgebiet ersichtlich wird. Nutzen Sie diesen Konstruktor, um statische Eigenschaften zu initialisieren.

Beispiel 9.14: Initialisieren der Eigenschaft *Startzeit*

C#

```
    public partial class Component1 : Component
    {
        static public DateTime Startzeit;
        static Component1()   // Class-Konstruktor
        {
            Startzeit = -DateTime.Now;
        }
...
```

HINWEIS: Beachten Sie, dass die Eigenschaft *Startzeit* für alle späteren Instanzen der Klasse den gleichen Wert hat (es handelt sich eben um eine statische Klasseneigenschaft).

Beispiel 9.15: Die Verwendung im aufrufenden Programm

C#

```
        private void Form1_Load(object sender, EventArgs e)
        {
            Text = Component1.Startzeit.ToString();
        }
```

9.6.3 Destruktor

Da sich in .NET bekanntlich der Garbage Collector um die endgültige Zerstörung von nicht mehr benötigten Objekten kümmert, wird ein Destruktor im herkömmlichen Sinn nicht mehr gebraucht. Möchten Sie dennoch auf das relativ unbestimmte Ende Ihres Steuerelements reagieren, können Sie eine private Methode mit dem Namen der Klasse erstellen. Zur Unterscheidung vom Konstruktor wird ein „~“-Zeichen vorangestellt.

Beispiel 9.16: Destruktor

C#

```
~Component1()
 {
   Debug.WriteLine("Ich bin am Ende");
 }
```

HINWEIS: Den Destruktor selbst können Sie nicht per Code aufrufen!

9.6.4 Aufruf von Basisklassen-Methoden

Müssen Sie eine Methode der Basisklasse aufrufen, verwenden Sie den Bezeichner *base*.

Beispiel 9.17: Überschreiben von *OnPaint* und Aufruf der Basisklassen-Methode

C#

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    e.Graphics.DrawLine(Pens.Red, 10, 10, 100, 100);
    ...
}
```

9.7 Ereignisse (Events)

Nicht ganz so einfach wie das Programmieren von Methoden ist die Realisierung von Ereignissen bzw. Ereignisprozeduren. Events stellen einen Mechanismus dar, der ein externes Ereignis (zum Beispiel eine Windows-Botschaft) oder ein Nutzerereignis (zum Beispiel der Klick auf einen Button) mit einer eigenen Routine (Eventhandler) verbindet.

Fünf Schritte sind für das Implementieren eines Events erforderlich:

- eventuell eine neue Klasse für den *EventArgs*-Parameter (diese wird von *System.Event-Args* abgeleitet) definieren,
- eine Delegate-Deklaration (Definition des Ereignistyps),
- ein interner Delegate für den Event,
- eine Event-Deklaration,
- das eigentliche Auslösen des Ereignisses.

9.7.1 Ereignis mit Standardargument definieren

Relativ einfach ist das Implementieren eines Ereignisses, das sich mit den Standardargumenten vom Typ *System.EventArgs* zufriedengibt. Allerdings sollten Sie sich schon hier mit einigen Grundkonventionen und Eigenheiten vertraut machen.

Beispiel 9.18: Ereignis mit Standardargument definieren

C#

Anhand einer *TextBox*, die ein zusätzliches Ereignis erhält, sollen die wichtigsten Schritte erläutert werden.

```
public partial class TextBoxEvent : TextBox
{
    public TextBoxEvent()
    {
        InitializeComponent();
    }
```

Zunächst instanziieren wir das Ereignis *TextboxFull* auf Basis des *EventHandler*-Delegaten:

```
    public event EventHandler TextboxFull;
```

Die Deklaration einer Ereignismethode[1], in der das Ereignis ausgelöst wird:

```
    protected void OnTextBoxFull()
    {
```

Nur wenn dem Ereignis im Programm auch eine Ereignisprozedur zugewiesen wurde, wird auch das Ereignis ausgelöst:

```
        TextboxFull?.Invoke(this, EventArgs.Empty);
        // Ereignis wird ausgelöst

    }
```

Microsoft empfiehlt übrigens, beim Ableiten von Komponenten statt der Verwendung eines Eventhandlers besser die Ereignismethode zu überschreiben.

[1] Entsprechend der Microsoft-Konvention sollten die Ereignismethoden immer mit *On...* beginnen.

Hier überwachen wir die *TextBox* und rufen gegebenenfalls die Methode *OnTextBoxFull* auf:

```
protected override void OnTextChanged(EventArgs e)
{
    if (Text.Length > 8)
    {
        OnTextBoxFull();
    }
    base.OnTextChanged(e);
}
```

Achtung: Vergessen Sie nicht *base.OnTextChanged* aufzurufen, andernfalls fällt das Ereignis *TextChanged* unter den Tisch!

Im Anwenderprogramm findet sich jetzt ein neues Ereignis:

```
private void TextBoxEvent1_TextboxFull(object sender, EventArgs e)
{
    MessageBox.Show("Hilfe nicht so viel Buchstaben ...");
}
```

9.7.2 Ereignis mit eigenen Argumenten

Etwas aufwendiger ist die Definition eigener Argumenttypen.

Beispiel 9.19: Der Parameter soll zusätzlich eine Message an das aufrufende Programm zurückgeben.

C#

Leiten Sie dazu eine Klasse vom Typ *System.EventArgs* ab:

```
public class MyEventArgs : EventArgs
{
```

Die neue Eigenschaft für das Argument:

```
    public string Message { get; set; }
```

Implementieren Sie gegebenenfalls einen neuen Konstruktor für das neue Argument:

```
    public MyEventArgs(string message)
    {
        Message = message;  }
    }
```

Die eigentliche Komponente:

```
public partial class TextBoxEvent : TextBox
{
```

Fügen Sie der Komponente einen Delegaten für den obigen Ereignistyp hinzu:

```
    public delegate void TextBoxFullHandler(object sender, MyEventArgs e);
```

Die Deklaration des Ereignisses:

```
public event TextBoxFullHandler TextBoxFull2;
```

Alternativ können Sie auch die folgende Deklaration nutzen:

```
public event EventHandler<MyEventArgs> TextBoxFull2;
```

Die weitere Verwendung entspricht der bisherigen Vorgehensweise, beim Aufruf des Delegaten wird jetzt jedoch ein anderer Konstruktor genutzt:

```
protected void OnTextBoxFull2()
{
    MyEventArgs args = new MyEventArgs("Ich bin voll!!!!!!");
    TextBoxFull2?.Invoke(this, args);
    }
}
protected override void OnTextChanged(EventArgs e)
{
    if (this.Text.Length > 8)
    {
         OnTextBoxFull();
         OnTextBoxFull2();
    }
    base.OnTextChanged(e);
}
}
```

Das war es auch schon, im Programm können Sie den Parameter wie folgt verwenden:

```
private void TextBoxEvent1_TextBoxFull2 (object sender, MyEventArgs e)
{
    MessageBox.Show(e.Message);
}
```

9.7.3 Ein Default-Ereignis festlegen

Ausnahmsweise ist diese Aufgabe mit einer einzigen Zeile „Code“ erledigt. Fügen Sie einfach vor die betreffende Klassendefinition das Attribut *[DefaultEvent]* ein.

Beispiel 9.20: Default-Ereignis definieren

C#

```
[DefaultEvent("TextBoxFull")]
public partial class TextBoxEvent : TextBox
{
...
 }
```

Nach einem Doppelklick auf die spätere Komponente wird jetzt automatisch eine Ereignismethode für *TextBoxFull* erzeugt.

9.8 Weitere Themen

Dieser Abschnitt fasst in loser Folge einige interessante Themen rund um die Komponentenprogrammierung zusammen.

9.8.1 Wohin mit der Komponente?

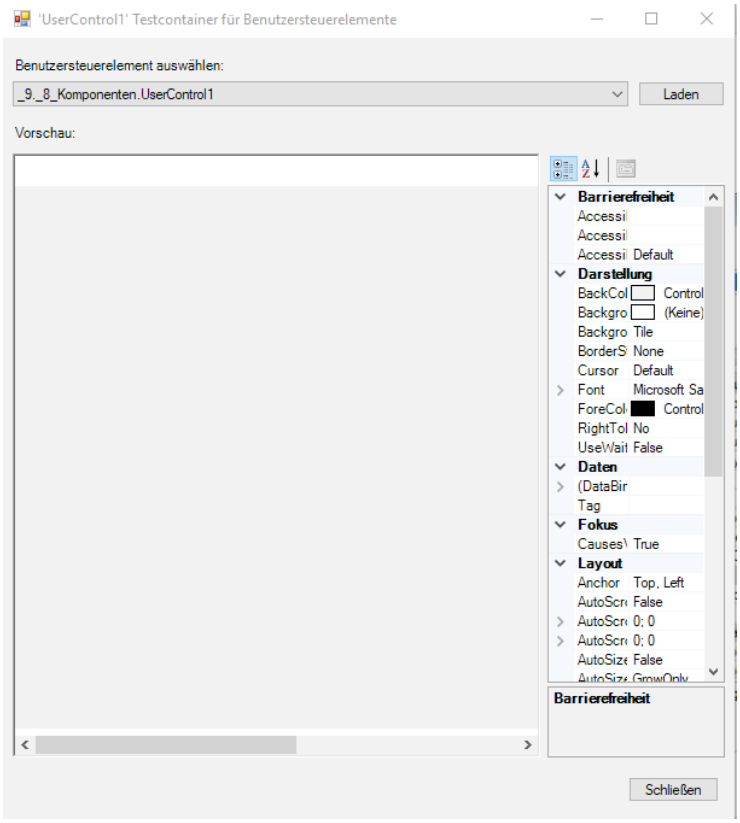
In den bisherigen Beispielen haben wir es uns einfach gemacht und unsere Komponenten jeweils in das Windows-Projekt mit aufgenommen. Das ist zwar sehr praktisch, wenn Sie Komponenten testen wollen, aber im Normalfall soll die Komponente ja in mehreren Anwendungen eingesetzt werden.

Zwei Varianten bieten sich beim Blick in den Dialog „Neues Projekt“ an:

- Klassenbibliothek,
- Windows Forms-Steuerelementebibliothek.

Worin liegt der Unterschied?

Zunächst die allgemeine Antwort: Sie können beide Projektarten für Ihre Steuerelemente verwenden. In jedem Fall wird eine Assembly mit *.dll*-Extension erzeugt.

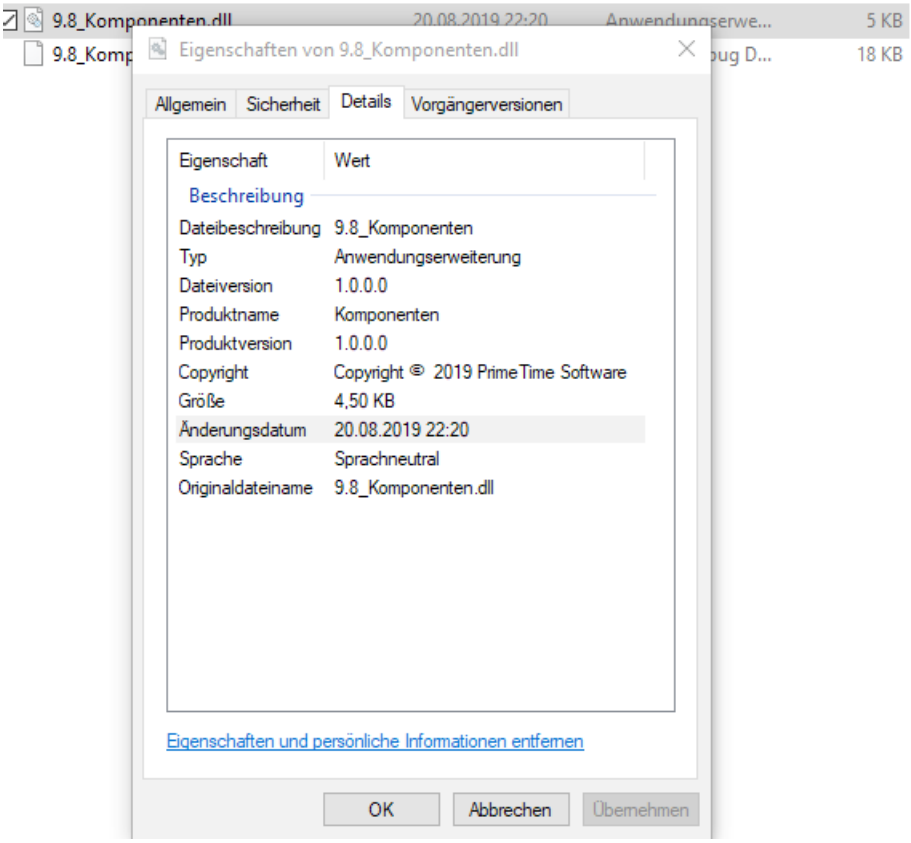


Die Unterschiede zeigen sich bei den Feinheiten. So bietet die „Windows Forms-Steuerelementebibliothek“ bereits eingebettete Ressourcen sowie einen einfachen Test-Container für Ihre UserControls. Diesen starten Sie wie gewohnt mit *F5*, ein weiterer Unterschied zu einer einfachen Klassenbibliothek, die Sie zwar erstellen, aber nicht „ausführen“ können.

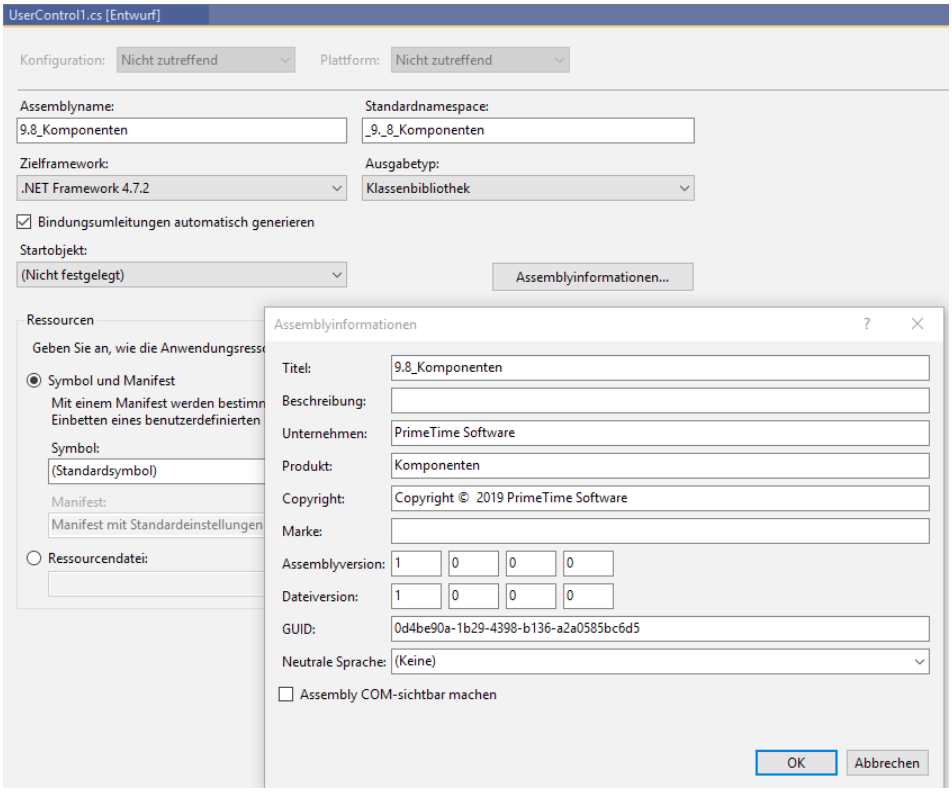
Wie Sie obiger Abbildung entnehmen, beschränkt sich der Test-Container auf die Anzeige der verfügbaren Eigenschaften. Zum Testen Ihrer Komponente ist das sicher ein nettes Feature.

9.8.2 Assembly-Informationen festlegen

In einer Assembly werden neben dem reinen Programmcode auch weitere Informationen abgelegt, die unter anderem für das Erzeugen von *Strong Names* (Starke Namen) Verwendung finden. Doch auch der normale Anwender Ihrer Assembly kann von diesen Informationen profitieren, indem er sich über das Kontextmenü (im Windows Explorer) die Eigenschaften anzeigen lässt.



Die entsprechenden Informationen können Sie in Visual Studio über einen Projekteigenschaften-Dialog festlegen:



Alternativ können Sie Assembly-Infos auch direkt in die Datei *AssemblyInfo.cs* eintragen.

Beispiel 9.21: *AssemblyInfo.cs*

C#

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

// Allgemeine Informationen über eine Assembly werden über die folgenden
// Attribute gesteuert. Ändern Sie diese Attributwerte, um die Informationen zu
ändern,
// die einer Assembly zugeordnet sind.
[assembly: AssemblyTitle("9.8_Komponenten")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("PrimeTime Software")]
[assembly: AssemblyProduct("Komponenten")]
[assembly: AssemblyCopyright("Copyright ©  2019 PrimeTime Software")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
// Durch Festlegen von ComVisible auf FALSE werden die Typen in dieser Assembly
// für COM-Komponenten unsichtbar.  Wenn Sie auf einen Typ in dieser Assembly von
// COM aus zugreifen müssen, sollten Sie das ComVisible-Attribut
```

```
// für diesen Typ auf "True" festlegen.
[assembly: ComVisible(false)]

// Die folgende GUID bestimmt die ID der Typbibliothek,
// wenn dieses Projekt für COM verfügbar gemacht wird
[assembly: Guid("0d4be90a-1b29-4398-b136-a2a0585bc6d5")]

// Versionsinformationen für eine Assembly bestehen aus den folgenden vier Werten:
//
//      Hauptversion
//      Nebenversion
//      Buildnummer
//      Revision
//
// Sie können alle Werte angeben oder Standardwerte für die Build- und
Revisionsnummern verwenden,
// indem Sie "*" wie unten gezeigt eingeben:
// [assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

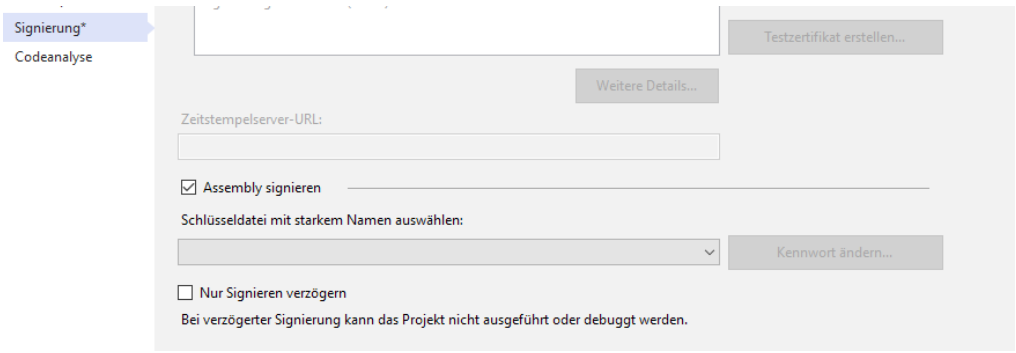
In diesem Zusammenhang ist auch die Bezeichnung *Strong Name* von Bedeutung. Dieser setzt sich aus dem Namen, der Versionsnummer, der Kulturinformation sowie einem öffentlichen Schlüssel und einer digitalen Signatur zusammen. Wie Sie diesen Schlüssel erzeugen, um einen *Strong Name* zu generieren, zeigt der folgende Abschnitt.

9.8.3 Assemblies signieren

Sicher stellt sich auch Ihnen die Frage, ob Sie Ihre Assembly signieren müssen oder nicht. Eine kurze Antwort darauf: Ihre Assembly **müssen** Sie in jedem Fall signieren, wenn Sie diese im *Global Assembly Cache* (GAC) ablegen wollen. Der erzeugte *Strong Name* stellt einen eindeutigen Bezeichner für Ihre Assembly dar.

Erstellen eines AssemblyKey-File

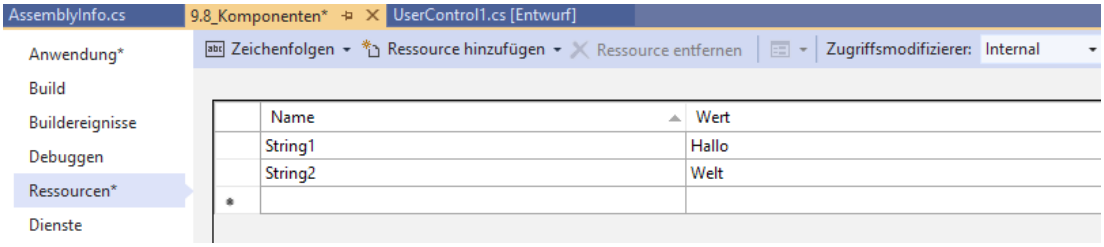
Die zur Signierung erforderliche Schlüsseldatei mit dem Schlüsselpaar wird mit dem NET-Kommandozeilentool *sn.exe* erzeugt. Seit Visual Studio 2005 ist die entsprechende Funktionalität auch über die Projekt-Eigenschaften verfügbar:



Hier erfolgt auch gleich die Verknüpfung mit dem Projekt. Sie brauchen sich also nicht mehr mit Quellcode oder der Kommandozeile herumzuplagen.

9.8.4 Komponentenressourcen einbetten

Eine Komponente besteht meist nicht nur aus Quellcode, sondern erfordert auch die Ausgabe von Grafiken und Sound bzw. diversen Zeichenfolgen etc. Alle diese „Ressourcen" können ganz normal in eine zum Projekt gehörende Ressourcendatei (*Resources.resx*) eingebettet werden und stehen über die Projekteigenschaften zur Verfügung:



Der Zugriff in der Komponente erfolgt dann wie gewohnt per *Properties.Resources....*

9.8.5 Der Komponente ein Icon zuordnen

Nachdem wir uns nun schon geraume Zeit mit der Komponentenprogrammierung herumgeschlagen haben, kommt bei den Ästheten unter den Lesern sicher bald auch der Wunsch nach einer besseren Optik für die selbst erstellten Komponenten auf. Das freudlose Icon, das standardmäßig eingeblendet wird, ist wenig informativ und wohl auch nicht jedermanns Sache.

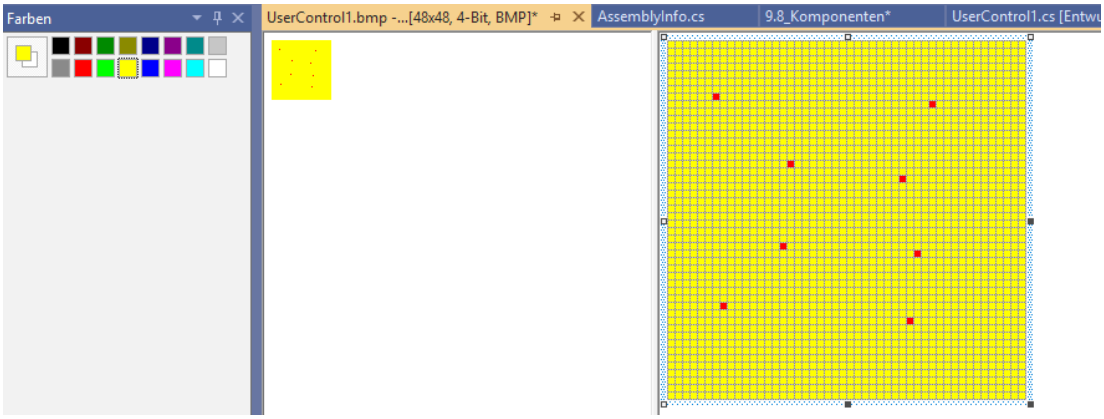
Icon erstellen

Erzeugen (oder kopieren) Sie zunächst eine 16x16 Pixel große Bitmap-Datei. Dazu können Sie zum Beispiel die Visual-Studio-Bordmittel verwenden (**Einfügen | Neues Element Bitmap**).

Speichern Sie jetzt die Datei im Projekt ab (als eingebettete Ressource). Doch schon an dieser Stelle sind einige Konventionen einzuhalten. So leitet sich der Dateiname direkt von der betreffenden Komponentenklasse ab:

```
<Klassenname>.bmp
```

Für unser kleines Beispiel lautet der Name also *UserControl1.bmp*.



HINWEIS: Alternativ können Sie auch das *ToolboxBitmap*-Attribut für die Komponentenklasse verwenden. In diesem Fall brauchen Sie sich nicht auf den Klassennamen zu beschränken.

9.8.6 Den Designmodus erkennen

Nicht immer läuft Ihre Komponente nur im späteren Programm des Anwenders. Auch im Visual-Studio-Form-Designer sollte die Komponente „eine gute Figur machen“. Meist stehen zu diesem Zeitpunkt nicht alle Ressourcen zur Verfügung oder das Verhalten der Komponente soll an den Designer angepasst werden. Doch wie können Sie als Entwickler den Designmodus eigentlich vom normalen Laufzeitmodus unterscheiden?

Die Antwort ist recht einfach, die Eigenschaft *DesignMode* liefert uns die gewünschte Information.

Beispiel 9.22: Unterscheidung Entwurfsmodus/Laufzeitmodus

C#

```
Bitmap bmp = new Bitmap(10,10);
if (DesignMode)
{
    e.Graphics.DrawImage(bmp, 0, 0);
}
else
{
    e.Graphics.DrawImage(bmp, 0, 0);
    ImageAnimator.UpdateFrames();
}
```

9.9 Praxisbeispiele

9.9.1 AnimGif – Anzeige von Animationen

Mit dem folgenden einfachen Beispiel möchten wir Ihnen die bisherigen Ausführungen zur Komponentenentwicklung noch einmal im Zusammenhang demonstrieren.

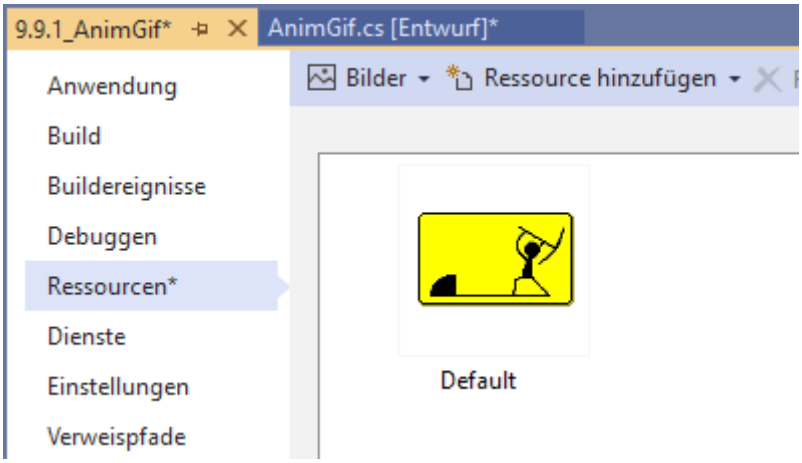
Ziel ist das Erstellen einer Komponente zur Anzeige animierter GIF-Grafiken. Neben dem standardmäßig mitgelieferten Bild soll der Anwender auch eigene Grafiken zuweisen können. Die Komponente soll sich daraufhin an die Abmessungen der Grafik anpassen. Auf diverse Extras, wie Ein-/Ausschalten etc., verzichten wir an dieser Stelle.

Oberfläche/Ressourcen

Erstellen Sie zunächst eine neue Klassenbibliothek und fügen Sie ein *Benutzerdefiniertes Steuerelement* hinzu. Speichern Sie dieses unter dem Namen *AnimGif* ab.

Nachfolgend können Sie sich im Internet auf die Suche nach einer Default-Grafik für Ihr neues Control machen. Unter den Stichworten „Animated GIF“ werden Sie ganz sicher fündig, es gibt Tausende derartiger „Nervtöter“.

Speichern Sie die Grafik auf Ihrem PC ab. Über den Menüpunkt **Projekt | Eigenschaften** können Sie die Grafik den Assembly-Ressourcen zuordnen. Vergeben Sie hier den Namen *Default*:



Damit sind die „Oberflächlichkeiten“ abgeschlossen und wir können uns den Innereien zuwenden.

Quelltext

```
namespace _9._9._1_AnimGif
{
```

Da der Vorfahrtyp *Control* bereits alle Eigenschaften aufweist, die für uns wichtig sind, belassen wir es bei diesem Typ:

```
public partial class AnimGif : Control
{
```

Für die aktuelle Grafik eine private Variable:

```
    private Bitmap bmp;
```

Der Konstruktor kümmert sich um das Auslesen der Grafik aus den Ressourcen und die Vorbereitungen für eine flackerfreie Darstellung (*Double Buffering*):

```
    public AnimGif()
    {
        InitializeComponent();
        SetStyle(ControlStyles.UserPaint, true);
        SetStyle(ControlStyles.AllPaintingInWmPaint, true);
        SetStyle(ControlStyles.DoubleBuffer, true);
        bmp = Properties.Resources.Default;
    }
```

Die neue Eigenschaft *Gif* ermöglicht es dem Anwender, eine neue Grafik zuzuordnen:

```
    public Bitmap Gif
    {
        get { return bmp; }
        set
        {
            bmp = value;
            if (bmp == null)
            {
                bmp = Properties.Resources.Default;
            }
```

Größenanpassung an die Grafik:

```
            Width = bmp.Width;
            Height = bmp.Height;
        }
    }
```

HINWEIS: Wird die Eigenschaft zurückgesetzt, das heißt, der Anwender weist keine Grafik zu, nehmen wir automatisch die Defaultgrafik aus den Ressourcen.

Mit dem Initialisieren des Controls wird es für uns ernst:

```
    protected override void InitLayout()
    {
        base.InitLayout();
```

Handelt es sich nicht um den Designmodus und ist die Grafik animierbar (was für ein Wort!), dann nutzen wir die Klasse *ImageAnimator*, um ein automatisches Umschalten der Einzelbilder per integriertem Timer zu erreichen:

```
            if ((!DesignMode)&(ImageAnimator.CanAnimate(bmp)))
            {
                ImageAnimator.Animate(bmp, OnNextFrame);
            }
        }
```

Diese Methode wird immer aufgerufen, wenn ein neuer Frame angezeigt werden soll:

```
        private void OnNextFrame(object o, EventArgs e)
        {
            Invalidate();
        }
```

Die Hauptarbeit leistet die überschriebene *OnPaint*-Methode:

```
        protected override void OnPaint(PaintEventArgs pe)
        {
            Graphics g = pe.Graphics;
```

Im Designmodus wird immer dasselbe Bild angezeigt:

```
            if (DesignMode)
            {
                 g.DrawImage(bmp, 0, 0);
             }
```

Andernfalls schalten wir zum nächsten Frame um:

```
            else
            {
                g.DrawImage(bmp, 0, 0);
                ImageAnimator.UpdateFrames();
            }
        }
    }
}
```

Test

Kompilieren Sie die Klassenbibliothek und fügen Sie die Komponente in die Toolbox ein. In einem neuen Windows Forms-Projekt können Sie sich von der Funktionstüchtigkeit überzeugen.

9.9.2 Eine FontComboBox entwickeln

Zum Schluss noch ein „quick and dirty“-Beispiel.

Lassen Sie uns die *ComboBox* so modifizieren, dass später ohne zusätzlichen Programmieraufwand alle verfügbaren Schriftarten angezeigt und als fertiger Font abgerufen werden können.

Quelltext

Erster Schritt zur fertigen Komponente ist das Ableiten von einer bereits vorhandenen. Öffnen Sie dazu ein neues Projekt und fügen Sie über den Menüpunkt **Projekt | Neues Element hinzufügen | Benutzerdefiniertes Steuerelement** ein neues Steuerelement in das Projekt ein.

Das bereits bestehende Code-Gerüst ändern Sie dahingehend, dass Sie statt der Basisklasse *Control* eine *ComboBox* verwenden:

```
using System.Drawing.Text;
public partial class FontCombo : ComboBox
```

Die Klasse im Überblick:

```
    public partial class FontCombo : ComboBox
    {
```

Initialisieren:

```
        public FontCombo()
        {
            InitializeComponent();
```

Wir zeichnen die Einträge selbst:

```
            DrawMode = DrawMode.OwnerDrawFixed;
            ItemHeight = 28;
            DropDownStyle = ComboBoxStyle.DropDownList;
            DropDownWidth = 200;
            SetStyle(ControlStyles.Selectable, false);
        }
```

Zur Laufzeit werden die Fonts ermittelt und in der *Items*-Collection gespeichert:

```
        protected override void InitLayout()
        {
            base.InitLayout();
            Items.Clear();
            if (!DesignMode)
            {
                foreach (FontFamily ff in (new InstalledFontCollection().Families))
                    try
                    {
                        Items.Add(new Font(ff, 12));
                    }
                    catch
                    { }
            }
        }
```

Wir müssen uns um das Zeichnen eines Eintrags kümmern:

```
        protected override void OnDrawItem(DrawItemEventArgs e)
        {
```

Keine Auswahl:

```
            if (e.Index == -1)
            {
                return;
            }
```

Hintergrund zeichnen:

```
            e.DrawBackground();
```

Auswahlrechteck zeichnen:

```
            if ((e.State & DrawItemState.Focus) != 0)
            {
                 e.DrawFocusRectangle();
            }
```

String mit dem Namen der Schriftart und dem jeweiligen Font ausgeben:

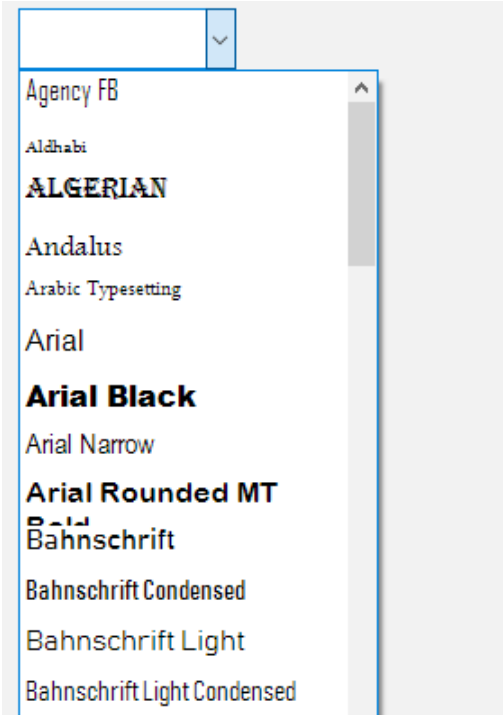
```
            e.Graphics.DrawString((Items[e.Index] as Font).Name,
                        (Items[e.Index] as Font), Brushes.Black, e.Bounds);
        }
```

Über die Eigenschaft *SelectedFont* kann der Nutzer den ausgewählten Font abfragen und gleich nutzen:

```
        public Font SelectedFont
        {
            get
              {
                  if (SelectedIndex == -1)
                  {
                      return null;
                  }
                  else
                  {
                      return (SelectedItem as Font);
                  }
              }
        }
    }
}
```

Test

Erstellen Sie zunächst die Assembly und binden Sie dann die Komponente in den Werkzeugkasten ein. Ein kleines Testprogramm beweist die Funktionsfähigkeit:



Noch einmal: Das war ein **einfach** gehaltenes Beispiel. Sie können natürlich auch unterschiedliche Zeilenhöhen festlegen, die Stringlänge messen usw.

9.9.3 Das PropertyGrid verwenden

Sicher haben Sie sich auch schon gefragt, wie Sie zur Laufzeit die Eigenschaften von Objekten möglichst einfach und ohne großen Aufwand verändern können. Bevor Sie jetzt lange suchen, sollten Sie einen Blick auf die recht unscheinbare *PropertyGrid*-Komponente werfen, die schon zu .NET-1.x-Zeiten ein Aschenputteldasein in der Toolbox fristete.

Für unser Beispiel werden wir eine eigene Klasse definieren, deren Eigenschaften Sie per *PropertyGrid* zur Laufzeit bearbeiten können.

Oberfläche

Fügen Sie in ein Windows-Formular eine *PropertyGrid*-Komponente ein. Das ist zunächst alles.

Quelltext

Zunächst definieren wir die gewünschte Klasse (Sie können das *PropertyGrid* aber auch an jede beliebige Komponente „klemmen“).

```
using System.ComponetModel;

    class Mitarbeiter
    {
        public Mitarbeiter(string name, string vorname, DateTime geboren)
        {
            Name = name;
            Vorname = vorname;
            Geboren = geboren;
            Gehalt = 0;
        }

        [Category("Personaldaten")]
        public string Name { get; set; }

        [Category("Personaldaten")]
        public string Vorname{ get; set; }


        [Category("Lohndaten")]
        public float Gehalt { get; set; }

        [Category("Lohndaten")]
        public int Kinder { get; set; }
        [Category("Personaldaten")]
        public DateTime Geboren { get; set; }

    }
```

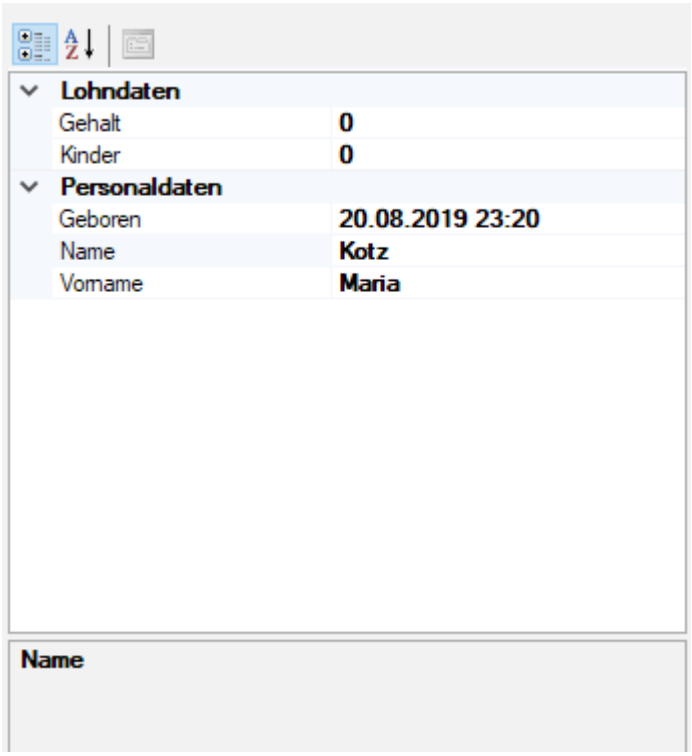
HINWEIS: Das Beispiel eignet sich auch recht gut, wenn Sie den Einfluss von Attributen testen wollen.

Die Verbindung von Grid und späterem Objekt ist absolut trivial:

```
private void Form1_Load(object sender, EventArgs e)
{
   Mitarbeiter mitarbeiter = new Mitarbeiter("Kotz", "Maria", System.DateTime.Now);

   propertyGrid1.SelectedObject = mitarbeiter;
}
```

Test

Beim Start der Anwendung sollten auch schon alle Eigenschaften des Objekts *mitarbeiter* angezeigt werden:


Lohndaten
Gehalt
0
Kinder
0
Personaldaten
Geboren
20.08.2019 23:20
Name
Kotz
Vorname
Maria
Name

Teil II:
WPF-Bonuskapitel

- Drucken in WPF
- Weitere Steuerelemente in WPF
- Verteilen von Anwendungen

10 Druckausgabe mit WPF

Mit der Einführung von WPF wurde auch eine grundsätzlich andere Vorgehensweise bei der Druckausgabe von Dokumenten gewählt, die bei mehr als einer Druckausgabeseite schnell an Komplexität gewinnt. Erschwerend kommt hinzu, dass für Druckausgabe und Druckvorschau[1] keine einheitliche Lösung gewählt wurde, was es dem Programmierer auch nicht einfacher macht.

Aus diesem Grund wollen wir Sie zunächst mit der "Trivial-Lösung" für die Ausgabe einzelner Seiten sowie die Auswahl des Zieldruckers vertraut machen, bevor wir uns den komplexeren Themen wie

- Druckvorschau
- und mehrseitige Dokumente

zuwenden wollen. Doch bevor es so weit ist, wollen wir Sie in einer Kurzübersicht mit den Konzepten der WPF-Druckausgabe vertraut machen.

10.1 Grundlagen

Eine Einführung zum Drucken in WPF wäre nicht komplett, wenn wir nicht kurz auf das Thema XPS-Dokumente eingehen würden.

10.1.1 XPS-Dokumente

Mit XPS (*XML Paper Specification*) versucht Microsoft einen Pendant zum derzeit weit verbreiteten Adobe PDF-Format zu etablieren. Im Grunde handelt es sich um eine geräteunabhängige vektororientierte Seitenbeschreibungssprache, die einen ungehinderten Informationsfluss aus der Anwendung bis zum finalen Ausgabegerät sicherstellen soll. Dafür findet sich nicht zuletzt auch ab Windows Vista ein entsprechender Druckertreiber, dessen Ausga-

[1] Eine direkte Druckvorschau-Komponente werden Sie auch nicht finden, wir nehmen dafür den *DocumentViewer*.

ben in einer Datei landen, die Sie wiederum mit dem entsprechenden Betrachter anzeigen können.

Über die Vor- und Nachteile zum bereits etablierten PDF lässt sicher sicher streiten, was aber nichts an der Tatsache ändert, dass XPS nun mal der zentrale Weg für die Druckausgaben unserer WPF-Anwendungen ist. Allerdings hat auch Microsoft schon richtig eingeschätzt, dass eine relevante XPS-Unterstützung für das Format von kaum einem Druckerhersteller zu erwarten ist, und so wird innerhalb der Druckausgabe eine Umwandlung der XPS-Daten in die bekannten GDI- bzw. PDL-Daten vorgenommen, um auch mit den derzeit am Markt befindlichen Geräten sinnvoll arbeiten zu können.

Um all diese Hintergründe müssen Sie sich als Programmierer jedoch nicht selbst kümmern, das für Sie interessante *XpsDocumentWriter*-Objekt nimmt, je nach Endgerät, die erforderliche XPS zu GDI-Umwandlung automatisch vor.

Für die Interaktion mit dem XPS-Ausgabesystem stehen Ihnen in C# die beiden Namespaces

- *System.Printing*
- und *System.Windows.Xps*

mit den entsprechenden Klassen zur Verfügung. Zusätzlich findet sich ganz unscheinbar im Namespace *System.Windows.Controls* auch eine *PrintDialog*-Komponente, die Sie in keinem Fall mit der entsprechenden Windows Forms-Komponente verwechseln sollten. Die *PrintDialog*-Komponente bietet neben der vermuteten Dialog-Funktionalität vor allem einen einfachen Zugriff auf die installierten Drucker, um zum Beispiel Controls (und das kann auch, wie von WPF gewohnt, eine geschachtelte Anordnung sein) oder auch ganze XPS-Dokumente zu drucken. Mehr dazu finden Sie im Abschnitt 10.2.

10.1.2 System.Printing

Dieser Namespace bietet mit den enthaltenen Klassen (Auszug)

- *PrintServer*
  (repräsentiert den aktuellen Druckserver, d. h. den Computer)
- *PrintQueue*
  (repräsentiert den aktuellen Drucker und dessen Druckerwarteschlange)
- *PrintSystemJobInfo*
  (ein Druckjob und dessen Status)
- *PrintTicket*
  (die Konfiguration des Druckauftrags, wie Seitenformat, Seitendrehung etc.)

die Grundlage für die Arbeit mit der Druckausgabe in WPF.

HINWEIS: Bevor Sie auf diesen Namespace zugreifen können, müssen Sie unter Verweise noch die Assembly *System.Printing.dll* nachträglich einbinden. Je nach Funktionsumfang kann es auch nötig sein, zusätzlich die Assembly *ReachFramework.dll* einzubinden.

10.1.3 System.Windows.Xps

Wie schon erwähnt, ist XPS der Dreh- und Angelpunkt der WPF-Druckausgabe. Über den Namespace *System.Windows.Xps* stehen Ihnen die nötigen Schnittstellen-Klassen *VisualsToXpsDocument* und *XpsDocumentWriter* zur Verfügung.

Zusätzlich finden sich weitere Namespaces wie *System.IO.Packaging* und *System.Windows.Xps.Packaging*, die im Zusammenhang mit der Ausgabe von XPS-Dateien eine Rollen spielen. Hier auf alle Einzelheiten einzugehen, würde den Rahmen des Kapitels sprengen.

HINWEIS: Bevor Sie auf diesen Namespace komplett zugreifen können, müssen Sie Verweise auf die Assemblies *System.Printing.dll* und *ReachFramework.dll* einbinden.

10.2 Einfache Druckausgaben mit dem PrintDialog

Hoffentlich haben wir Sie mit den Kurzausführungen zu XPS nicht ganz verschreckt, nicht in jedem Fall müssen Sie sich mit der Gesamtkomplexität der WPF-Druckausgabe herumschlagen, es geht teilweise auch ganz einfach, wie es auch das folgende Beispiel zeigt:

Beispiel 10.1: Verwendung von *PrintDialog*

C#

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

Instanz erstellen:

```
            PrintDialog dialog = new PrintDialog();
```

Dialog anzeigen:

```
            if (dialog.ShowDialog() == true)
            {
```

Wird auf die Ok-Taste geklickt, drucken wir ein Control aus:

```
                dialog.PrintVisual(Button, "Erster Test");
            }
        }
```

Ergebnis

Das Ergebnis dürfte etwa so aussehen:

Der Druckdialog selbst wird Ihnen sicher von den Windows Forms her noch bekannt sein, die Funktionalität ist zunächst gleich. Doch nach der Auswahl von Drucker, Seitenbereich etc. geht es hier erst richtig los. Über die Methode *PrintVisual* besteht die Möglichkeit, XAML-Elemente Ihres Formulars (oder auch gleich das ganze Formular) direkt an den Drucker zu senden. Da diese ohnehin vektorbasiert sind, ist auch die Druckqualität im Vergleich zu Windows Forms wesentlich besser.

HINWEIS: Wer jetzt befürchtet, immer den Dialog anzeigen zu müssen, liegt falsch, lassen Sie den Aufruf der Methode *ShowDialog* weg, wird automatisch der Standard-Drucker verwendet.

Doch Vorsicht – der oben gezeigte Aufruf von *PrintVisual* hat einige Einschränkungen, die Sie kennen sollten, bevor Sie darauf basierend Ihre Programme umschreiben:

1. Es wird immer nur **eine** Seite bedruckt, ist der zu druckende Content größer, wird er abgeschnitten.
2. Der Ausdruck wird immer an der linken oberen Blattecke ausgerichtet. Da viele Drucker jedoch einen Seitenoffset besitzen, wird in diesem Fall meist etwas am oberen und linken Rand abgeschnitten.

Als Lösung für 1. bietet sich die komplexere *PrintDocument*-Methode an, auf die wir noch zurückkommen werden, Problem 2 können Sie umgehen, wenn Sie dem zu druckenden Control einen entsprechenden *Margin* verpassen, der automatisch mit den Seitenrändern verrechnet wird.

Wo wir schon bei Seitenrändern sind: Über den *PrintDialog* können Sie auch einige wichtige Informationen zum aktuellen Ausgabemedium[2] in Erfahrung bringen:

- Seitenhöhe und Breite (PrintTicket.*PageMediaSize.Height*, *PrintTicket.PageMediaSize.Width*)
- Druckbereichshöhe und -breite (*PrintableAreaHeight*, *PrintableAreaWidth*)

[2] tolle Umschreibung für „Blatt“

- Randloser Druck möglich (*PrintTicket.PageBorderless*)
- Seitenausrichtung (*PrintTicket.PageOrientation*)

HINWEIS: Achtung: Ist das Blatt gedreht, hat dies keinen Einfluss auf die Seiten- bzw. Druckbereichsabmessungen, d. h., die Papierbreite ist beim Querformat mit *PrintableAreaHeight* zu bestimmen.

Doch was ist eigentlich mit reiner Textausgabe? Hier sollten Sie den Namen der *PrintVisual*-Methode nicht zu genau nehmen, mit dieser Methode können Sie natürlich auch Text bzw. ein *TextBlock*-Objekt ausgeben und das auf wesentlich komfortablere Art als bei den Windows Forms. Der *TextBlock* darf natürlich auch über Formatierungen etc. verfügen.

Beispiel 10.2: Textausgabe auf dem Drucker

C#

```
        private void Button_Click_1(object sender, RoutedEventArgs e)
        {
            PrintDialog dialogg = new PrintDialog();
            if (dialog.ShowDialog() == true)
            {
```

Neue *TextBlock*-Instanz unabhängig vom Formular erzeugen:

```
                TextBlock txt = new TextBlock();
```

Ränder festlegen, so umgehen wir das Problem mit dem Seitenoffset:

```
                txt.Margin = new Thickness(15);
```

Hier der eigentliche Ausgabetext:

```
                txt.Text = "Hier folgt ein selten belangloser Text, den Sie besser
nicht lesen sollten. Aber so wird Ihnen schnell klar, wie einfach die Textausgabe
ist.";
```

Schriftgröße und Schriftart bestimmen:

```
                txt.FontSize = 25;
                txt.FontFamily = new FontFamily("Arial");
```

Textumbruch aktivieren:

```
                txt.TextWrapping = TextWrapping.Wrap;
```

Und jetzt müssen wir uns um die Größe des Controls kümmern (Layout-Aktualisierung), sonst ist es nicht zu sehen:

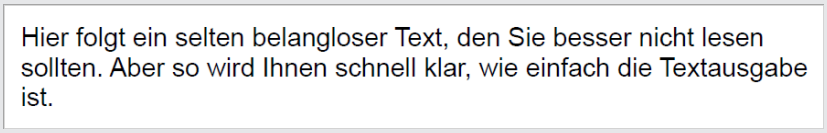
```
                txt.Measure(new Size(dialog.PrintableAreaWidth,
                            dialog.PrintableAreaHeight));
                txt.Arrange(new Rect(0, 0, txt.DesiredSize.Width,
                            txt.DesiredSize.Height));
```

Last, but no least, die Druckausgabe:

```
                dialog.PrintVisual(txt, "Erster Test");
            }
        }
```

Ergebnis

Damit kann man doch schon ganz zufrieden sein:



Damit wollen wir uns aus der „Programmierer-Kuschelecke“ verabschieden und uns in die „Wildnis“ von Druckvorschau und mehrseitiger Ausgabe wagen.

10.3 Mehrseitige Druckvorschau-Funktion

Puh …, ganz schön viele Wünsche auf einmal und WPF lässt uns an dieser Stelle doch etwas im Regen stehen. Ein Blick in den Werkzeugkasten verheißt nichts Gutes, weder für das eine noch für das andere findet sich **die** Standardlösung.

Wer den Blick in die Hilfe bzw. ins Internet wagt, findet sicher auch recht schnell die Standardlösung mit einer abgeleitetem *DocumentPaginator*-Klasse, deren Methoden Sie implementieren müssen. Doch so ganz befriedigend ist diese Lösung nicht, von der Übersicht ganz zu schweigen.

Aus diesem Grund hat sich der Autor diverse Einzellösungen im Internet angesehen und aus all diesen Hinweisen/Lösungsvorschlägen etc. zwei halbwegs nutzbare Lösungen realisiert, die sowohl unsere Wünsche nach mehrseitigen Dokumenten erfüllen als auch eine Druckvorschau bieten.

Doch wie immer findet sich auch hier ein Haar in der Suppe: Sie müssen sich entscheiden, ob Sie mit einem flexiblen Dokument arbeiten, das über Fließtext mit eingebetteten Grafiken etc. verfügt (ein typisches Flow-Dokument) oder ob Sie die einzelnen Elemente Layout-orientiert bzw. absolut anordnen wollen (Fix-Dokument). Für beides finden Sie im Folgenden die Lösung.

10.3.1 Fix-Dokumente

Sehen wir uns zunächst die layout- und seitenorientierte Ausgabe von Dokumenten an. Wie auch bei den noch zu betrachtenden Flow-Dokumenten wollen wir die erforderliche Funktionalität in einer eigenen Klasse kapseln, um uns ganz auf die reine Ausgabe konzentrieren zu können. Ziel war es, mit so wenig Code wie möglich eine Druckausgabe und eine Druckvorschaufunktion zu implementieren.

Einsatzbeispiel

Bevor wir uns in die Details vertiefen, wollen wir uns ansehen, wie wir die spätere Klasse *FixedPrintManager* einsetzen können und was wir alles zu Papier bringen können. Sie werden sehen, dass viele Konzepte bereits in den vorhergehenden Kapiteln erläutert wurden.

Beispiel 10.3: Verwendung der Klasse *FixedPrintManager*

C#

Binden Sie zunächst die beiden Assemblies *System.Printing.dll* und *ReachFramework.dll* ein. Erstellen Sie nachfolgend ein Formular, in das Sie eine Schaltfläche und einen *DocumentViewer*[3] einbinden.

Zunächst die nötigen Namespaces:

```
...
using System.IO;
using System.Windows.Xps;
using System.Windows.Xps.Packaging;
using System.Windows.Markup;
using System.IO.Packaging;
...
```

Nach dem Klick auf die Schaltfläche setzen die hektischen Aktivitäten ein:

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

Wir erstellen eine Instanz unserer *FixedPrintManager*-Klasse:

```
                FixedPrintManager pm = new FixedPrintManager(
                                       new Thickness(15), true);
```

Parameter sind die Breite der Seitenränder und die Option, ob ein Druckerauswahldialog angezeigt werden soll.

Und damit sind wir auch schon beim Darstellen der ersten Seite:

Zentrales Element ist ein *StackPanel*, wie Sie es auch von den WPF-Formularen kennen:

```
                StackPanel panel = new StackPanel();
```

In das *StackPanel* fügen wir einen *TextBlock* ein. Die entsprechende Instanz erzeugen wir allerdings per *NewTextBlock*-Methode unserer *FixedPrintManager*-Instanz:

```
                TextBlock txt = pm.NewTextBlock("Times New Roman", 40);
```

Der Vorteil: Wir müssen uns nicht um das recht umständliche Konfigurieren des *TextBlock*s kümmern, Defaultparameter ermöglichen das einfache und komfortable Setzen der Eigenschaften.

[3] Sie ahnen es sicher schon, der *DocumentViewer* wird unsere Druckvorschau.

Noch etwas Text definieren und den *TextBlock* dem *StackPanel* hinzufügen:

```
                txt.Text =
 "Hier steht zum Beispiel jede Menge Text. Hier steht zum Beispiel
                     jede Menge Text.Hier steht zum Beispiel jede Menge Text.
.......... ";
                panel.Children.Add(txt);
```

Ein weiterer *TextBlock:*

```
                txt = pm.NewTextBlock(
                      text: "Hier steht zum Beispiel jede Menge Text.
                      Hier steht zum Beispiel jede Menge Text.Hier steht
                      zum ..........");
                txt.TextAlignment = TextAlignment.Justify;
                txt.Margin = new Thickness(0, 10, 0, 20);
                panel.Children.Add(txt);
```

Ein *Image* in das *StackPanel* einfügen:

```
                Image img = new Image();
```

Im Gegensatz zu den beiden vorhergehenden Beispielen müssen wir im Fall des *Image*-Controls selbst für die Größenanpassung des *Image* sorgen:

```
                img.Width = pm.PageSize.Width - pm.Borders.Left –
                            pm.Borders.Right;
                img.Stretch = Stretch.Uniform;
                img.Source = new BitmapImage(
                    new Uri("pack://application:,,,/Fuesschen.bmp"));
                panel.Children.Add(img);
```

Wir schließen die erste „Druck“-Seite ab, indem wir das Stackpanel an die *NewPage*-Methode übergeben:

```
                pm.NewPage(panel);
```

Hier erzeugen wir eine reine Textseite:

```
                txt = pm.NewTextBlock(fontsize: 22);
                txt.Text = "Hier ist die zweite Seite mit Fließtext.
                        Hier ist die zweite
                        Seite mit Fließtext. Hier ist die zweite Seite ..... ";
```

Abschließen der zweiten Seite:

```
                pm.NewPage(txt);
```

Ein ganz triviales Beispiel für die Ausgabe einzelner Controls:

```
                pm.NewPage(new Calendar());
```

Natürlich können Sie bei der Druckausgabe auch alle Bildtransformationen nutzen, die Sie bereits kennengelernt haben.

Als Grundlage dient uns in diesem Fall ein *Canvas,* der auch das absolute Positionieren des Controls zulässt:

```
Canvas cv = new Canvas();
img = new Image();
img.Width = pm.PageSize.Width - pm.Borders.Left –
            pm.Borders.Right;
img.Stretch = Stretch.Uniform;
img.Source = new BitmapImage(
    new Uri("pack://application:,,,/Fuesschen.bmp"));
img.SetValue(Canvas.TopProperty, 100.0);
img.SetValue(Canvas.LeftProperty, 1.0);
```

Die Transformation anwenden:

```
img.LayoutTransform = new RotateTransform(45);
cv.Children.Add(img);
pm.NewPage(cv);
```

So, damit ist das Dokument erstellt, Sie haben jetzt zwei Möglichkeiten: Entweder Sie zeigen das Dokument in einem *DocumentViewer* an oder Sie gehen sofort zum Druck über:

```
documentViewer1.Document = pm.Document;
pm.Print();
    }
```

Ergebnis

Das Ergebnis unserer Bemühungen zeigen die folgenden Abbildungen:

Vielleicht sind Sie auch der Meinung, dass der Aufwand für das Erstellen des Dokuments nicht allzu hoch war. Die Verwendung der Textblöcke hatte sogar den Vorteil, dass wir uns um Zeilenumbrüche keinen Kopf machen mussten. Die restlichen Ausgaben waren weitgehend mit der Formulardarstellung identisch, neue Konzepte müssen Sie bei dieser Form der Druckausgabe nicht erlernen.

Die Klasse FixedPrintManager

Doch nun wollen wir einen Blick auf die verwendete Klasse *FixedPrintManager* werfen:

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Markup;
using System.Windows.Media;
```

```
...

    public class FixedPrintManager
    {
```

Eine interne Variable für den PrintDialog:

```
        private PrintDialog dialog;
```

Die *Borders*-Eigenschaft (Abfrage der Seitenränder):

```
        public Thickness Borders { get; }
```

Die *PageSize*-Eigenschaft (Abfrage der Seitengröße, diese wird im Konstruktor gesetzt):

```
        public Size PageSize { get; }
```

Die Rückgabe des Dokuments für die Druckvorschau:

```
        public FixedDocument Document { get ; }
```

Der Konstruktor unserer Klasse:

```
        public FixedPrintManager(Thickness borders, bool showPrintDialog = false)
        {
```

Ein neues *FixedDocument* erstellen:

```
            Document = new FixedDocument();
```

Intern nutzen wir den *PrintDialog* für die Druckausgabe und die Bestimmung von Drucker und Seitenrändern:

```
            dialog = new PrintDialog();
            if (showPrintDialog)
            {
                dialog.ShowDialog();
            }
            PageSize = new Size(dialog.PrintableAreaWidth,
                                dialog.PrintableAreaHeight);
            Document.DocumentPaginator.PageSize = PageSize;
            Borders = borders;
        }
```

Das Starten der Druckausgabe erfolgt mit der folgenden Methode:

```
        public void Print(string title = "Mein Druckauftrag")
        {
            dialog.PrintDocument(_PrintDocument.DocumentPaginator, title);
        }
```

Zum Erstellen einer neuen Seite nutzen Sie folgende Methode:

```
        public void NewPage(UIElement content)
        {
```

Erzeugen einer neuen *FixedPage,* diese wird im Folgenden in das interne *FixedDocument* eingefügt:

```
            FixedPage page = new FixedPage();
```

Seitengröße und -ränder bestimmen:

```
            page.Width = Document.DocumentPaginator.PageSize.Width;
            page.Height = Document.DocumentPaginator.PageSize.Height;
            page.Margin = Borders;
```

Den übergebenen Seiteninhalt einfügen:

```
            page.Children.Add(content);
```

Hier wir es etwas komplizierter, da ein direkter Zugriff auf die *AddChild*-Methode nicht möglich ist:

```
            PageContent pageContent = new PageContent();
            ((IAddChild)pageContent).AddChild(page);
```

Anhängen an das *FixedDocument*:

```
            Document.Pages.Add(pageContent);
        }
```

Last but not least, noch unsere Methode zum Erstellen von Textblöcken. Diese macht ausgiebig von der Verwendung optionaler Parameter Gebrauch und nimmt uns das lästige Parametrieren des *TextBlock*s ab:

```
        public TextBlock NewTextBlock(string fontname = "Arial",
                                      int fontsize = 12, string text = "")
        {
            TextBlock txt = new TextBlock();
            txt.Width = PageSize.Width - Borders.Left - Borders.Right;
            txt.FontFamily = new FontFamily(fontname);
            txt.FontSize = fontsize;
            txt.Text = text;
            txt.TextWrapping = TextWrapping.WrapWithOverflow;
            return txt;
        }
    }
```

Damit haben Sie bereits ein recht umfangreiches Grundgerüst für eigene Erweiterungen. Insbesondere die Ausgabe von Linien-Zeichnungen etc. ist sicher noch verbesserungswürdig, aber Sie wollen ja auch noch etwas zu tun haben.

10.3.2 Flow-Dokumente

Nachdem wir uns mit der seitenorientierten Methode, d. h. den *FixedDocuments* beschäftigt haben, wollen wir und mit den flexibleren Flow-Dokumenten befassen (siehe auch Buchkapitel 9.17 und 9.18). Deren Vorteil liegt in der Beschreibung von formatierten Fließtexten,

die wiederum aus Absätzen (*Paragraph*), eingefügten Controls (*BlockUIContainer*), Auflistungen (*List*), *Section*s und Tabellen (*Table*) bestehen können. Um Seitenränder, Seitengrößen und -ausrichtungen müssen Sie sich zum Zeitpunkt der Dokumenterstellung keinen Kopf machen, dies erfolgt automatisch bei der finalen Ausgabe auf dem jeweiligen Ausgabegerät.

Einführungsbeispiel

Ein kleines Beispiel soll Sie von den Vorzügen unserer *FlowPrintManager*-Klasse überzeugen, die wie auch die *FixedDocument*-Klasse über eine *Document*-Eigenschaft (für die Druckvorschau) und eine *Print*-Methode verfügt.

Beispiel 10.4: Verwendung der *FlowPrintManager*-Klasse

C#

Auch hier benötigen wir die bereits eingebundenen Assemblies *System.Printing.dll* und *ReachFramework.dll*. Erweitern Sie nachfolgend das Window um eine Schaltfläche.

```
        private void Button_Click_1(object sender, RoutedEventArgs e)
        {
```

Eine Instanz unserer Klasse erstellen:

```
            FlowPrintManager fpm = new FlowPrintManager(
                            new Thickness(75, 25,25,25), true);
```

Wir erzeugen einen neuen Abschnitt:

```
            fpm.FlowDoc.Blocks.Add(new Paragraph(new Run("Hier steht zum Beispiel
jede Menge Text. Hier steht zum Beispiel jede Menge Text.Hier steht zum Beispiel
jede Menge Text.Hier steht zum Beispiel jede Menge Text.")));
```

Einen leeren Absatz:

```
            fpm.FlowDoc.Blocks.Add(new Paragraph(new LineBreak()));
```

Und hier noch ein paar weitere Absätze:

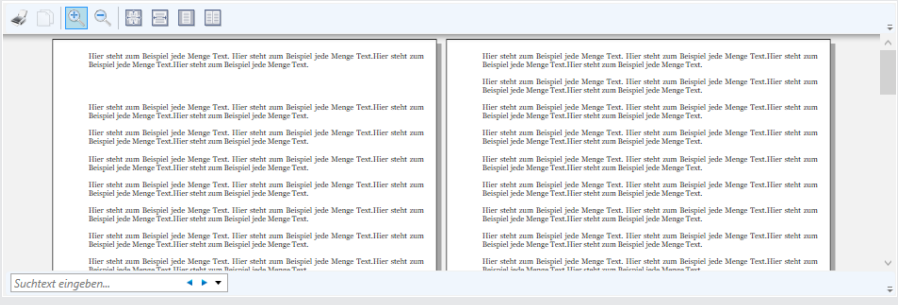
```
            for (int i=0; i<40; i++)
            {
               fpm.FlowDoc.Blocks.Add(new Paragraph(new Run("Hier steht zum
Beispiel jede Menge Text. Hier steht zum Beispiel jede Menge Text.Hier steht zum
Beispiel jede Menge Text.Hier steht zum Beispiel jede Menge Text.")));
            }
```

Und schon können Sie dieses Dokument in der Druckvorschau anzeigen:

```
            documentViewer1.Document = fpm.Document;
        }
```

Ergebnis

Die Druckvorschau bringt es an den Tag, durch unsere kleine Schleife bei der Ausgabe des letzten Absatzes wird der Text so lang, dass er nicht mehr auf eine Seite passt:



Doch wie funktioniert die Umwandlung des oben erzeugten *FlowDocument* in ein *FixedDocument*, das wir für die Druckvorschau (*DocumentViewer*) benötigen?

Die Klasse FlowPrintManager

Hier hilft uns die Möglichkeit weiter, mittels *XpsDocumentWriter* das *FlowDocument* in ein XPS-Dokument zu schreiben und aus diesem wiederum ein *FixedDocument* zu erzeugen. Doch leider ist diese Lösung im Normalfall mit einer physischen Datei verbunden, was nicht nur unschön aussieht, sondern teilweise auch Probleme nach sich ziehen kann. Aus diesem Grund haben wir uns für den Weg über einen *MemoryStream* entschieden, das komplette Handling erfolgt also im Speicher.

Doch nun zu den Details unserer *FlowPrintManager*-Klasse:

```
using System.IO;
using System.IO.Packaging;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Xps;
using System.Windows.Xps.Packaging;
...
    public class FlowPrintManager
    {
```

Die internen Variablen:

```
        private FixedDocumentSequence document;
        private MemoryStream ms;
        private Package package;
        private PrintDialog dialog;
```

Die Eigenschaft *FlowDoc*, über die wir unsere *FlowDocument* zusammenbasteln können:

```
        public FlowDocument FlowDoc { get; }
```

Die Eigenschaft *Document* liefert uns die gewünschte *FixedDocumentSequence* für die Druckvorschau:

```
        public FixedDocumentSequence Document
        {
            get
            {
```

Und hier wird es schnell etwas unübersichtlich. Grundlage des Schreibens von XPS-Dokumenten ist zunächst ein Package (ein Container mit Kompression), in welches das eigentliche XPS-Dokument eingefügt wird.

Am Anfang löschen wir zunächst einmal pauschal ein möglicherweise vom letzten Durchlauf noch vorhandenes Package:

```
                PackageStore.RemovePackage(new Uri("memorystream://data.xps"));
```

Neues Package erzeugen:

```
                PackageStore.AddPackage(new Uri("memorystream://data.xps"),
                              package);
```

Neues *XpsDocument* im Package erzeugen:

```
                XpsDocument xpsDocument = new XpsDocument(package,
                    CompressionOption.Fast, "memorystream://data.xps");
```

XpsDocumentWriter für das *XpsDocument* erstellen:

```
                XpsDocumentWriter writer =
                          XpsDocument.CreateXpsDocumentWriter(xpsDocument);
```

Schreiben der Daten per *XpsDocumentWriter*:

```
                writer.Write(((IDocumentPaginatorSourceFlowDoc).DocumentPaginator);
```

Für das neue XPS-Dokument rufen wir eine *FixedDocumentSequence* ab und geben diese zurück:

```
                document = xpsDocument.GetFixedDocumentSequence();
                xpsDocument.Close();
                return document;
            }
        }
```

Unser Konstruktor kümmert sich um das Initialisieren der internen Variablen:

```
        public FlowPrintManager(Thickness borders, bool ShowPrintDialog = false)
        {
```

MemoryStream und *Package* erzeugen:

```
            ms = new MemoryStream();
            package = Package.Open(ms, FileMode.Create, FileAccess.ReadWrite);
```

Eventuell den *PrintDialog* anzeigen:

```
            dialog = new PrintDialog();
            if (ShowPrintDialog)
            {
                dialog.ShowDialog();
            }
```

Standardeinstellungen vornehmen:

```
            FlowDoc = new FlowDocument();
            FlowDoc.ColumnWidth = dialog.PrintableAreaWidth;
            FlowDoc.PageHeight = dialog.PrintableAreaHeight;
            FlowDoc.PagePadding = borders;
        }
```

Für die direkte Druckausgabe nutzen wir wieder den *PrintDialog*:

```
        public void Print(string title = "Mein Druckauftrag")
        {
            dialog.PrintDocument(Document.DocumentPaginator, title);
        }
    }
```

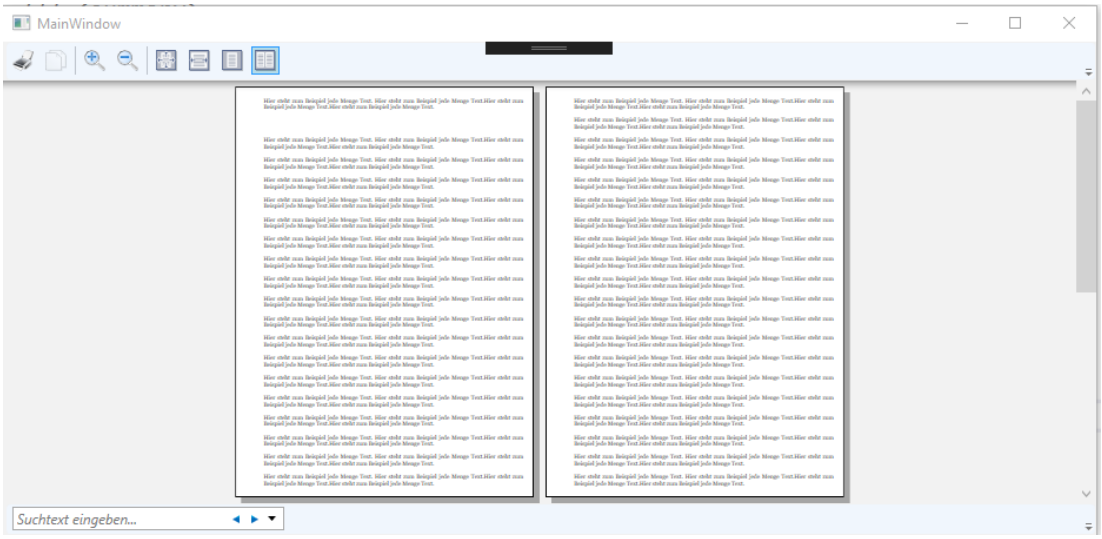
Damit haben Sie auch eine recht einfache Lösung für die Ausgabe von Flow-Dokumenten.

11 Weitere WPF-Controls

Hier finden Sie Beschreibungen von zwei weiteren WPF-Controls die es nicht ins Buch geschafft haben.

11.1 DocumentViewer

Neben den auf die Flow-Dokumente festgelegten Controls findet sich auch ein *DocumentViewer*-Control, das ausschließlich zur Anzeige von XPS-Dokumenten verwendet werden kann. Neben einer Druckoption können Sie die Daten auch in die Zwischenablage kopieren, die Ansicht skalieren und zwischen unterschiedlichen Seitendarstellungen wechseln. Last, but not least, verfügt das Control auch über ein einfache Suchfunktion innerhalb des Textes.



Wie Sie ein externes XPS-Dokument laden, zeigt das folgende Beispiel:

Beispiel 11.1: Laden eines XPS-Dokuments in den *DocumentViewer*

XAML

```
<Window x:Class="_11._1_DocumentViewer.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        Title="MainWindow" Height="238" Width="348" Loaded="Window_Loaded"
        WindowStartupLocation="CenterScreen">
        <DocumentViewer Name="documentViewer1" />
</Window>
```

C#

Fügen Sie zunächst die Assembly *ReachFramework.dll* als Verweis zu Ihrem Projekt hinzu. Nachfolgend können Sie den entsprechenden Ereigniscode übernehmen.

Namespaces importieren:

```
...
using System.Windows.Xps.Packaging;
using System.IO;

namespace _11._1_DocumentViewer
{
    public partial class MainWindow : Window
    {
...
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
```

Zunächst ein *XpsDocument* aus der Datei „Test.xps“ erstellen, nachfolgend können wir eine *FixedDocumentSequence* für die Anzeige abrufen:

```
            XpsDocument doc = new XpsDocument(«test.xps»,
                                  FileAccess.Read);
            documentViewer1.Document = doc.GetFixedDocumentSequence();
        }
```

HINWEIS: Mehr zu diesem Thema finden Sie in Abschnitt 10.1.1, wo wir uns sowohl dem Erstellen von XPS-Dokumenten als auch der Verwendung der *DocumentViewer*-Komponente zuwenden werden.

11.2 InkCanvas

Mit dem *InkCanvas* möchten wir Ihnen noch ein zunächst recht unscheinbares Control vorstellen, das im Zuge der größer werdenden Verbreitung von Tablet-PCs sicher noch an Bedeutung gewinnen wird. Das Control stellt eine Zeichenfläche zur Verfügung, in der Sie mit einem Stift (oder auch der Maus) freie Zeichnungen realisieren oder Markierungen vornehmen können. Dazu kann optional in den Hintergrund des Controls eine Grafik eingeblendet werden (z. B. ein Wegeplan).

Die von Maus, Stift oder Code erzeugten Linien/Striche werden in einer internen *Strokes*-Collection verwaltet. Einzelne *Stroke*-Objekte bestehen wiederum aus Zeichenpunkten (*StylusPoints*), haben eine Stiftform (*StylusTip*), eine Größe (*Width, Height*) und weisen eine Farbe auf.

Die Größe des Controls richtet sich beim Entwurf zunächst nach dem umgebenden Layout-Control. Handelt es sich bei diesem zum Beispiel um einen *ScrollViewer,* wird die Größe des *InkCanvas* zur Laufzeit an die maximalen Stiftpositionen angepasst. Das heißt, zeichnen Sie über den "Rand" des Controls hinweg, werden die Stiftbewegungen zunächst aufgezeichnet. Erst nach dem Zeichenende werden die Abmessungen des Controls an die nun neuen maximalen Ausdehnungen angepasst.

11.2.1 Stift-Parameter definieren

Wie schon erwähnt, kann der Zeichenstift verschiedene Eigenschaften aufweisen, die zusammen mit den Eingabekoordinaten in der *Strokes*-Collection abgespeichert werden.

Einstellen können Sie diese Werte über die *DefaultDrawingAttributes*-Eigenschaft des *InkCanvas*:

Eigenschaft	Beschreibung
Color	Die für den Stift verwendete Farbe.
FitToCurve	Sollen die Linien geglättet werden[1], setzen Sie den Wert auf *True*. Nach der Zeichenoperation wird in diesem Fall die Linie automatisch an einen Kurvenverlauf angepasst.
Height, Width	Bestimmt die Höhe und Breite des Zeichenstifts.
StylusTip	Bestimmt die Grundform des Zeichenstifts (Rectangle, Ellipse).

[1] z. B. um Zitterbewegungen und Ruckler auszugleichen …

Beispiel 11.2: Linienverlauf ohne und mit *FitToCurve*

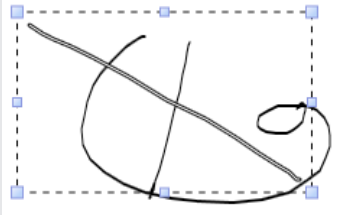
Beispiel 11.3: Linie mit rundem und eckigem Stift

11.2.2 Die Zeichenmodi

Der *InkCanvas* kennt verschiedene Zeichenmodi (*EditingMode*), über die Sie die jeweils ausführbare Aktion vorgeben:

EditingMode	Beschreibung
None	Keine Reaktion auf Stifteingaben.
Ink	Normales Zeichnen.
GestureOnly	Während der Stiftbewegung ist die Figur sichtbar, diese wird nach dem Zeichenende gelöscht. Dieser Modus dient der Auswertung der Gestik (z. B. ein gezeichneter Kreis, eine Linie etc.).
InkAndGesture	Zeichnen und Gestikauswertung.
Select	Auswahl von *Stroke*-Objekten, diese können zum Beispiel verschoben oder gelöscht werden.
EraseByPoint	Löschen von Linienstücken (ähnlich Radiergummi in einem Malprogramm).
EraseByStroke	Löschen kompletter *Stroke*-Objekte (ähnlich einem Vektorgrafikprogramm).

Beispiel 11.4: Markierte (Select) *Stroke*-Objekte



11.2.3 Inhalte laden und sichern

Möchten Sie die Zeichnung sichern bzw. zu einem späteren Zeitpunkt erneut laden, müssen Sie sich mit der *Strokes*-Collection beschäftigen. Diese verfügt über die erforderliche *Save*-Methode bzw. einen geeigneten Konstruktor, um die Daten aus einem Stream einzulesen.

Beispiel 11.5: Sichern der Daten

C#

```
using System.IO;
...

        private void Button_Click(object sender, RoutedEventArgs e)
        {
            using (FileStream fs = new FileStream("test.ink",
                   FileMode.Create, FileAccess.Write))
            {
                inkCanvas1.Strokes.Save(fs);
            }
        }
```

Beispiel 11.6: Laden der Daten

C#

```
Using System.IO;
using System.Windows.Ink;
...
        private void Button_Click_1(object sender, RoutedEventArgs e)
        {
            inkCanvas1.Strokes.Clear();
            using (FileStream fs = new FileStream("test.ink",
                   FileMode.Open, FileAccess.Read))
            {
                inkCanvas1.Strokes = new StrokeCollection(fs);
            }
        }
```

11.2.4 Konvertieren in eine Bitmap

Außer im *InkCanvas* können Sie mit den gesicherten Daten nichts anfangen. Da stellt sich schnell die Frage, wie Sie Ihre Kunstwerke in einem lesbaren Format sichern können. Am Beispiel des BMP-Formats wollen wir Ihnen die Vorgehensweise aufzeigen.

Beispiel 11.7: Sichern der aktuellen Zeichnung im BMP-Format

C#

```
        private void Button_Click_2(object sender, RoutedEventArgs e)
        {
            using (FileStream fs = new FileStream("test.bmp",
```

```
                FileMode.Create, FileAccess.Write))
            {
```

Zielbitmap entsprechend der Größe der Zeichenfläche erzeugen:

```
                RenderTargetBitmap rtb =
                   new RenderTargetBitmap((int)inkCanvas1.ActualWidth,
                   (int)inkCanvas1.ActualHeight, 0, 0,
                   PixelFormats.Default);
```

Daten ausgeben:

```
                rtb.Render(inkCanvas1);
```

Mit dem BMP-Encoder kodieren:

```
                BmpBitmapEncoder encoder = new BmpBitmapEncoder();
                encoder.Frames.Add(BitmapFrame.Create(rtb));
```

Speichern:

```
                encoder.Save(fs);
            }
        }
```

11.2.5 Weitere Eigenschaften

Mit Hilfe der Eigenschaft *IsHighlighter* schalten Sie den Stift in einen teiltransparenten Modus, sodass darunter liegende Zeichnungen sichtbar bleiben.

Verwenden Sie Zeichentablets (z. B. von Watcom), hat der Anpressdruck des Stift einen Einfluss auf die Linienbreite. Setzen Sie *IgnorePressure* auf *True,* um dies zu verhindern.

Nicht in jedem Fall ist die aktuelle Zeichnung komplett zu sehen, bzw. Einzelheiten sind so klein, dass sie nicht sichtbar sind. Hier hilft die Verwendung einer Layout-Transformation weiter (Sie erinnern sich, dass es sich bei allen WPF-Controls um Vektorgrafiken handelt).

Beispiel 11.8: Skalieren des *InkCanvas* mit Hilfe eines *Slider*-Controls

XAML

```
    <InkCanvas Name="inkCanvas1">
      <InkCanvas.LayoutTransform>
        <ScaleTransform ScaleX="{Binding ElementName=slider1,Path=Value}"
                        ScaleY="{Binding ElementName=slider1,Path=Value}" />
      </InkCanvas.LayoutTransform>
    </InkCanvas>
...
    <Slider Height="26" Name="slider1" Maximum="5" Minimum=".1" Value="1"
Width="120" />
```

Ergebnis

Die Skalierung in Aktion:

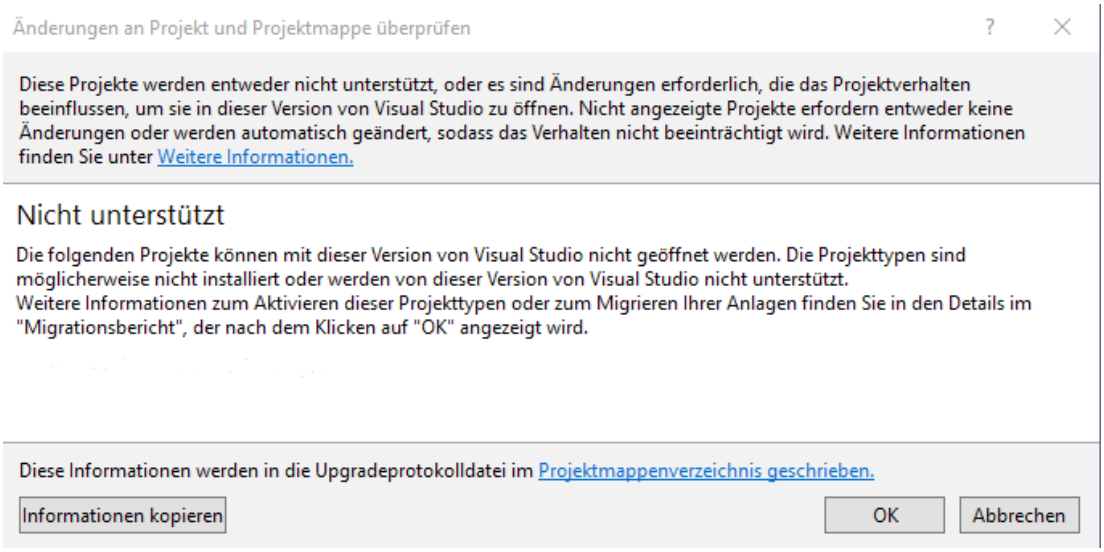
12 Verteilen von Anwendungen

Dem .NET-Programmierer bieten sich in Visual Studio zwei Hauptvarianten an, mit denen er seine Anwendung an den Endkunden[1] weitergeben kann:

- ClickOnce-Deployment
- Setup-Projekte

Während es sich beim ClickOnce-Deployment um eine Technologie handelt, bei der die Installationen über das Web abgerufen/aktualisiert werden können, ist die Verwendung von Setup-Projekten sicher jedem Windows-Nutzer bekannt.

Wer seine Altprojekte mit einem enthaltenen Setup-Projekt in Visual Studio öffnet, wird eine zunächst böse Überraschung erleben:



Die guten alten Microsoft-Setup-Projekte sind nicht mehr Teil der Standardinstallation, können jedoch nachinstalliert werden. Mehr dazu in Abschnitt 12.2, Setup-Projekte.

[1] Sehen wir einmal von den Distributionsmöglichkeiten für Webanwendungen ab.

An dieser Stelle möchte ich noch den Hinweis auf einen anderen Setup-Typen geben. Gerade bei sehr großen Projekten wird sehr häufig das sogenannte Wix-Toolset (Windows Installer und XML) eingesetzt. Mehr Informationen erhalten Sie unter *http://wixtoolset.org/*.

12.1 ClickOnce-Deployment

Mit *ClickOnce* bietet sich dem .NET-Entwickler eine interessante Möglichkeit, seine Programme zu verteilen. Das Grundprinzip dieses Verfahrens ist es, das Verteilen und Aktualisieren von Anwendungen zu zentralisieren und damit auch zu vereinfachen.

12.1.1 Übersicht/Einschränkungen

Im Gegensatz zu den lokal installierten Windows-Anwendungen werden bei ClickOnce die Applikationen von einem zentralen Ort aus zur Verfügung gestellt. Dies darf eine Website, ein FTP-Server oder auch ein Dateipfad sein. Installiert bzw. ausgeführt werden kann die Anwendung von einer CD/DVD, einer Website oder einem UNC-Pfad.

HINWEIS: Sie entscheiden bereits beim Erstellen des Projekts, ob die ClickOnce-Anwendung nur online oder auch offline verfügbar ist. Im letzteren Fall wird dem Startmenü des Anwenders ein entsprechender Eintrag hinzugefügt.

Beim Herunterladen der Anwendungsdateien werden diese jedoch nicht wie gewohnt unterhalb des Verzeichnisses *\Programme* abgelegt, sondern im Benutzerprofil des aktuellen Nutzers (*\\Dokumente und Einstellungen\<Benutzer>\Lokale Einstellungen\Apps*).

HINWEIS: Aus obiger Tatsache resultieren auch einige Einschränkungen der ClickOnce-Distribution. So steht keine Möglichkeit zur Pfadauswahl zur Verfügung, auch die Installation selbst muss auf administrative Rechte/Aufgaben (Dateisystem, Registry) verzichten.

Ist die Anwendung auch offline verfügbar, bietet das ClickOnce-Verfahren die Möglichkeit, die Anwendungsdateien automatisch zu aktualisieren, ohne dass sich der Nutzer explizit darum kümmern muss.

Gesteuert wird das Verhalten der ClickOnce-Anwendung mit zwei Manifestdateien:

- Das Anwendungsmanifest beschreibt die für die Applikation erforderlichen Assemblies und weitere Dateien.
- Über das Bereitstellungsmanifest werden Versionsinformationen, Speicherorte und der Pfad des Anwendungsmanifests beschrieben.

HINWEIS: Die beiden Manifestdateien werden im Hintergrund vom Visual-Studio-Assistenten, basierend auf Ihren Vorgaben, erstellt. Sie brauchen sich also nicht selbst darum zu kümmern.

Doch grau ist alle Theorie und so wollen wir Ihnen auch diesmal mit einem einfachen Beispiel die Schritte zum fertigen ClickOnce-Projekt aufzeigen.

12.1.2 Die Vorgehensweise

Visual Studio bietet zumindest zwei Wege, um das *ClickOnce*-Deployment vorzubereiten:

- Manuelles Einstellen der Deployment-Optionen, die Teil der Projekteigenschaften sind, auf den Registerkarten *Sicherheit* und *Veröffentlichen* und anschließendes Betätigen der Schaltfläche *Jetzt veröffentlichen…* auf der Registerkarte *Veröffentlichen*.
- Verwenden des Webpublishing-Assistenten durch Betätigen der Schaltfläche *Webpublishing-Assistent...* auf der Registerkarte *Veröffentlichen* oder Aufruf des Kontextmenüs *Veröffentlichen* im Projektmappen-Explorer.

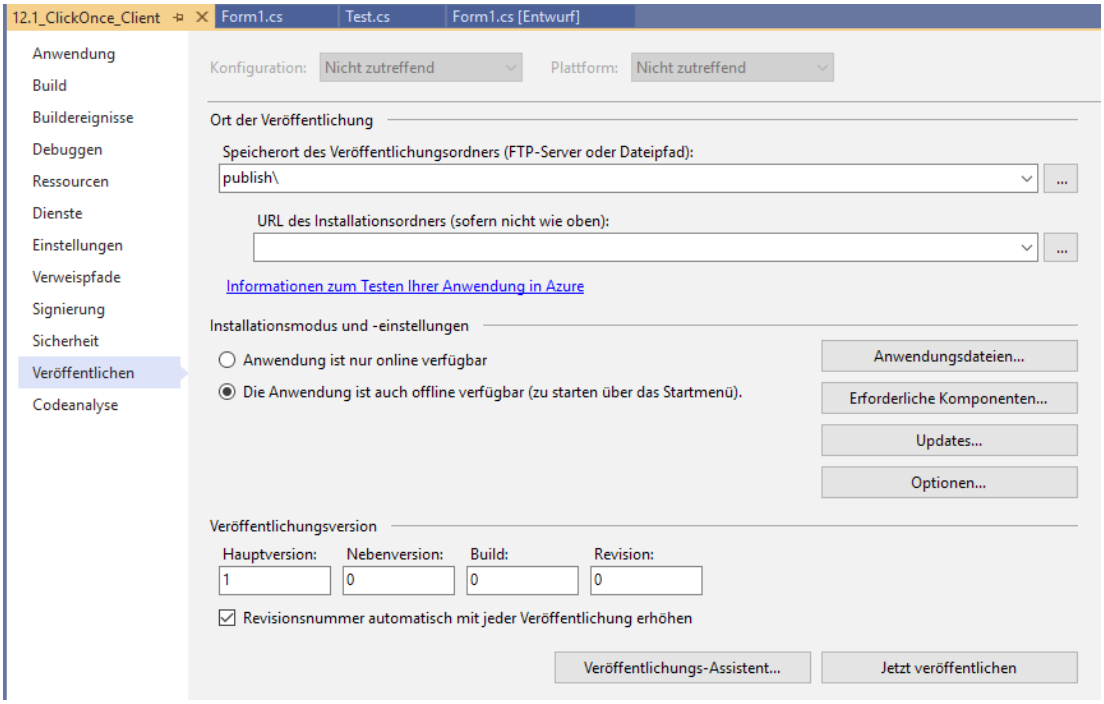
Im Folgenden wollen wir die Vorgehensweise am Beispiel einer aus zwei Komponenten bzw. Projekten (Client und Server) bestehenden Anwendung demonstrieren.

Da Sie zum *ClickOnce*-Deployment keinerlei zusätzlichen Code schreiben müssen, steht es Ihnen frei, stattdessen auch ein beliebiges anderes Projekt aus Ihrem Fundus zu verwenden. Allerdings sollte es nicht zu komplex sein, da während der Installation keinerlei Administratorenrechte zur Verfügung stehen und z. B. Einträge in die Registry nicht vorgenommen werden können.

Wir wollen zunächst den ersten der beiden oben genannten Wege beschreiten, also auf den direkten Einsatz des Webpublishing-Assistenten verzichten.

12.1.3 Ort der Veröffentlichung

Rufen Sie das Menü *Projekt/ClickOnce_Client-Eigenschaften...* des Projekts *ClickOnce_Client* auf und wählen Sie die Registerseite *Veröffentlichen*:

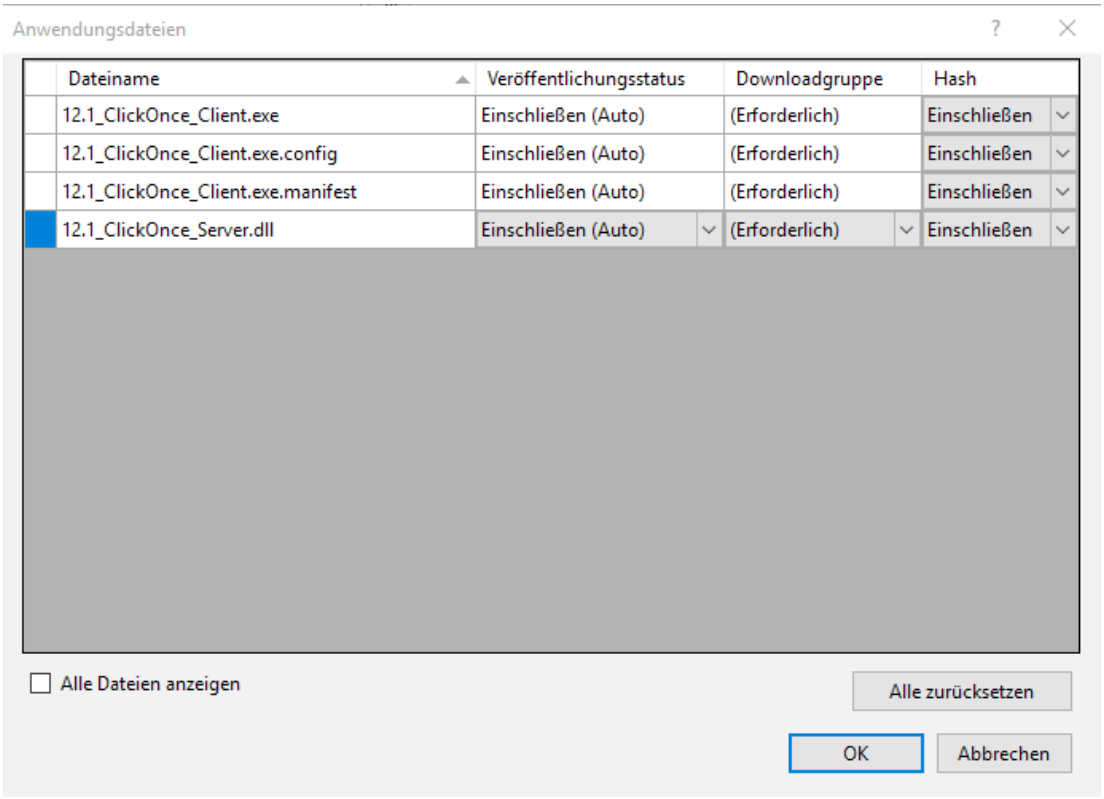


Tragen Sie dort den Ort der Veröffentlichung ein, für einen ersten Test eignet sich das Filesystem, alternativ können Sie auch einen FTP-Server angeben.

HINWEIS: Die Registerseiten *Sicherheit* und *Veröffentlichen* werden Sie beim *ClickOnce_Server*-Projekt vergeblich suchen, da es sich hier um keine für das Deployment geeignete (**.exe*-)Anwendung handelt.

12.1.4 Anwendungsdateien

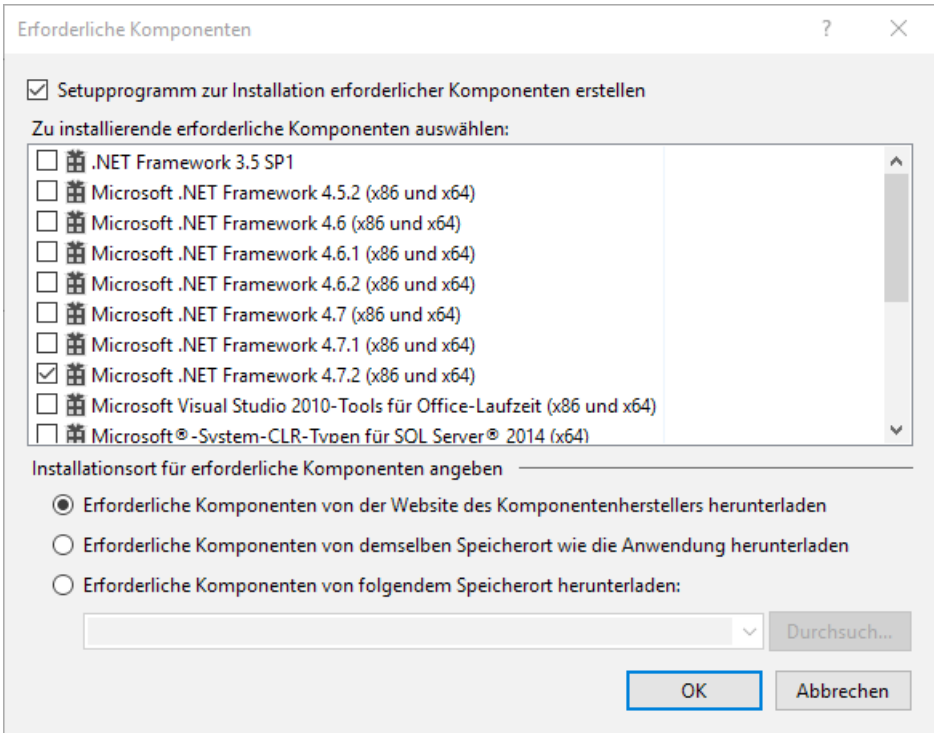
Klicken Sie auf die Schaltfläche *Anwendungsdateien...*, so öffnet sich ein Dialog, in dem Sie angeben können, welche Dateien auf den Deployment-Server zu kopieren sind. *ClickOnce* erkennt selbstständig, welche Abhängigkeiten zwischen den Dateien bestehen und welche Dateien unbedingt erforderlich sind.



12.1.5 Erforderliche Komponenten

Nach Klick auf die Schaltfläche *Erforderliche Komponenten...* zeigt Ihnen ein Dialog die Komponenten an, die bei Bedarf ebenfalls mit auf den Deployment-Server kopiert werden können.

Neben den unterschiedlichen Versionen des *.NET Framework* stehen Ihnen auch Versionen der *SQL Server Express/LocalDB*und einige andere zur Verfügung. Für unser einfaches Beispiel können wir aber auf zusätzliche Komponenten verzichten.



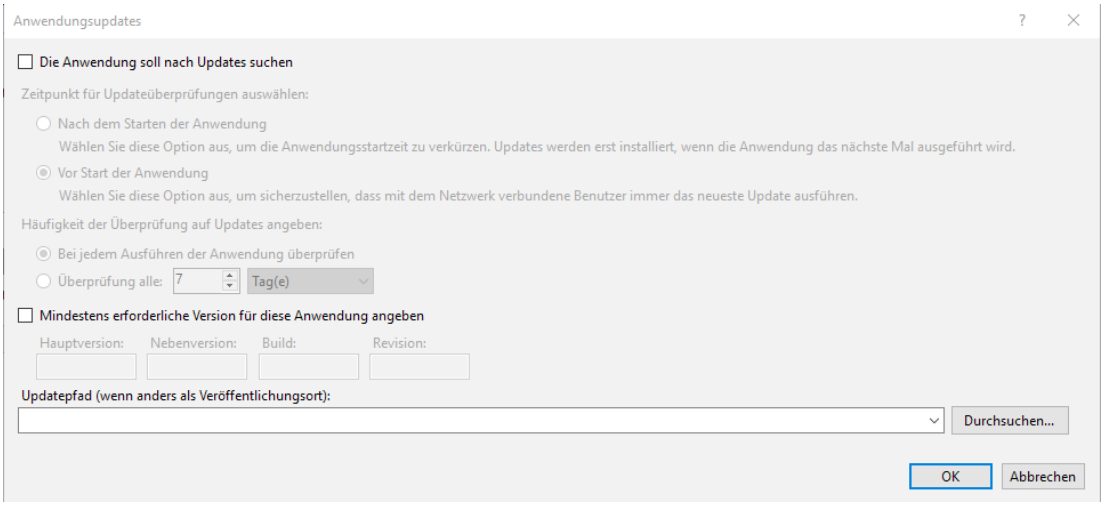
12.1.6 Aktualisierungen

Ein entscheidendes Merkmal der *ClickOnce*-Technologie ist das Update-Management, welches auf Wunsch auch vollautomatisch ablaufen kann.

Nach dem Klick auf die *Updates...*-Schaltfläche öffnet sich der entsprechende Dialog (siehe folgende Abbildung), der – wie übrigens alle anderen auch – so gut beschriftet ist, dass zusätzliche Erklärungen an dieser Stelle fast schon Papierverschwendung wären.

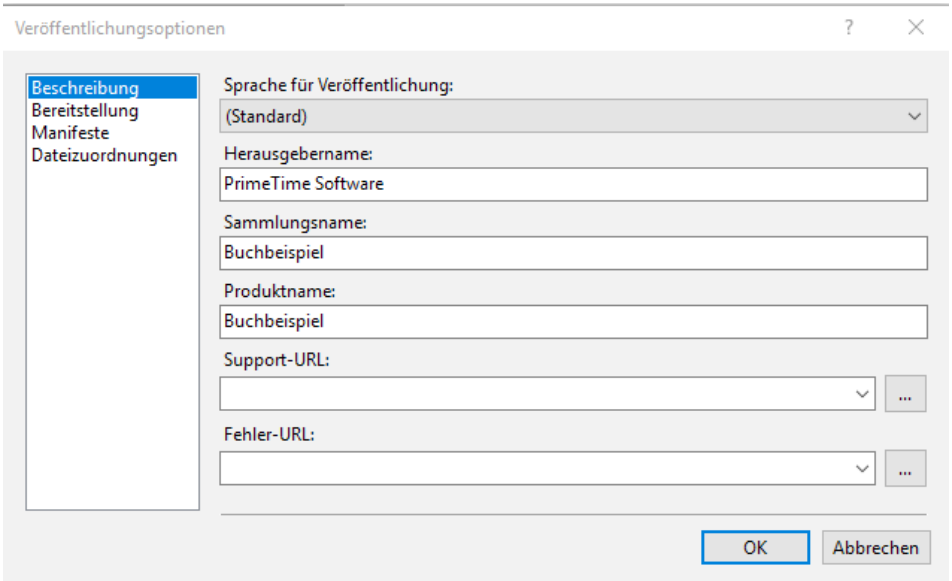
Sie entscheiden, ob (und wenn ja wann) auf Updates geprüft werden soll (vor oder nach dem Start der Anwendung). Bedenken Sie, dass Internetverbindungen nicht immer rasend schnell sein müssen, meist reicht es aus, nach dem Start zu prüfen und beim nächsten Programmstart steht die neue Version dann zur Verfügung.

Weiterhin können Sie das Update-Intervall, d. h. die Häufigkeit der Update-Suche, bestimmen.



12.1.7 Veröffentlichungsoptionen

Legen Sie hier die wesentlichen Daten zu Ihrem Setup-Projekt fest (Herausgeber, Bereitstellungsseite, Dateizuordnungen etc.):



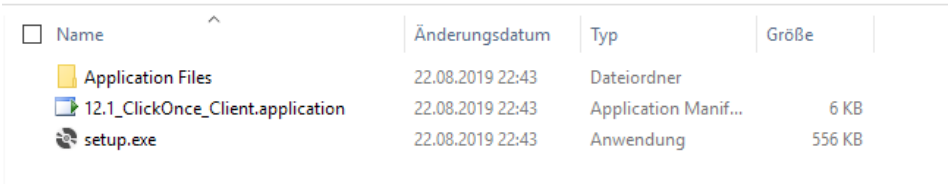
12.1.8 Veröffentlichen

Haben Sie alle Einstellungen vorgenommen, so steht dem entscheidenden Klick auf die Schaltfläche „Jetzt veröffentlichen“ nichts mehr im Wege.

Es vergeht etwas Zeit bis im Explorer ein neuer Ordner sichtbar wird, in dem die Programmdateien liegen.

12.1.9 Verzeichnisstruktur

Im Filesystem wird folgend dargestelltes Verzeichnis angelegt:

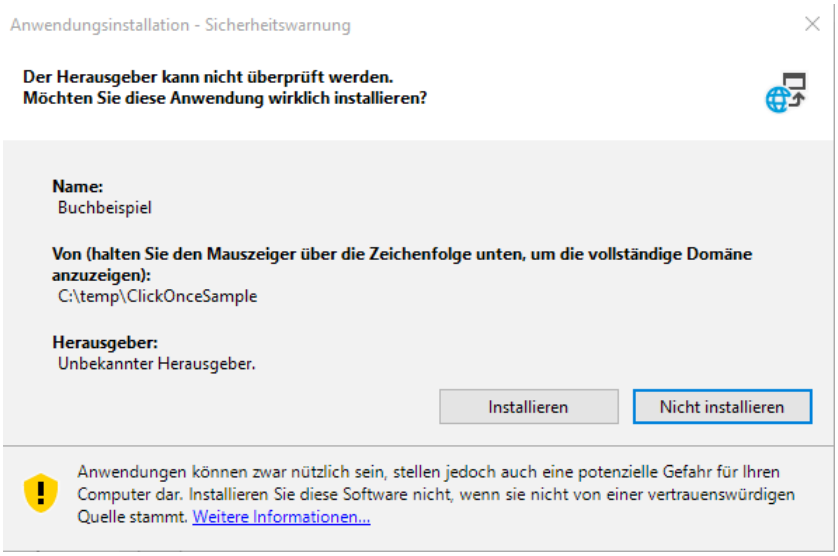


Ein paar Worte zu den einzelnen Dateien:

- Die Datei *setup.exe* startet die Installation
- *12.1_ClickOnce_Client.application* ist eine XML-Datei, sie enthält das Deploymentmanifest, welches Setup und Update der Anwendung konfiguriert.
- Im Unterverzeichnis *Application Files*\12.1_ClickOnce_*Client_1_0_0_0* befindet sich das eigentliche Programm. Es besteht aus der *exe*-Datei *12.1_ClickOnce_Client.exe.deploy*, der Klassenbibliothek *12.1_ClickOnce_Server.dll.deploy* und dem Application-Manifest *12.1_ClickOnce_Client.exe.manifest*.

Es folgt ein Hinweis zu Lizenzbedingungen des .NET Frameworks, den Sie bedenkenlos akzeptieren können:

Danach kommt der Dialog zur Installation den Sie mit Installieren bestätigen:

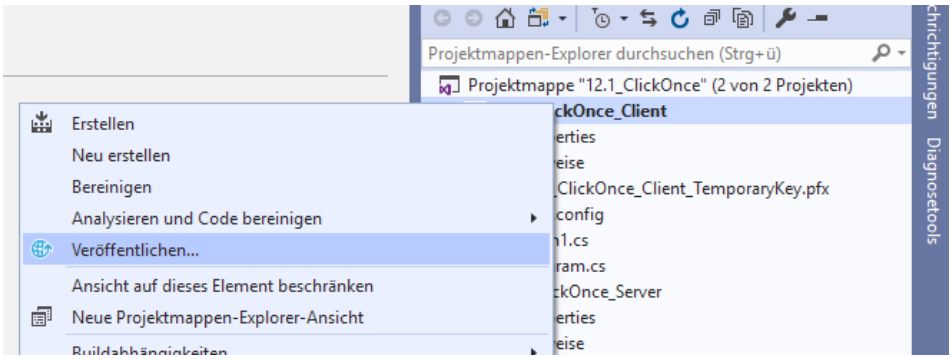


Nach dem Klick auf die *Installieren*-Schaltfläche und dem erfolgreichen Abschluss der Installation wird das Programm gestartet. Außerdem können Sie es – wie jedes andere Programm auch – über die *Start*-Schaltfläche des Windows Desktop aufrufen, selbst wenn keine Verbindung zum Netzwerk mehr besteht.

12.1.10 Der Webpublishing-Assistent

Dieser Assistent soll vor allem dem Einsteiger die Arbeit erleichtern, da dieser die wichtigsten Deployment-Optionen nicht mehr unbedingt auf den Registerseiten *Sicherheit* und *Veröffentlichen* des Projekteigenschaften-Dialogs einstellen muss. Stattdessen erfolgt ein Frage-Antwort-Spiel.

Am einfachsten starten Sie den Assistenten über das Kontextmenü des *ClickOnce_Client*-Projekts:



Nach Beantwortung einfacher Fragen („Wird die Anwendung offline verfügbar sein?“, „Wo möchten Sie die Anwendung veröffentlichen?“) landet man auch hier an der Stelle wie oben bereits beschrieben.

12.1.11 Neue Versionen erstellen

Um sich in *ClickOnce* so richtig fit zu machen, sollten Sie bereits jetzt eine neue Version der Anwendung erstellen, zum Beispiel mit einer neuen Methode im *ClickOnce_Server*-Projekt oder einfach nur mit einer veränderten Beschriftung im *ClickOnce_Client*. Nach erneutem Aufruf des Webpublishing-Assistenten werden Sie im Explorerein neues Verzeichnis *12.1_ClickOnce_Client_1_0_0_1* entdecken, in welchem die neue Programmversion abgelegt ist. Das Deployment-Manifest *12.1_ClickOnce_Client.application* wurde so geändert, dass es jetzt auf das Application-Manifest der neuen Version verweist.

In Abhängigkeit von den vorgenommenen Einstellungen wird die Anwendung beim Aufruf sofort automatisch oder erst auf Nachfrage aktualisiert. Läuft etwas schief, so können Sie problemlos die Vorgängerversion zurückholen.

12.2 Setup-Projekte

Nachdem wir uns im vorhergehenden Abschnitt intensiv mit dem ClickOnce-Deployment befasst haben, wollen wir uns nun mit dem von älteren Visual-Studio-Versionen bekannten Setup-Projekt beschäftigen. Da das Setup-Projekt seit der Version 2012 nicht mehr standardmäßig in Visual Studio integriert ist, müssen wir es zuerst als Add-on nachinstallieren.

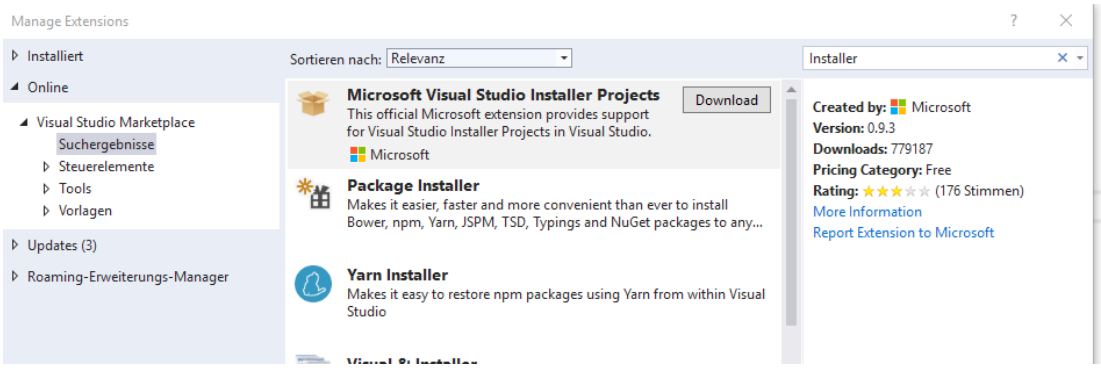
Mithilfe dieser zusätzlich installierten Projektvorlage können Sie nun ein professionelles Setup-Programm erstellen.

Alle zu einem Setup-Projekt gehörenden Dateien sowie die Anweisungen zur Steuerung des Setups werden vom Setup-Assistenten in eine *.msi*-Datei gepackt (das gilt auch im wörtlichen Sinne). Diese wiederum wird, zusammen mit dem Installationsprogramm, in eine *Setup.exe* verpackt. Spätestens diese Datei dürfte Ihnen als Windows-Anwender bekannt vorkommen.

Doch bevor es so weit ist, müssen wir uns zunächst um die Installation der Setup-Projektvorlage kümmern, denn dies ist im Zusammenhang mit der Installation von Visual Studio noch nicht erfolgt.

12.2.1 Installation

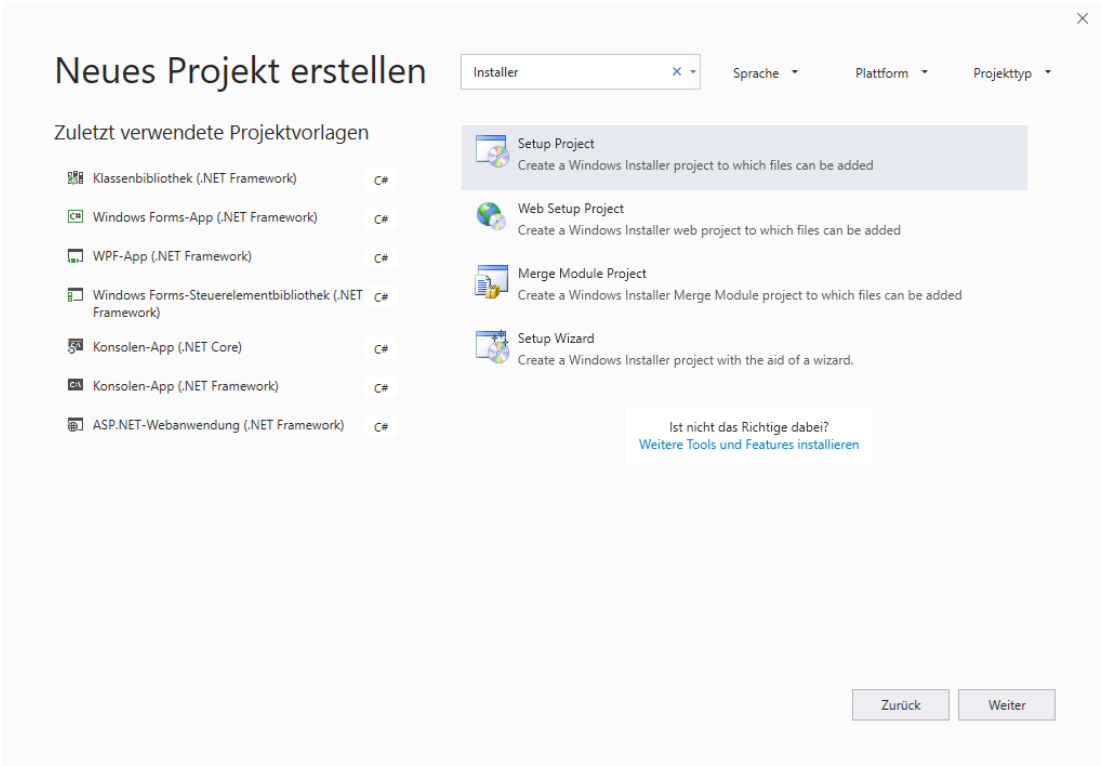
Öffnen Sie dazu den Menübefehl *Erweiterungen/Erweiterungen Verwalten*, wählen Sie dort den Punkt *Online* und suchen Sie im *Visual Studio Marketplace* nach *Installer*.



Laden Sie dann die Erweiterung *Microsoft Visual Studio Installer Projects* herunter, installieren Sie die Erweiterung und starten Sie Visual Studio neu.

Nach dem Neustart von Visual Studio finden Sie unter *Andere Projekttypen/Visual Sudio Installer* das neue *Setup Projekt*.

Wenn Sie jetzt eine ältere Projektmappe mit einem integrierten Setup-Projekt öffnen, wird dieses ab diesem Zeitpunkt auch wieder geladen.



12.2.2 Ein neues Setup-Projekt

Wenn Sie einen Installer für Ihr Projekt erstellen wollen, dann öffnen Sie zunächst das C#-Projekt, für das ein Setup erstellt werden soll. Wählen Sie danach den Menüpunkt *Datei/Hinzufügen/Projekt...* und wählen Sie unter *Andere Projekttypen/Visual Studio Installer* die Vorlage *Setup-Projekt* oder noch einfacher *Setup-Wizard* aus.

Wenn Sie den Setup-Wizard ausgewählt haben, werden Sie durch die Erstellung des Setup-Projekts geführt.

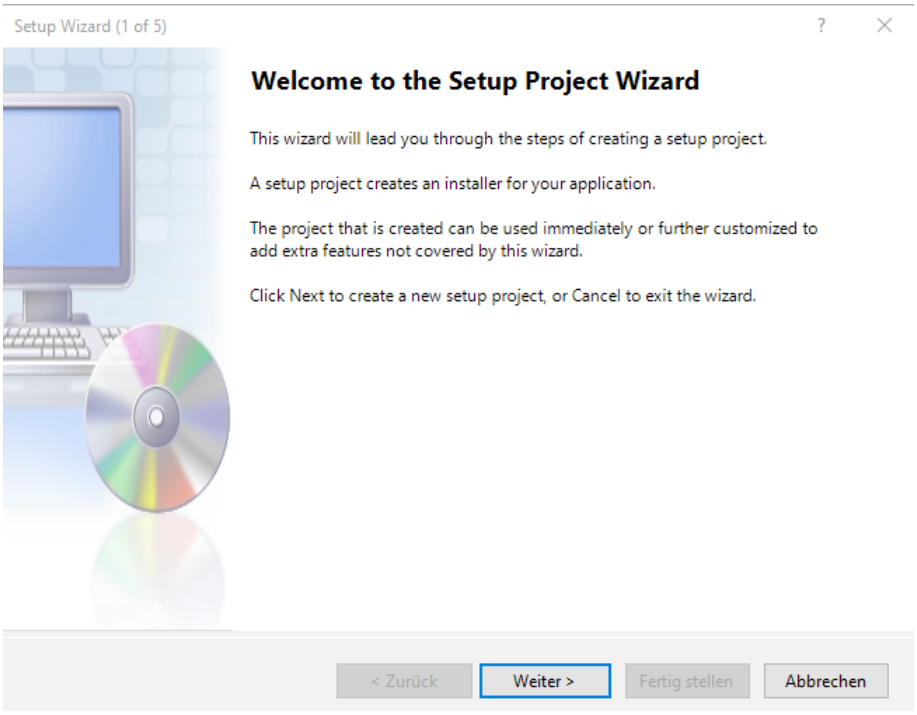
Bestätigen Sie zuerst den Willkommensdialog.

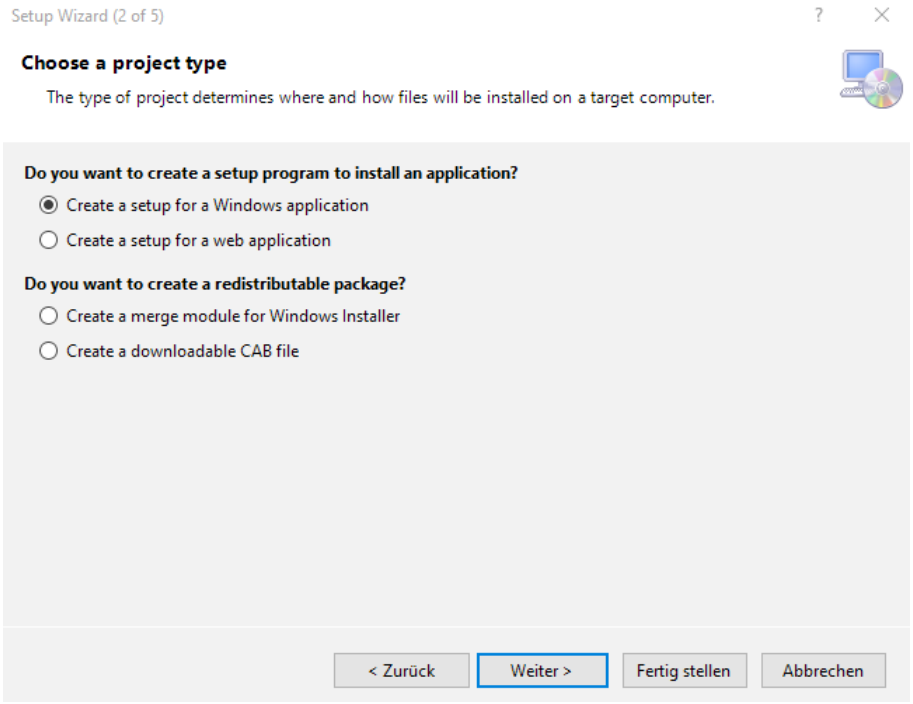
Im nächsten Dialog können Sie den Projekttyp auswählen. Wir wollen in diesem Fall ein Setup für eine Windows-Applikation erzeugen.

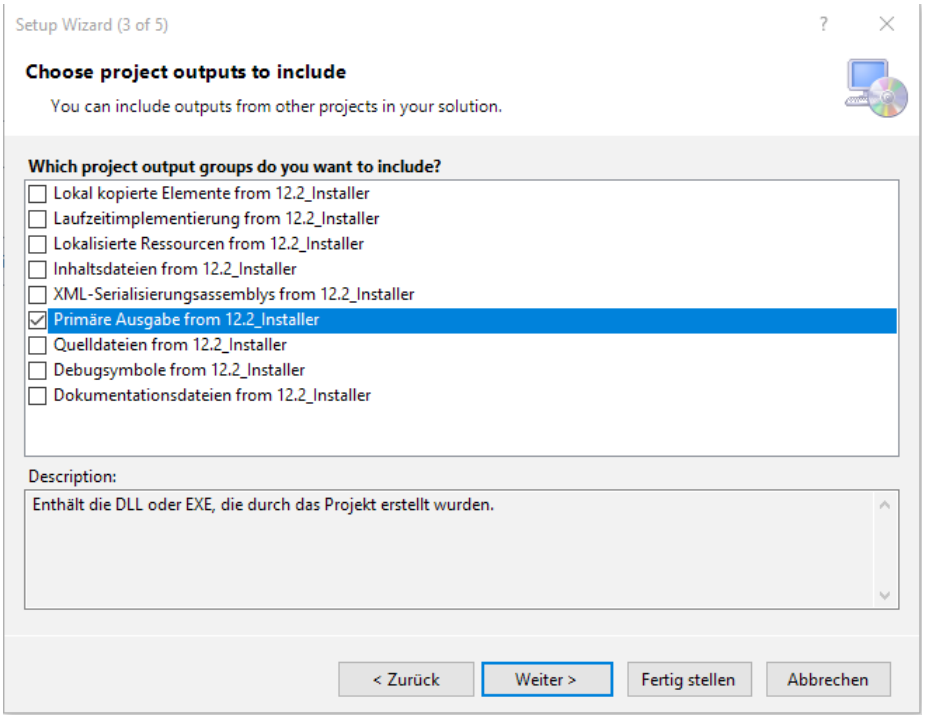
Danach wählen Sie noch aus, welche Ausgaben Sie von Ihrem Projekt dem Setup hinzufügen wollen. Im einfachsten Fall ist es die primäre Ausgabe Ihres Projekts. Optional können Sie natürlich noch lokalisierte Ressourcen oder andere Inhaltsdateien zu Ihrem Setup hinzufügen.

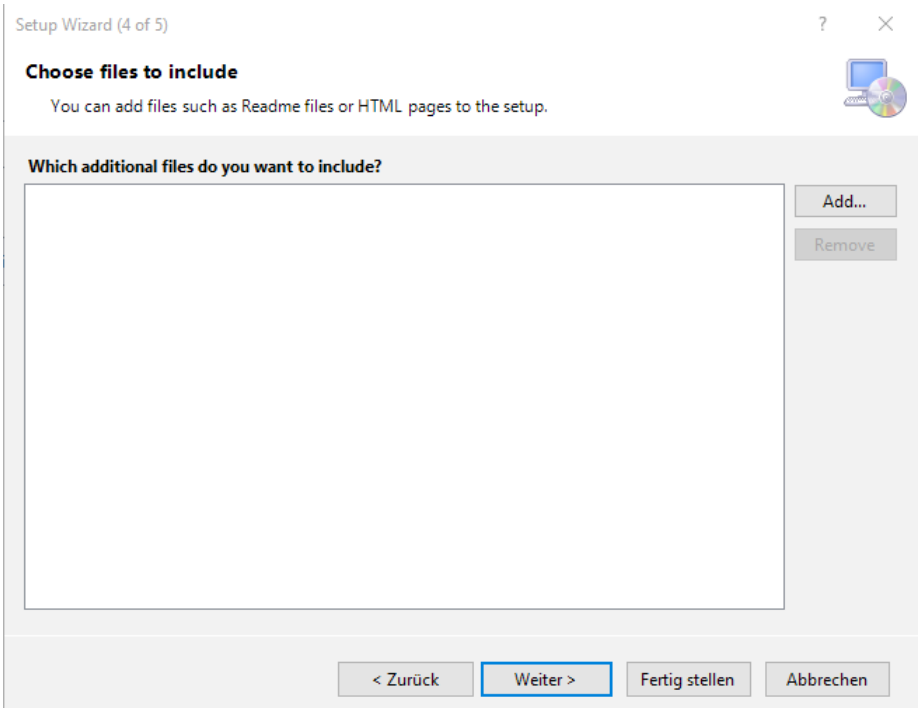
Im nächsten Schritt könnten Sie noch weitere Dateien (XML-Files, lokale Datenbanken oder sonstige Dateien) hinzufügen.

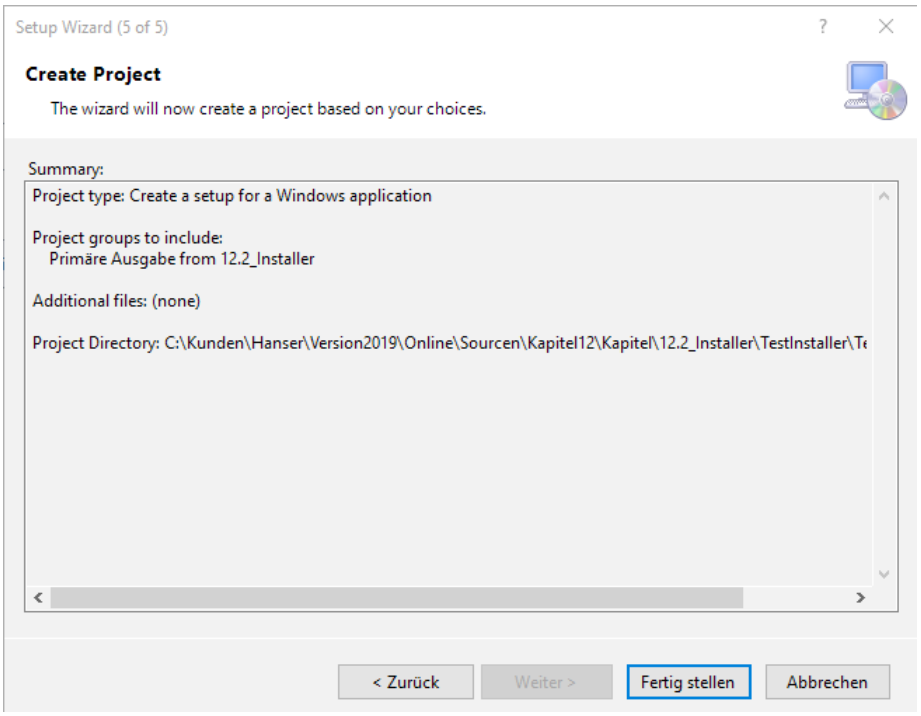
Und im letzten Dialog bestätigen Sie nochmals Ihre Eingaben und das Projekt wird nach der Betätigung der Schaltfläche *Fertig stellen* komplett für Sie angelegt.




Setup Wizard (2 of 5)
Choose a project type
The type of project determines where and how files will be installed on a target computer.
Do you want to create a setup program to install an application?
Create a setup for a Windows application
Create a setup for a web application
Do you want to create a redistributable package?
Create a merge module for Windows Installer
Create a downloadable CAB file
< Zurück
Weiter >
Fertig stellen
Abbrechen


Setup Wizard (3 of 5)
Choose project outputs to include
You can include outputs from other projects in your solution.
Which project output groups do you want to include?
Lokal kopierte Elemente from 12.2_Installer
Laufzeitimplementierung from 12.2_Installer
Lokalisierte Ressourcen from 12.2_Installer
Inhaltsdateien from 12.2_Installer
XML-Serialisierungsassemblys from 12.2_Installer
Primäre Ausgabe from 12.2_Installer
Quelldateien from 12.2_Installer
Debugsymbole from 12.2_Installer
Dokumentationsdateien from 12.2_Installer
Description:
Enthält die DLL oder EXE, die durch das Projekt erstellt wurden.
< Zurück
Weiter >
Fertig stellen
Abbrechen


Setup Wizard (4 of 5)
Choose files to include
You can add files such as Readme files or HTML pages to the setup.
Which additional files do you want to include?
Add...
Remove
< Zurück
Weiter >
Fertig stellen
Abbrechen

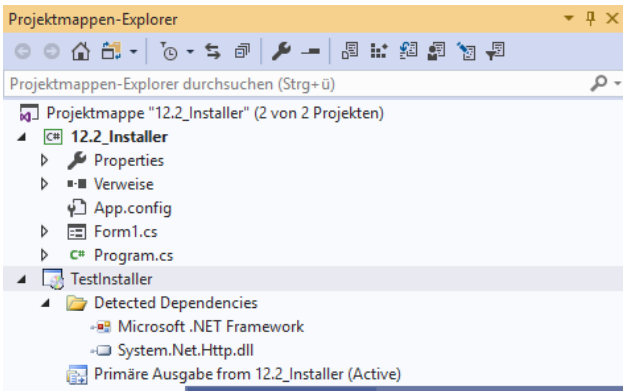

Setup Wizard (5 of 5)
Create Project
The wizard will now create a project based on your choices.
Summary:
Project type: Create a setup for a Windows application
Project groups to include:
Primäre Ausgabe from 12.2_Installer
Additional files: (none)
Project Directory: C:\Kunden\Hanser\Version2019\Online\Sourcen\Kapitel12\Kapitel\12.2_Installer\TestInstaller\Te
< Zurück
Weiter >
Fertig stellen
Abbrechen

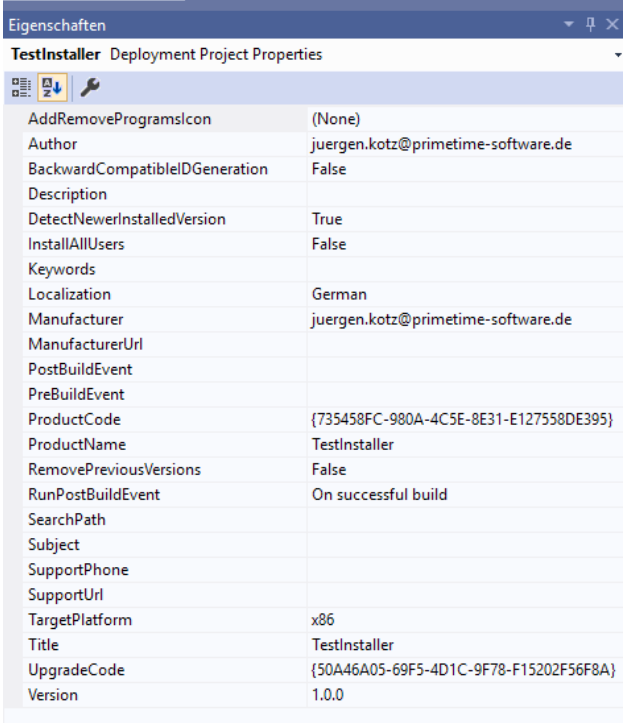
Doch betrachten wir, was überhaupt genau angelegt wurde.

Zum einen finden wir im Projektmappenexplorer ein neues Projekt (ich habe dem Projekt den Namen 12.2_Installer gegeben).

Unter den Projekteigenschaften können Sie verschiedene Einstellungen für Ihr Setup-Projekt einstellen, wie zum Beispiel Localization (in welcher Sprache das Setup bei der Installation ausgeführt wird) oder Versionsnummer des Setups.

Bei einer Änderung der Versionsnummer wird Ihnen empfohlen, auch den Product-Code zu ändern, das sollten Sie in der Regel immer tun, ansonsten bekommen Sie bei einer weiteren Ausführung eines aktualisierten Setups für ein bestehendes Projekt immer die Meldung, dass diese Version schon installiert ist.





Vielleicht sind Ihnen auch die zusätzlichen Symbole im Projektmappenexplorer aufgefallen.

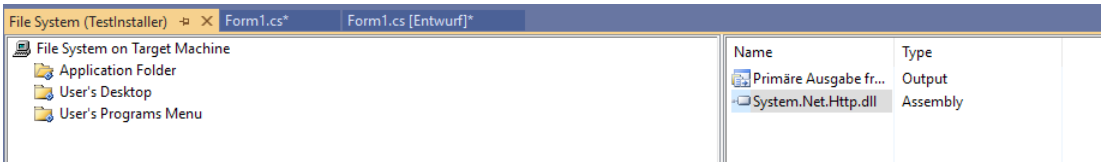
Damit können Sie durch die verschiedenen Dialoge des Setup-Projekts navigieren, um Anpassungen an Ihrem Setup durchzuführen.

Diese sechs Dialoge werden in den folgenden Abschnitten kurz beschrieben.

12.2.3 Dateisystem-Editor

In diesem Editor können Sie einstellen, welche Dateien wo im Dateisystem angelegt werden sollen.

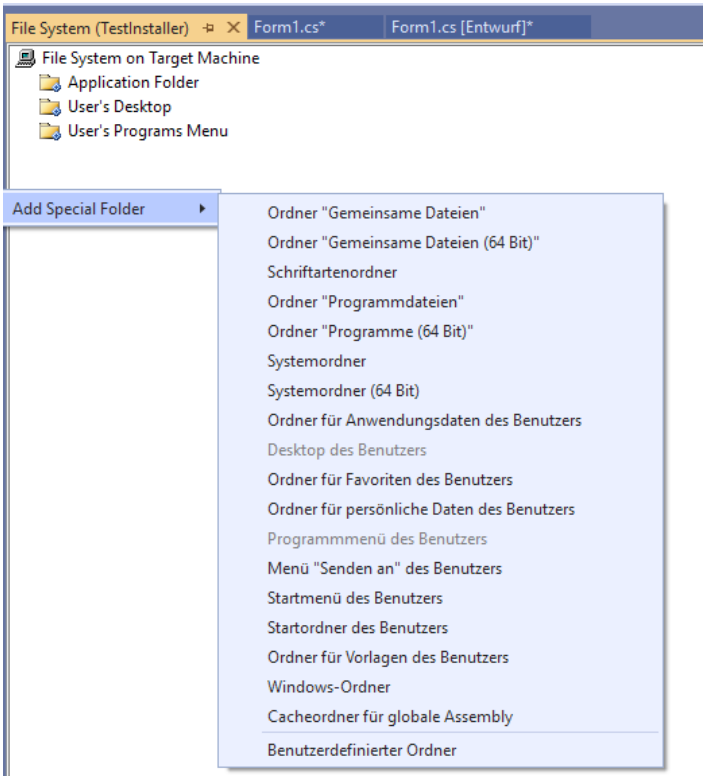
Im Applikationsordner werden alle Dateien aufgeführt, die in den entsprechenden Ordner gelegt werden.



Unter *Users Desktop* bzw. *Users Program Menu* können Sie Verknüpfungen zu Ihrer Anwendung definieren.

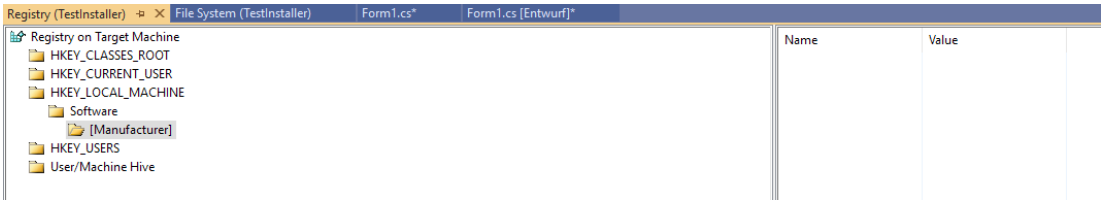
Für die Auswahl des *Icon* steht unter den Eigenschaften der Verknüpfung eine entsprechende Property zur Verfügung.

Wenn Sie zusätzliche Dateien in bestimmten Pfaden ablegen wollen (z. B. besondere Schriftarten etc.), dann können Sie über ein Kontextmenü weitere Dateiordner zu Ihrem Setup-Projekt hinzufügen.



12.2.4 Registrierungs-Editor

Im Registrierungs-Editor können Sie definieren, ob und in welchen Zweigen bei der Installation Registrierungsschlüssel in die Registry eingetragen werden.

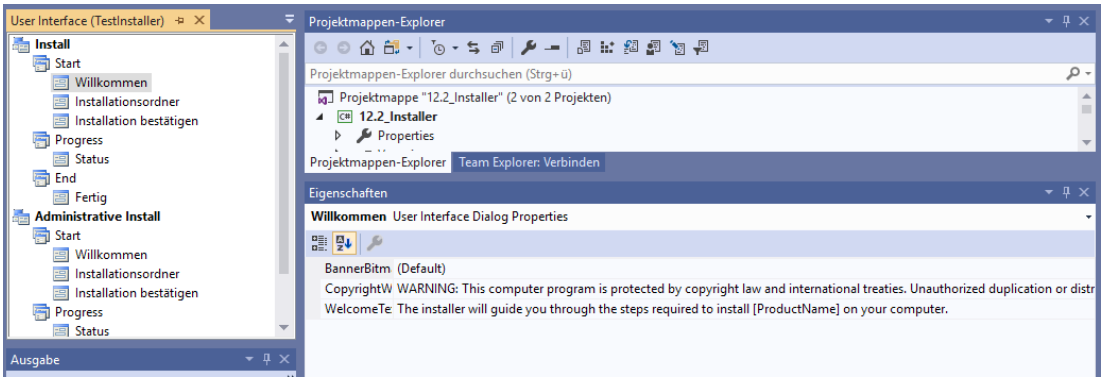


12.2.5 Dateityp-Editor

Im Dateityp-Editor können Sie für bestimmte Dateitypen(-endungen) Aktionen definieren. Zum Beispiel, welches Programm bei einem Doppelklick im Explorer gestartet werden soll oder ob es im Explorer ein Kontextmenü für Drucken geben soll etc.

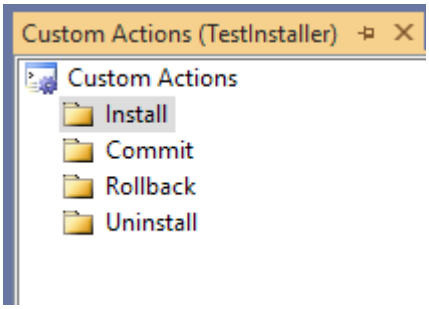
12.2.6 Benutzeroberflächen-Editor

Im Benutzeroberflächen-Editor können Sie verschiedenste Beschriftungen von Dialogen steuern, die im Laufe der Installation durchlaufen werden.

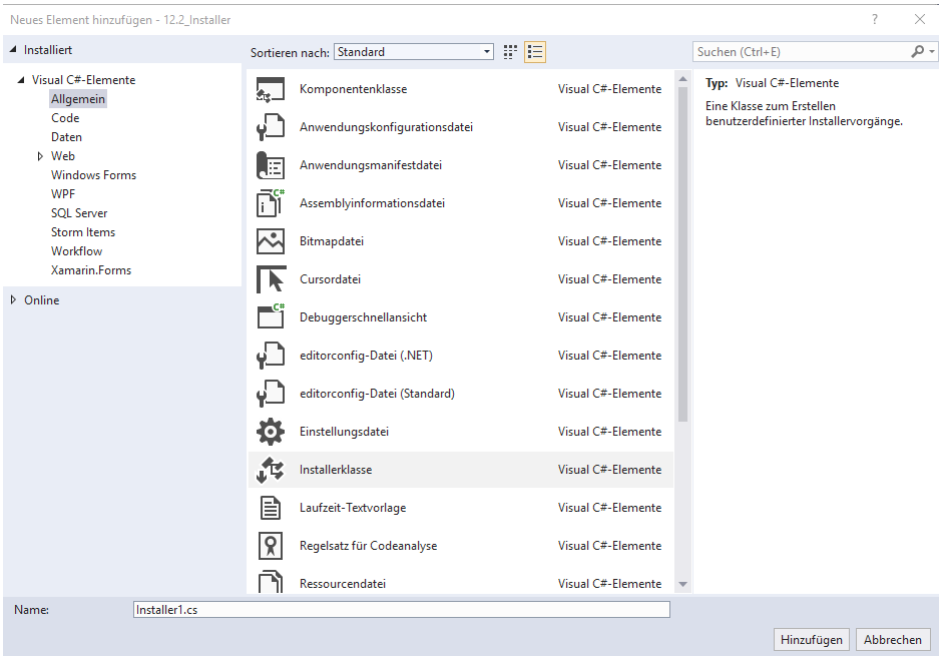


12.2.7 Editor für benutzerdefinierte Aktionen

Im Editor für benutzerdefinierte Aktionen können Sie definieren, dass bestimmte komplexere Aktionen bei der Installation, am Ende der Installation (*Commit*), beim Abbruch der Installation (*Rollback*) sowie beim Deinstallieren (*Uninstall*) ausgeführt werden.



Die komplexen Aktionen können Sie ganz einfach in Ihrem Projekt durch Hinzufügen einer Klasse vom Typ *Installerklasse ausführen*.



Diese Klasse besitzt ein Attribut *RunInstaller* und erbt von einer Basisklasse *Installer*. In dieser Klasse können Sie dann die entsprechenden Methoden für *Install*, *Uninstall*, *Commit* und *Rollback* überschreiben.

12.2.8 Editor für Startbedingungen

Im Editor für Startbedingungen können Sie bestimmte Voraussetzungen auf der Zielplattform überprüfen. Dies können bestimmte Dateien, Registrierungsschlüssel, Windows-Installer-Versionen, .NET Framework oder Ähnliches sein.