C# lernen

Version: 2.0 (22.02.2016)

Autor: Benjamin Jung

Veröffentlichung: Das große Comuter ABC

Das große Computer ABC

C# lernen

Einführung: Vorwort

Die Entwicklung der Programmiersprachen begann bereits in den 1840er-Jahren durch Ada Lovelace, welche einige Merkmale der heutigen Programmiersprachen festlegte. Konrad Zuse, der den ersten Computer der Welt baute, veröffentlichte 1946 die erste höhere Programmiersprache namens Plankalkül. 1972 wurde die Programmiersprache C veröffentlicht, die bis heute eine große Bedeutung hat.
Die Entwicklung der Programmiersprachen ging weiter und hält bis heute an. So wurden im 21. Jahrhundert auch die Programmiersprachen C# und Visual Basic von Microsoft ins Leben gerufen, welche auf dem sogenannten .NET Framework basieren. Durch den immer einfacher werdenden Syntax und die Allgegenwertigkeit des Computers, entscheiden sich auch immer mehr Privatpersonen dazu, eigene Programme zu entwickeln. Dabei setzen die Hobby-Programmierer weniger auf die Sprachen C und C++, vielmehr jedoch auf C#, Visual Basic oder ähnliche.
Diesen Einstieg in die Programmierung, in das .NET Framework und die Programmiersprache C# wollen wir Ihnen mit diesem ausführlichen und doch verständlich gehaltenen Tutorial nahe bringen. Egal ob Sie bereits eine andere Programmiersprache erlernt haben, oder Sie mit C# das Programmieren lernen wollen, hier sind Sie genau richtig.

Ich wünsche Ihnen viel Spaß, Erfolg und natürlich Freude mit C#.

Mit freundlichen Grüßen
Benjamin Jung
(Buchautor)

P. S: Mit dem untenstehenden Download-Icon können Sie die komplette Projektmappe aller C#-Beispielprojekte herunterladen.

Das große Computer ABC

C# lernen

Einführung: Webseite als E-Book

Viele Hobby-Programmierer sind während Ihrer „Programmierzeit“ online und können somit das Internet und dadurch auch diese Webseite erreichen. Doch immer wieder gibt es Zeiten, wo einmal kein Internet zur Verfügung steht. Genau für diese Fälle gibt es diese Webseite (also das „C#-Buch“) auch als E-Book **kostenfrei** zum Download. Das E-Book enthält, wie die Webseite auch, die Texte, Bilder und natürlich auch die Beispiel-Codes. Natürlich eignet sich das E-Book auch für Personen, die sich diese Webseite gerne als „Buch“ (z. B. mit Hilfe des Broschüren-Drucks) **ausdrucken** möchten. Mit dem untenstehenden Download-Icon können Sie das E-Book herunterladen. In den nächsten Themen und Kapiteln dient dieses Icon übrigens zum Herunterladen des jeweiligen Projektes.

Das große Computer ABC

C# lernen

Einführung: Information über C# und .NET

Was ist überhaupt C# und was bedeutet .NET?

.NET ist eine Software-Plattform von **Microsoft** mit welcher eine sogenannte **.NET-Anwendung** ausgeführt werden kann. Um solche .NET-Anwendungen ausführen zu können, ist das sogenannte **.NET Framework** von Nöten, welches bereits in den Windows-Betriebssystemen (ab Windows Server 2003 bzw. Windows Vista) enthalten ist. Dadurch können .NET-Anwendungen auf so gut wie jedem Windows Computer ausgeführt werden. Hierbei gibt es jedoch noch die Framework-Version zu beachten. In den Projekt-Einstellungen von Visual Studio, welches wir im nächsten Thema kennenlernen werden, können wir diese „Ziel-Version“ einstellen. Je weiter wir jedoch die Ziel-Version zurückstellen, umso weniger Framework-Bestandteile (z. B. Windows Forms oder WPF) kann unser Programm letztendlich enthalten. .NET-Anwendungen sind aufwärtskompatibel, d. h. Programme, die auf die Ziel-Version 2.0 eingestellt sind, können auch mit dem .NET Framework 4.0 geöffnet werden. Alle in diesem Buch enthaltenen Beispielprogramme (ausgenommen sind die Beispiele „Diagramme“ im Kapitel „Erweiterte Programmierthemen“ und „Menüband“ im Kapitel „WPF“) sind auf .NET Framework 3.5 eingestellt und somit ab dem Betriebssystem Windows XP lauffähig.

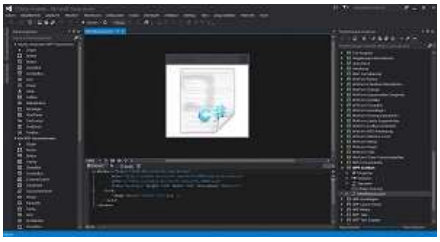
C# (auch Visual C# genannt) ist die Programmiersprache von Microsoft, mit welcher .NET-Anwendungen erstellt werden können. Eine Alternative zu C#, um .NET-Anwendungen zu programmieren, ist **Visual Basic**. Weitere etwas weniger bekannte Alternativen sind F# und J#. Alle Funktionen, Objekte etc. des .NET Frameworks sind in C# programmiert und greifen durch das Importieren von C++ DLLs (Klassenbibliotheken) auf die Windows eigenen Funktionen zu. Die Sprache C# ist, objektorientiert und ereignisorientiert. Als Entwicklungsumgebung für C# wird hauptsächlich Visual Studio von Microsoft eingesetzt. Vom Sprachsyntax ähnelt C# den weit verbreiteten Programmiersprachen C und C++. C# wird ständig und oft gleichzeitig mit dem .NET Framework erneuert und verbessert.

Die Funktionalität von .NET und C# lässt sich wie folgt erklären: Der User bzw. Programmierer erstellt einen Programmcode in C#. Nun kompiliert er den Programmcode. Daraus entsteht eine exe-Datei, welche einen **CIL-Code** (Common Intermediate Language) enthält. Zur Laufzeit des Programms wird dieser CIL-Code mit der sogenannten **CLR** (Common Language Runtime), also dem installierten .NET Framework des Ziel-Computers, und dem dazugehörigen **JIT-Compiler** („Just In Time“) in Maschinencodes übersetzt, welche durch den Prozessor verarbeitet werden können.

Das große Computer ABC

C# lernen

Einführung: Einführung in Visual Studio



Visual Studio ist eine **Entwicklungsumgebung** von Microsoft mit welchem .NET-Anwendungen und auch Programme mit Hilfe der WinAPI (C und C++) entwickelt werden können. Auch Web-Anwendungen oder Apps für Windows Phone können mit Visual Studio entwickelt werden. Visual Studio gibt es in verschiedenen Versionen. Bis einschließlich Visual Studio 2013 gab es eine **Express und Professional Edition**. Ab Visual Studio 2015 gibt es die sogenannte **Community Edition**. Im Vergleich zu der Express Edition ist die Community Edition nicht eingeschränkt und somit vergleichbar mit der Professional Edition. Des Weiteren gab es noch diverse andere Editionen, welche nur in manchen Versionen erhältlich waren (z. B. die Ultimate Edition). Sowohl die Express Edition als auch die Community Edition ist **kostenlos** erhältlich. Visual Studio besteht aus einem Editor mit **Syntax-Highlighting**, einem **Compiler** (für C, C++, C# und andere) und weiteren **Tools**.

Die Beispiele dieses C#-Kurses sind mit **Visual Studio 2013 Professional** erstellt worden. Die Projekte können jedoch auch unter neueren oder vorherigen Versionen geöffnet werden. Die Express-Versionen sind immer auf eine Programmiersprache (z. B. C++ oder C#) bzw. ein Programmiergebiet (z. B. Web Anwendungen oder Apps) beschränkt.

Um in Visual Studio Windows Forms Programme (also Programme mit grafischer Oberfläche) zu erstellen bietet uns Visual Studio einen **Designer**. Über den **Werkzeugkasten** können nun verschiedene Steuerelemente auf das Formular gezogen werden. Die Eigenschaften und Events dieses Steuerelementes können im **Eigenschaften-Fenster** editiert werden. Dazu muss davor das zu bearbeitende Steuerelement angeklickt werden. Durch das Doppelklicken auf ein Steuerelement erstellt Visual Studio automatisch eine Event-Funktion für das Standard-Event (dazu später mehr). Für WPF-Anwendungen bietet Visual Studio einen ähnlichen Designer. Hier wird jedoch zusätzlich zum grafischen Fenster auch ein **XAML-Editor** angezeigt.

Das große Computer ABC

C# lernen

Einführung: „Hallo Welt“-Programm

Wie in so gut wie jedem Programmier-Tutorial beginnen wir mit dem bekannten „Hallo Welt“-Beispiel. Das „Hallo Welt“-Beispiel ist ein einfaches Programm, in welches wir als Programmierer lediglich eine Zeile (oder falls nötig auch mehrere Zeilen) Quellcode einfügen, welcher eben den Text „Hallo Welt“ ausgibt.
Im Falle von C# werden durch die Entwicklungsumgebung bereits einige Zeilen Code eingesetzt. Dies kommt daher, dass C# eine objekorientierte Programmiersprache ist und daher eine Art „**Grundgerüst**“ benötigt. Um das Programm übersichtlicher zu halten, haben wir bereits einige Zeilen mit verschiedenen *using*-Befehlen (dazu später mehr) entfernt. Wenn wir uns den Code ansehen, sehen wir, dass wir mehrere „**Blöcke**“ haben, welche mit geschweiften Klammern gekennzeichnet sind. Diese Blöcke sind verschaltet. Im innersten Block (bei welchem es sich um die Main-Funktion handelt) sehen wir die Befehle *Console.WriteLine("Hallo Welt!");* und *Console.ReadKey();*. Wenn wir uns das Syntax-Highlighting im unteren Code anschauen, erkennen wir, dass einige Wörter hervorgehoben sind: *using*, *namespace*, *class*, *static*, *void* und string. Auf die Bedeutung von diesen sogenannten **Schlüsselwörtern** werden wir im Laufe des nächsten Kapitels genauer eingehen.

Program.cs

```
1  using System;
2
3  namespace CSV20.Hallo_Welt
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Hallo Welt!");
10             Console.ReadKey();
11         }
12     }
13 }
```

Das große Computer ABC

C# lernen

Grundlagen: Grundlegender Syntax

Der Syntax, also die verschiedenen Zeichen die verwendet werden, um bestimmte Bereiche oder Zeilen zu trennen bzw. zu gruppieren, ähnelt dem Syntax der Programmiersprache C und C++. Dies kommt daher, dass C# sich an den Konzepten von C und C++ orientiert. Jedoch gibt es auch ein paar Unterschiede. Auf den genauen Syntax gehen wir in den nächsten Absätzen detaillierter ein.

Schauen wir uns dazu den Beispiel-Code vom „Hallo Welt“-Programm etwas genauer an. Die erste Zeile bindet den **Namensraum** System ein. Hierfür wird das Schlüsselwort *using* verwendet. Ein Namensraum dient zur Gruppierung mehrerer Klassen. Da das NET Framework aus sehr vielen Klassen besteht, sind diese in verschiedenen Namensräumen untergeordnet. Der Namensraum *System* enthält unter anderem die Klasse *Console*, welche wir in den ersten Beispielen benötigen. Anstatt den Namensraum über den *using*-Befehl einzubinden, könnten wir auch bei jeder Verwendung einer Klasse aus diesem Namensraum, den Namensraum voranstellen. Ein Beispiel hierzu: Anstatt Console.WriteLine("Hallo Welt!"); könnten wir auch System.Console.WriteLine("Hallo Welt!"); schreiben. Der *using*-Befehl ist mit einem Semikolon abgeschlossen. Alle Befehle oder auch **Statements** genannt in C# werden mit einem Semikolon abgeschlossen.

Schon in der nächsten Zeile (Leerzeile übersprungen) sehen wir das Schlüsselwort *namespace* und anschließend den Namen *CSV20.Hallo_Welt*. Damit geben wir den Namensraum dieses Blocks an. Ein Projekt kann selbstverständlich mehrere Namensräume enthalten. Die Aufteilung eines Projektes in mehrere Namensräume macht jedoch nur dann Sinn, wenn das Projekt groß ist. Die nächste Zeile enthält eine öffnende geschweifte Klammer, welche in der letzten Zeile der Datei wieder geschlossen wird. Wie bereits vorher schon angesprochen, handelt es sich dabei um einen Block. Ein Block gruppiert weitere Blöcke und / oder Statements. In dem Beispielcode enthält der namespace-Block einen weiteren Block, bei welchem das Schlüsselwort *class* vorangestellt ist. Mit dem Schlüsselwort *class* deklarieren wir eine **Klasse** (dazu später mehr). Unter Visual Studio entspricht der Klassenname zumeist dem Name der C#-Datei (Endung .cs). Beim Erstellen eines Konsolen-Projektes mit Visual Studio wird standardmäßig die Klasse *Program* in der dazugehörigen Datei Program.cs erzeugt.

Der Block der Klasse *Program* enthält einen weiteren Block, welcher im Gegensatz zu den anderen etwas komplexer erscheint. Hierbei handelt es sich um eine **Funktion**. Am Anfang der Zeile befinden sich die Schlüsselwörter *static* und *void*. Diese Schlüsselwörter legen verschiedene Eigenschaften und Verhaltensweisen der Funktion fest. Die genauere Bedeutung werden wir später kennenlernen. Hinter diesen zwei Schlüsselwörtern folgt nun der Name *Main*. Jedes Programm enthält eine solche **Main-Funktion** (oder auch Main-Methode genannt). Die Main-Methode ist der Haupteinstiegspunkt des Programms, d. h. beim Starten der exe-Datei wird als erstes die Main-Methode aufgerufen. Hinter dem Namen werden runde Klammern notiert. Innerhalb dieser Klammern können Parameter festgelegt werden. Im Falle der Main-Funktion, werden die Kommandozeilen-Parameter übergeben (command line arguments). Beim Erstellen eigener Funktionen können wir natürlich selbst festlegen, welche Parameter wir übergeben wollen und wie die Funktion heißen soll.

Innerhalb des Funktion-Blocks werden die Statements *Console.WriteLine("Hallo Welt!");* und *Console.ReadKey();* notiert. *WriteLine()* und *ReadKey()* sind Funktionen, welche der Klasse Console angehören. Der *WriteLine()*-Funktion wird der Text „Hallo Welt“ übergeben. Ein solcher Text wird in der Fachsprache als Zeichenkette bezeichnet. Mit der *ReadKey()*-Funktion können wir die Konsole offen halten. Die Konsole wird dadurch so lange offen gehalten, bis eine Taste gedrückt wird. Auf weitere Befehle der Konsole gehen wir später ebenfalls noch genauer ein. Der Punkt zwischen dem Klassennamen und dem Funktionsname ist ein wichtiges Zeichen, welches seinen Einsatz in vielen Positionen findet, um zwei Namen voneinander zu trennen. Man spricht vom **Memberzugriff**.

Wichtig in der Programmierung ist auch die **Kommentierung**, also verschiedenen Blöcken oder Statements einen Kommentar hinzuzufügen, dass Sie später noch wissen, warum Sie diese Abfrage oder etwas anderes so programmiert haben. Um den Rest einer Zeile (muss nicht zwangsläufig am Anfang stehen) zu kommentieren, notieren Sie zwei Slahses hintereinander (//). Ein solcher Kommentar endet automatisch mit dem Zeilenende. Einen mehrzeiligen Kommentar können wir mit /* beginnen und mit */ beenden. Diese Art von Kommentierung eignet sich auch, wenn ein bestimmter Teil des Programmcodes vorrübergehend nicht ausgeführt werden soll.

Das große Computer ABC

C# lernen

Grundlagen: Datentypen und Variablen

In C# gibt es, so wie in anderen Programmiersprachen auch, sogenannte **Datentypen**. Ein Datentyp ist ein Typ, der beschreibt, welche Art Daten oder welcher **Wertebereich** in einer Variablen dieses Datentyps gespeichert werden kann. **Variablen** werden zur Laufzeit erstellt und sind an einen Datentyp gebunden. Grundsätzlich unterscheidet man in C# zwischen Ganzzahlen, Gleitkommazahlen (auch Fließkommazahlen genannt), Flags (also ja oder nein), Zeichen (einzelnes Zeichen oder eine sogenannte Zeichenkette als Reihe mehrerer Zeichen) und Objekten.
Für **Ganzzahlen** sind die Datentypen *byte*, *sbyte*, *short*, *ushort*, *int*, *uint*, *long* und *ulong*. Der jeweilige Wertebereich kann dem Quellcode unten entnommen werden. Das s in *sbyte* steht für *signed*, also vorzeichenbehaftet. Das u in *ushort*, *uint* und *ulong* steht dagegen für *unsigned*, also vorzeichenlos. Bei **Gleitkommazahlen** stehen die Datentypen *float*, double und *decimal* zur Verfügung. Wenn wir Gleitkommazahlen im Quelltext notieren, geht der Compiler davon aus, dass es sich um den double-Datentyp handelt. Um dem Compiler mitzuteilen, dass es sich um den *float*- oder *decimal*-Datentyp handelt werden die Suffixe f oder M benötigt, welche an das Ende der Zahl notiert werden müssen.
Für **Flags**, gibt es in C# den Datentyp *bool*. *bool* besitzt nur zwei Zustände *true* (1 bzw. wahr) und *false* (0 bzw. unwahr). Wichtig zu wissen ist jedoch, dass die heutigen Prozessoren nicht bitweise arbeiten, sondern byteweise, d. h. beim Anlegen einer *bool* Variable wird immer 1 Byte Speicher benötigt. Man könnte als meinen es entspricht dem Datentyp *byte*. Jedoch ist der C#-Compiler so programmiert, dass es nicht möglich ist in einer *bool*-Variable z. B. den Wert 2 zu speichern.
Der Datentyp *char* kann ein **Unicode-Zeichen** speichern. In einigen Sprachen, wie z. B. C und C++ gibt es den Datentyp *byte* nicht. Hier wird einfach der Datentyp *unsigned char* verwendet. Dadurch entsteht der Eindruck, dass ein *char* ebenfalls wie der Datentyp *byte* 1 Byte Speicher benötigt und somit einen Wertebereich von 0 bis 255 hat. Da Windows jedoch sehr viel mit Unicode-Zeichen arbeitet, reicht ein Byte für den Datentyp *char* nicht aus. Deshalb entspricht der Datentyp *char* in C# dem Datentyp *ushort*. Auch hier ist der C#-Compiler so programmiert, dass er das Speichern eines Zeichens in einem *ushort* niemals zulassen wird (nur über einen *cast*, dazu später mehr). Ein einzelnes Zeichen wird in einfachen Anführungszeichen notiert. Der Datentyp *string* hingegen ist eine Aneinanderreihung von einzelnen Zeichen, man spricht von einer **Zeichenkette**. Man kann also vereinfacht sagen, dass ein *string*-Datentyp aus mehreren *char*-Datentypen besteht. Eine Zeichenkette wird in doppelten Anführungszeichen angegeben. Im Vergleich zu der Sprache C und C++ ist der *string*-Datentyp eine praktische Erfindung, da dieser auch jederzeit verlängert werden kann, denn in C und C++ müssen Zeichenketten mit Hilfe eines Arrays (dazu später mehr) aus *char*-Datentypen gebaut werden. In der Realität ist jedoch der Datentyp *string* nichts anderes als ein Array mit *char*-Datentypen, welcher jedoch automatisch verlängert werden kann. Diese Verwaltung übernimmt die Programmiersprache C# für uns.
Für alle **Objekte** kann der globale Datentyp *object* verwendet werden (dazu jedoch später mehr). In C# sind alle Datentypen an Objekte gebunden, dies ist sehr ungewöhnlich, führt jedoch andererseits zu einer sehr abstrakten und einheitlichen Struktur.
Man unterscheidet bei verschiedenen Aktionen für Variablen zwischen der Variablendeklaration, Variableninitialisierung und Wertezuweisung. Bei der **Variablendeklaration** legen wir die Variablen an. Dazu bestimmen wir den Datentyp und den frei wählbaren Namen (z. B. bool bFlag;). Der Name einer Variablen sollte sinnvoll gewählt werden. Des Weiteren darf dieser nur einmal im deklarierten Block vorhanden sein. Variablen die direkt innerhalb einer Klasse definiert werden, werden als Member-Variablen bezeichnet. Die **Variableninitialisierung** ist die erste Zuweisung einer Variablen. C# ist soweit abgefangen, dass ein Zugriff auf eine uninitialisierte Variable, vom Compiler als Fehler quittiert wird. Oftmals wird eine Variable auch direkt mit der Variablendeklaration initialisiert (wie im unteren Beispielcode). Die **Wertzuweisung** und die Variableninitialisierung erfolgt indem man den Variablennamen, ein Gleichheitszeichen und den Wert notiert (z. B. bFlag = false;). Einer Variablen kann mehrmals ein gleicher oder auch anderer Wert zugewiesen werden.
Nun wollen wir noch etwas genauer auf die Speicherung von Variablen eingehen. In der Programmierung gibt es den Stack-Speicher und den Heap-Speicher. Ab diesem Zeitpunkt müssen wir bei den Datentypen zwischen zwei Typen unterscheiden: Werttypen und Referenztypen. Zu den **Werttypen** gehören die Datentypen *byte*, *sbyte*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *float*, *double*, *decimal*, *bool*, *char* und *enum* (dazu später mehr). Diese Datentypen werden alle im **Stack-Speicher** abgelegt. Zu den **Referenztypen** gehören die Datentypen *string* und *object*, bei welchen im Stack-Speicher lediglich eine Referenz zum **Heap-Speicher** abgelegt wird. Die Daten der Referenztypen liegen im Heap-Speicher. Diese Aufteilung existiert deshalb, da Datentypen vom Typ *string* und *object* u. U. einiges an Speicherplatz benötigen. Die Größe des Stack-Speichers ist jedoch begrenzt, weshalb man sich dazu entschieden hat, große Datenmengen (also Zeichenketten und Objekte) im Heap-Speicher zu speichern. Referenztypen kann der Wert *null* zugewiesen werden, sie sind *Nullable*. Der Wert *null* entspricht dem programmiertechnischen nichts und darf nicht mit der Zahl 0 verwechselt werden. Durch das Setzen eines Referenztyps auf den Wert *null* wird die Referenz gelöscht (also die Referenz auf dem Stack-Speicher). Der **Garbage Collector** (kurz GC) von C# kann nun prüfen, ob das Objekt noch von irgendeiner Stelle referenziert wird (also ob noch irgendeine Variable eine Referenz auf das Objekt hat). Falls das Objekt nicht mehr referenziert ist, kann das Objekt auf dem Heap-Speicher vom GC gelöscht werden. Der GC kann durch den Programmcode manuell aufgerufen werden, wird jedoch vom .NET Framework auch automatisch in unregelmäßigen Abständen (im Leerlauf oder bei Speicherengpässen) aufgerufen.
In allen Beispielen dieses und der weiteren Kapitel (abgesehen von dem aktuellen Beispiel unten) wird Ihnen auffallen, dass wir die Variablen am Anfang immer mit einem Kleinbuchstaben versehen haben (z. B. i, s, b oder o). Hierbei handelt es sich um die sogenannte **Namenskonvention**. Um sich im eigenen oder fremden Programm schneller zurecht zu finden, halten die meisten „besseren“ Programmierer diese Regeln ein. Die Buchstaben sind hierbei vom Datentyp abgeleitet (i für *int*, s für *string*, b für *byte* oder *bool* und o für *object*). Des Weiteren werden alle Variablen eines Blocks am Anfang definiert. Näheres

dazu am Ende des C#-Buches.

Program.cs

```
byte vByte = 200;                                   // 0 bis 255
sbyte vSByte = -45;                                 // -128 bis 127
short vShort = -15784;                              // -32.768 bis 32.767
ushort vUShort = 45960;                             // 0 bis 65.535
int vInt = -1894112307;                             // -2.147.483.648 bis 2.147.483.647
uint vUInt = 3489215047;                            // 0 bis 4.294.967.296
long vLong = -3996794549303736183;                  // -9.223.372.036.854.775.808 bis
9.223.372.036.854.775.807
ulong vULong = 14125657448224163497;                // 0 bis 18.446.744.073.709.551.615
float vFloat = 39751.48f;                           // -3.402823e38 bis 3.402823e38
double vDouble = 976252561462.7912;                 // -1.79769313486232e308 bis
1.79769313486232e308
decimal vDecimal = 6441868926456551289 68.34768426M; // +/- 1,0 × 10e?28 zu +/- 7,9 × 10e28
bool vBool = false;                                 // true (1) oder false (0)
char vChar = 'c';                                   // Unicode-Zeichen (0 - 65.535)
string vString = "Hallo Welt!";                     // Aneinanderreiung von char-Typen
object vObject = new Program();                     // globaler Typ für alle Objekte
int vZahl;
const int vZahlKonstant = 16;

Console.WriteLine(vByte);
Console.WriteLine(vSByte);
Console.WriteLine(vShort);
Console.WriteLine(vUShort);
Console.WriteLine(vInt);
Console.WriteLine(vUInt);
Console.WriteLine(vLong);
Console.WriteLine(vULong);
Console.WriteLine(vFloat);
Console.WriteLine(vDouble);
Console.WriteLine(vDecimal);
Console.WriteLine(vBool);
Console.WriteLine(vChar);
Console.WriteLine(vString);
Console.WriteLine(vObject);

Console.WriteLine();

// Variablen können zur jeder Zeit geändert werden
vZahl = 418;
vZahl = 9752;
Console.WriteLine(vZahl);

// Konstante Variablen können nicht geändert werden
// vZahlKonstant = 123; --> nicht möglich, da Konstant
Console.WriteLine(vZahlKonstant);

Console.ReadKey();
```

Das große Computer ABC

C# lernen

Grundlagen: Konsolen-Funktionen

Wie in unserem 1. Beispiel wollen wir uns als erstes etwas näher mit den Konsolen-Funktionen von C# beschäftigen. Die Konsolen-Funktionen von C# befinden sich in der Klasse *Console*. Die Klasse *Console* verfügt über einige Funktionen und Eigenschaften. Einige wichtige Eigenschaften und Funktionen wollen wir uns jetzt etwas genauer anschauen.
Über die Eigenschaft *Title* können wir den Titel des Konsolen-Fensters ändern. Der Titel wird in der Titelleiste des Fensters angezeigt. Über die Eigenschaften *BackgroundColor* und *ForegroundColor* kann die Hintergrund- und Schriftfarbe geändert werden. Hierfür wird ein Wert der **Enumeration** (dazu später mehr) *ConsoleColor* benötigt. Ein kleiner Tipp: Visual Studio hilft uns bei der Eingabe von unbekannten Datentypen durch das Tool **IntelliSense,** welches eine Art Auto-Vervollständigung mit Hinweisen bietet. Wenn wir den unteren Code kompilieren und ausführen sehen wir, dass nur der Text „Ihre Eingabe:“ mit blau hinterlegt ist, das kommt daher, da nach dem Ändern der Hintergrundfarbe die Clear-Funktion nicht aufgerufen wurde, welche den kompletten Konsolen-Inhalt löscht und dabei auch alle Zellen mit der richtigen Hintergrundfarbe füllt. Wird dies nicht gemacht, werden nur alle ausgegebenen oder durch den Nutzer eingegebenen Informationen mit dieser Farbe hinterlegt.
Um etwas auf die Konsole zu schreiben, gibt es die *Write()*- und *WriteLine()*-Funktion. Die Write-Funktion führt im Gegensatz zu der *WriteLine()*-Funktion keinen Zeilenumbruch aus. Natürlich könnten wir auch mit der Write-Funktion den Zeilenumbruch „manuell“ auslösen (Console.Write("
");), was jedoch eher unüblich ist. Die *Write()*- und *WriteLine()*-Funktion erwartet als Parameter standardmäßig einen Parameter, bei dem der Typ nicht von Bedeutung ist (dies kommt daher, dass die Funktionen mit verschieden Datentypen überladen sind, dazu später mehr). Ein Aufruf von *WirteLine()* ohne Parameter erzeugt einen Zeilenumbruch. Um eine Eingabe des Benutzers abzufangen und zu bearbeiten, können wir die Funktion *ReadLine()* und *ReadKey()* verwenden. *ReadKey()* liest nur ein Tastendruck ein, wohingegen *ReadLine()* solange liest, bis die Eingabe-Taste (Enter) gedrückt wird. Die *ReadLine()*-Funktion wird gerne verwendet um eine **Eingabe des Benutzers** zu erhalten, dabei gibt die Funktion eine Zeichenkette (*string*) zurück. Die *ReadKey()*-Funktion wird gerne am Ende einer Konsolen-Anwendung verwendet, um die Konsole so lange offen zu halten bis der Benutzer eine Taste drückt. Oftmals wird in einem solchen Fall auch ein wie Text „Bitte beliebige Taste drücken …“ ausgegeben, welcher mit der Write-Funktion zuvor auf die Konsole geschrieben werden muss. Die Funktion *SetCursorPosition()* setzt die Cursor-Position. In der Konsole wird eine **Rasterschrift** verwendet, wodurch sich die Konsole in Spalten und Zeilen einteilen lässt. Um nun den Cursor auf eine einzelne Zelle zu setzen, benötigen wir die Nummer der Zeile und die Nummer der Spalte. Zu beachten ist, dass die Zählung bei 0 beginnt. Die Position 0,0 ist also die erste Spalte und die erste Zeile. Die erste Zahl bestimmt die Spalte und die zweite Zahl die Zeile. Die Funktion *Clear()* leert die komplette Konsole. Dabei wird auch die Hintergrundfarbe für alle Zellen übernommen.

Program.cs

```
1  string sEingabe;
2
3  // Einstellungen ändern
4  Console.Title = "C#-Buch 2015";
5  Console.BackgroundColor = ConsoleColor.Blue;
6  Console.ForegroundColor = ConsoleColor.White;
7
8  // Eingabe auffordern
9  Console.Write("Ihre ");
10 Console.WriteLine("Eingabe:");
11 Console.SetCursorPosition(5, 1);
12 sEingabe = Console.ReadLine();
13
14 // Eingabe ausgeben
15 Console.WriteLine();
16 Console.WriteLine("Ihre Eingabe: " + sEingabe);
17 Console.WriteLine();
18 Console.WriteLine("Drücken Sie eine Taste ...");
19 Console.ReadKey();
20
21 // nachdem Löschen der Konsole ("Clear") ist der komplette
22 // Hintergrund an die gesetzte Farbe angepasst
23 Console.Clear();
24 Console.ForegroundColor = ConsoleColor.Magenta;
25
26 Console.WriteLine("Hallo Welt!");
27
28 Console.ReadKey();
```

Das große Computer ABC

C# lernen

Grundlagen: Mathematische Operationen

Standard-Operationen wie **Addition**, **Subtraktion**, **Multiplikation** und **Division** lassen sich in C# mit den gewöhnlichen Zeichen + - * / durchführen. Diese Zeichen und auch andere Zeichen (dazu später mehr) werden als **Operator** bezeichnet. Der Plus-Operator wird bei Zeichenketten hingegen dazu verwendet, mehrere Teilzeichenketten miteinander zu verketten. Im unteren Quellcode sehen wir, dass bei der Addition und Subtraktion eine Ganzzahl mit einer Fließkommazahl addiert bzw. subtrahiert wird. In dieser Zeile führt der Compiler automatisch eine **Typumwandlung** durch (die Ganzzahl wird in double umgewandelt), andernfalls wäre die Operation nicht möglich. Natürlich kann nicht nur mit konstanten Werten gerechnet werden, sondern auch mit Variablen. Um den Rest einer Division zu ermitteln, können wir den Modulo-Operator (%) verwenden.
Eine Klasse mit einigen Funktionen für erweiterte mathematische Operationen ist die Klasse *Math*. Hier sind neben Funktionen für das mathematische **Runden** (*Floor()* zum Abrunden, *Ceiling()* zum Aufrunden und *Round()* zum kaufmännischen Runden und Funktionen zur **Wurzel-Rechnung** (*Sqrt()* square-root) und **Exponenten-Rechnung** (*Pow()* x^y) auch Konstanten z. B. für die Zahl *PI* vorhanden. Auch Funktionen zur **Trigonometrie** befinden sich in der Math-Klasse.

Program.cs

```
1  // Standarmäßige mathematische Operationen
2  Console.WriteLine(4 + 8.5);
3  Console.WriteLine(2 - 45.7);
4  Console.WriteLine(9 * 7);
5  Console.WriteLine(81 / 3);
6
7  // Runden
8  Console.WriteLine(Math.Floor(48.41));
9  Console.WriteLine(Math.Ceiling(48.41));
10 Console.WriteLine(Math.Round(48.41));
11
12 // Quadratwurzel und Exponenten-Rechnung
13 Console.WriteLine(Math.Sqrt(81));
14 Console.WriteLine(Math.Pow(9, 3));
15
16 // Mathematische Konstanten
17 Console.WriteLine(Math.PI);
18
19 Console.ReadKey();5
```

Das große Computer ABC

C# lernen

Grundlagen: Binäre Operationen

Binäre Operationen sind Operationen, bei denen die Bits eines oder mehrere Bytes verändert oder geprüft werden. Eine wichtige Operation ist die **Schiebe-Operation**, bei der die Bits nach links (>>) oder rechts (<<) verschoben werden. Um Bits miteinander zu **verunden** (AND), wird das Und-Zeichen (&) verwendet. Die **Veroderung** (OR) von Bits wird mit einem Senkrechtstrich (|, auch Pipe genannt) durchgeführt. Zusätzlich zur normalen Veroderung gibt es noch das sogenannte **Exklusiv-Oder** (XOR). Exklusiv-Oder wird in der Programmierung mit dem Zirkumflex (^) dargestellt. Um die Bits eines Wertes zu tauschen (**negieren**), wird das Tilde-Zeichen (~) verwendet.
Eine wichtige Information zu allen Operatoren in C# zum Abschluss: Eine Programm-Zeile wie a = a + b lässt sich mit a += b **verkürzt darstellen**. Diese Verkürzung ist natürlich nicht nur bei mathematischen Operationen möglich, sondern kann auch für binäre Operationen angewendet werden (z. B. aus a = a & b wird a &= b). Für die Addition und Subtraktion gibt es des Weiteren noch eine weitere Variante zur verkürzten Darstellung, die gerne bei Schleifen (dazu später mehr) verwendet wird: Aus a = a + 1 bzw. a += 1 wird a++ oder auch ++a. Die Notation von a++ und ++a unterscheidet sich wie folgt: Stellen wir uns vor, der aktuelle Wert von a ist 1 und a wird einer anderen Variablen namens b gespeichert: Bei b = a++ besitzt b nun den Wert 1 (also den Wert von a vor der Addition) und a den Wert 2. Bei b = ++a besitzt b genauso wie a den Wert 2. Hier wird also die Addition von a ausgeführt, bevor er der Variablen b zugewiesen wird. Das gleiche Prinzip gilt auch für die Subtraktion (a-- und --a).

Program.cs

```
1  // Standarmäßige mathematische Operationen
2  Console.WriteLine(4 + 8.5);
3  Console.WriteLine(2 - 45.7);
4  Console.WriteLine(9 * 7);
5  Console.WriteLine(81 / 3);
6
7  // Runden
8  Console.WriteLine(Math.Floor(48.41));
9  Console.WriteLine(Math.Ceiling(48.41));
10 Console.WriteLine(Math.Round(48.41));
11
12 // Quadratwurzel und Exponenten-Rechnung
13 Console.WriteLine(Math.Sqrt(81));
14 Console.WriteLine(Math.Pow(9, 3));
15
16 // Mathematische Konstanten
17 Console.WriteLine(Math.PI);
18
19 Console.ReadKey();
```

Das große Computer ABC

C# lernen

Grundlagen: Funktionen für Zeichenketten

Der Datentyp *string* bietet einige Funktionen, um eine Zeichenkette zu **verändern** oder dessen Inhalt zu **prüfen**. Die Eigenschaft *Length* gibt die Länge der Zeichenkette zurück. Mit den Funktionen *ToUpper()* und *ToLower()* können wir alle in der Zeichenkette vorkommende Buchstaben in Groß- oder Kleinbuchstaben umwandeln. Die *Trim*-Funktion entfernt alle Leerzeichen am Anfang und am Ende der Zeichenkette. Grundsätzlich gilt für alle Zeichenketten, dass durch den Aufruf einer solchen Funktion die ursprüngliche Zeichenkette **nicht verändert wird**, sondern die geänderte Zeichenkette zurückgegeben wird. Möchte man also die Änderungen in der Quell-Zeichenkette speichern, müsste man z. B. sText = sText.ToUpper() notieren.
Eine Zeichenkette besteht, wie wir bereits gelernt haben, aus einzelnen Zeichen (*char*). Mit den Funktionen *Split()* und *Substring()* können wir eine Zeichenkette in verschiedene Teile **zerlegen**. Die Funktion *Split()* trennt die Zeichenkette an dem angegebenen Zeichen (Typ *char*) bzw. an der angegebenen Teilzeichenkette (Typ *string*). Als Rückgabe erhalten wir ein Array (dazu später mehr) bzw. vereinfacht gesagt eine Liste aus Teilzeichenketten. Im 1. Beispiel trennen wir unseren Text an allen Leerzeichen und zählen die Anzahl der Teilzeichenketten (Eigenschaft *Length* des Arrays). Vereinfacht gesagt, zählen wir die Anzahl der Wörter (mal abgesehen von den zusätzlich hinzukommenden führenden und endeten Leerzeichen). Die Split-Funktion erwartet als Parameter ein **einzelnes Zeichen** (*char*) oder ein Array (Liste) von *char*-Elementen (new char[] { 'a', 'b', 'c' }). Wenn wir nicht an einem einzelnen Zeichen trennen wollen, können wir die Split-Funktion auch mit einem Array von Zeichenketten (Typ *string*) aufrufen (siehe 2. Beispiel). Hier wollen wir den Quelltext an Leerzeichen und Bindestrichen trennen. Das Beispiel sieht am Anfang vielleicht etwas verwirrend aus, da wir hier ebenfalls ein *char*-Array hätten verwenden können, da es sich ja nur um einzelne Zeichen handelt. Hiervon wollen wir uns jedoch nicht verwirren lassen. Dort wird der Split-Funktion noch ein 2. Parameter übergeben, in welchem *RemoveEmptyEntries* der *StringSplitOptions*-Enumeration übergeben wird. Dadurch werden alle Teilzeichenketten, welche leer sind, nicht mit in das Ziel-Array übernommen. Dieser 2. Parameter ist immer notwendig, wenn wir der Split-Funktion ein string-Array übergeben. Um leere Elemente nicht zu ignorieren, muss der Wert *None* für die *StringSplitOptions*-Enumeration verwendet werden. Die Funktion Substring() **extrahiert** eine Teilzeichenkette aus einer Zeichenkette. Als Parameter müssen hier der **Startindex** (1. Zeichen = Index 0) und die **Länge** übergeben werden, wobei der Längen-Parameter optional ist.
Immer wieder kommt es vor, dass wir die **Position** eines bestimmten Zeichens oder einer Teilzeichenkette von einer Zeichenkette bestimmen wollen. Hierfür dient die Funktion *IndexOf()* und *LastIndexOf()*. Wobei *IndexOf()* das erste Vorkommen zurückgibt und *LastIndexOf()* das letzte Vorkommen. Wie der Name schon vermuten lässt, gibt die Funktion einen Index zurück, d. h. wenn das gesuchte Zeichen sich ganz am Anfang befindet, gibt die Funktion den Wert 0 zurück. Bleibt die Suche erfolglos, erhalten wir *-1* als Rückgabewert. *IndexOf()* und *LastIndexOf()* kann sowohl ein einzelnes Zeichen (*char*) als auch eine ganze Zeichenkette (*string*) suchen.
C# bietet noch weitere Funktionen, um eine Zeichenkette zu **prüfen**. Die Funktion *Contains()* prüft, ob die angegebene Teilzeichenkette in der Zeichenkette vorhanden ist. Dabei spielt es keine Rolle, ob das gesuchte einmal oder mehrmals vorkommt. Die Funktion *StartsWith()* prüft, wie der Name schon sagt, ob die Zeichenkette mit dem gesuchten Text beginnt. Bei der Funktion *EndsWith()* wird geprüft, ob die Zeichenkette mit dem gesuchten Text endet. Alle 3 Funktionen arbeiten **case-sensitive**, d. h. dass die Groß- und Kleinschreibung beachtet werden muss. Wollen Sie eine Prüfung durchführen, bei der die Groß- und Kleinschreibung keine Rolle spielt, müssen Sie die Zeichenkette erst in Groß- oder Kleinbuchstaben umwandeln (z. B. "Hallo wELt".ToUpper().Contains("WELT").
Die Funktion *Insert()* fügt eine gewisse Teilzeichenkette an einer gewünschten Position ein. Dazu wird als 1. Parameter die Position (0-basierend) übergeben und als 2. Paramater der einzufügende Text. *Remove()* ist das Gegenstück zur Insert-Funktion. Hiermit kann eine gewisse Anzahl an Zeichen (2. Parameter) an der angegebenen Position (ebenfalls 0-basierend) entfernt werden.

Program.cs

```
1  const string sText = "   > Hallo vom C#-Buch V2.0!    ";
2
3  Console.WriteLine(sText);
4  Console.WriteLine(sText.Length);
5
6  Console.WriteLine();
7
8  // Konvertierungs-Funktionen
9  Console.WriteLine(sText.ToLower());
10 Console.WriteLine(sText.ToUpper());
11 Console.WriteLine(sText.Trim());
12
13 Console.WriteLine();
14
15 // Trennungs-Funktionen
16 Console.WriteLine(sText.Split(' ').Length);     // Alternative:
   Console.WriteLine(sText.Split(new char[] { ' ' }).Length);
17 Console.WriteLine(sText.Split(new string[] { " ", "-" },
   StringSplitOptions.RemoveEmptyEntries).Length);
```

```
18  Console.WriteLine(sText.Substring(4, 5));
19
20  Console.WriteLine();
21
22  // Such-Funktionen
23  Console.WriteLine(sText.IndexOf(' '));
24  Console.WriteLine(sText.LastIndexOf(' '));
25
26  Console.WriteLine();
27
28  // Prüf-Funktionen (gibt true oder false zurück)
29  Console.WriteLine(sText.Contains("hallo")); // wird nicht gefunden (mit "Hallo" wäre es
    gefunden worden)
30  Console.WriteLine(sText.StartsWith("C"));
31  Console.WriteLine(sText.EndsWith("!"));
32
33  Console.WriteLine();
34
35  // Veränderungs-Funktionen
36  Console.WriteLine(sText.Insert(24, "1"));   // aus V2.0 wird V2.10
37  Console.WriteLine(sText.Remove(9, 4));      // entfernt "vom"
38
39  Console.ReadKey();
```

Das große Computer ABC

C# lernen

Grundlagen: Formatieren von Werten

Um Werte, also Zahlen, formatiert darzustellen, benötigen wir eine **Formatierungs-Funktion**. Hierfür eignet sich die Funktion *Format()* des Datentyps string (*string.Format()*). Als 1. Parameter müssen wir der Format-Funktion unsere Formatierungs-Zeichenkette übergeben. Nun können wir der Funktion noch weitere 1 bis 3 Parameter übergeben, welche zur Formatierung verwendet werden sollen. Der Datentyp der Parameter spielt dabei keine Rolle. Falls wir mehr als 3 Werte formatieren wollen oder müssen, kann auch ein Array mit zu formatierenden Werten übergeben werden. Um eine Formatierung durchführen zu können, benötigen wir bestimmte Zeichen und **Kennungen**. Ein **Verweis** auf einen zu formatierenden Wert muss innerhalb der Zeichenkette in geschweifte Klammern gesetzt werden. Innerhalb der geschweiften Klammern wird nun der **Index** des zu formatierenden Werts angegeben. Falls keine weiteren Angaben getroffen werden, wird die genauere Formatierung des Wertes durch den Compiler festgelegt. Um die konkrete Formatierung selbst festzulegen, kann hinter dem Index ein Doppelpunkt notiert werden. Hinter dem Doppelpunkt erfolgt nun die Angabe der Formatierung. Hierfür ist die Kenntnis von verschiedenen **Suffixen** von Nöten: d steht für Dezimal, c für Währung (im Beispiel wird das €-Zeichen vermutlich als Fragezeichen dargestellt!), F für Fließkommazahlen, n für Fließkommazahlen mit Tausendertrennzeichen und X für hexadezimale Schreibweise. Diese Suffixe lassen sich teilweise weiter spezifizieren: z. B. durch F1 wird eine Fließkommazahl mit einer Nachkommastelle angezeigt. Mit d6 wird eine Dezimalzahl mit Nullen aufgefüllt (z. B. wird 1234 zu 001234). Die Funktion *Console.Write()* und *Console.WriteLine()* enthält bereits eine Formatierungs-Funktion, wodurch wir *string.Format()* bei Konsolen-Anwendungen nicht unbedingt brauchen. Ein Beispiel zur alternativen Formatierung mit der *WriteLine()*-Funktion der *Console*-Klasse ist unten im Programmcode dargestellt.

Program.cs

```
1  Console.WriteLine(string.Format("E-Book-Version {0:F1} vom Jahr {1}", 2, 2015));
2  // für Konsolen-Programme auch: Console.WriteLine("E-Book-Version {0:F1} vom Jahr {1}", 2,
   2015);
3
4  Console.WriteLine(string.Format("{0:d6} {0:c} {0:n} {0:X4}", 1234));
5  // für Konsolen-Programme auch: Console.WriteLine("{0:d6} {0:c} {0:n} {0:X4}", 1234);
6
7  Console.ReadKey();
```

Das große Computer ABC

C# lernen

Grundlagen: Abfragen

Ein wichtiger Bestandteil von Programmen sind Abfragen. Für Abfragen können wir entweder den **if-Befehl** oder das **switch-case-Statement** verwenden.
Der if-Befehl kennzeichnet sich durch das Schlüsselwort *if*. Eine Abfrage mit *if* setzt sich aus dem Schlüsselwort *if* und einer Bedienung zusammen. Die Bedienung muss in runden Klammern notiert werden. Dabei muss diese immer vom Typ *bool* sein. Typische Operatoren für Vergleiche sind == (ist gleich), != (ist ungleich), > (größer als), >= (größer als oder gleich), < (kleiner), und <= (kleiner gleich). Natürlich kann auch eine bool-Variable oder der Rückgabewert einer Funktion (Voraussetzung ist der Rückgabewert bool) geprüft werden. Ein **„boolscher" Wert** kann durch das Ausrufezeichen negiert (also umgekehrt) werden. Die *if*-Abfrage wird immer von einem **Block** (geschweifte Klammern) gefolgt. Diese geschweiften Klammern können bei Blöcken von *if*-Abfragen und Schleifen (dazu später mehr) weggelassen werden, wenn nur 1 Statement notiert wird. Wenn wir bei einer *if*-Abfrage auch den Fall abfangen möchten, wenn die Bedingung nicht zutrifft, benutzen wir das Schlüsselwort *else*. Falls wir noch zusätzlich andere Bedingungen prüfen werden, welche eigene Anweisungen haben sollen, können wir mit den Schlüsselwörtern *else if* einen oder mehrere weitere Blöcke bilden. Natürlich können wir auch mehrere Bedingungen **verknüpfen**: Dafür dient die Verundung (&&) und die Veroderung (||). Bei der Kombination von „und" und „oder" können bzw. müssen wir noch weitere Klammern setzen.
Der *switch-case*-Befehl besteht aus 2 Teilen: Dem **zu prüfenden Wert** und den verschiedenen **zu überprüfenden Werten**. Der zu prüfende Wert kennzeichnet sich durch das vorangestellte Schlüsselwort *switch*. Der Wert wird dabei wie bei einer *if*-Abfrage in Klammern notiert. Der *switch*-Befehl bildet einen eigenen Block, in welchem mehrere *case*-Statements notiert werden können. Die Klammern bei einem *switch*-Block können nicht weggelassen werden. Ein *case*-Statement bildet sich aus dem Schlüsselwort, dem zu überprüfenden Wert und einem Doppelpunkt. Nun können weitere Befehle notiert werden, welche ausgeführt werden sollen, wenn der zu prüfende Wert mit dem zu überprüfenden Wert übereinstimmt. Auch hier bildet sich ein Block, wobei die Klammern nicht unbedingt notiert werden müssen. Am Ende aller Bedingungen für diesen Block muss jedoch immer der Befehl *break* notiert werden. Des Weiteren können *case*-Bedingungen gestapelt werden, so dass die Befehle ausgeführt werden, wenn eine der Bedingungen zutrifft. Auch beim *switch-case* gibt es eine Art *else*-Zweig (Alternativ-Zweig): Bei *switch-case* wird es jedoch mit *default* gekennzeichnet.
Für Programmieranfänger ist unter anderem dieses Thema etwas schwieriger zu verstehen, weshalb das dazugehörige Beispiel sehr wichtig ist. Eine Frage z. B. die sich Beginner stellen: Warum gibt es zwei verschiedene Arten von Abfragen? Und wann sollte ich welche verwenden? Die *switch-case*-Abfragen werden dann eingesetzt, wenn **mehrere Werte mit demselben Quellwert** überprüft werden sollen. Hierbei ist die *switch-case*-Abfrage vor allem sehr **übersichtlich**. Wenn es aber natürlich nur ein oder zwei „Möglichkeiten" (also den *if*- und evtl. den *else*-Zweig) gibt, macht ein *switch-case* keinen Sinn. Des Weiteren kann die *switch-case*-Bedingung nur verwendet werden, wenn ein **einzelner Wert** überprüft werden soll. Eine Verundung oder Veroderung ist nur bei *if*-Abfragen möglich. Für die Überprüfung von Zahlen gilt: Ein **Wertebereich** kann nur mit *if*-Abfragen ermöglicht werden.
Eine gerne verwendete, verkürzte Version einer *if*-Abfrage ist die sogenannte **einzeilige if-Abfrage** (inline *if* statement). Diese kann nur angewendet werden, wenn sowohl beim zutreffen der Bedingung, als auch bei dem nicht zutreffen der Bedingung, ein Statement ausgeführt werden soll, welches einen Wert vom gleichen Typ zurückgibt. Dafür wird Bedingung notiert und von einem Fragezeichen gefolgt. Dahinter folgt das Statement, welches ausgeführt werden soll, wenn die Bedingung zutrifft. Zum Schluss folgen noch ein Doppelpunkt und das Statement, welches ausgeführt werden soll, wenn die Bedingung nicht zutrifft. Im unteren Beispiel ist diese spezielle *if*-Abfrage nicht aufgeführt, Sie werden jedoch innerhalb unseres C#-Buchs ein paar Stellen finden, wo eine solche verkürzte *if*-Abfrage verwendet wird.

Program.cs

```
1  string sEingabe;
2
3  Console.Write("Bitte geben Sie etwas ein: ");
4  sEingabe = Console.ReadLine();
5
6  // Bedingung(en) prüfen
7  if (sEingabe == "switch")
8  {
9      Console.Write("Bitte geben Sie nochmals etwas ein: ");
10     switch (Console.ReadLine())
11     {
12         case "A":
13             Console.WriteLine("Sie haben A eingegeben");
14             break;
15         case "B":
16             Console.WriteLine("Sie haben B eingegeben");
17             break;
18         case "C":
19             Console.WriteLine("Sie haben C eingegeben");
20             break;
21         // case-Bedingungen können gestappelt werden
```

```
22          case "D":
23          case "E":
24              Console.WriteLine("D oder E wurde eingegeben");
25              break;
26          default:
27              Console.WriteLine("Eingabe ungültig!");
28              break;
29      }
30  }
31  else if (sEingabe == "Eingabe" || sEingabe == "Verarbeitung" || sEingabe == "Ausgabe")
32      Console.WriteLine("C# macht Spaß!");
33  else
34      Console.WriteLine("Es ist etwas anderes passiert!");
35
36  Console.ReadKey();
```

Das große Computer ABC

C# lernen

Grundlagen: Schleifen

Schleifen dienen zur ein- oder **mehrmaligen Ausführung** eines Befehls oder mehrerer Befehle. Die Schleife bildet dabei immer einen **Block** (also mit geschweiften Klammern). In manchen Fällen kann es auch auftreten, dass die Befehle des Schleifen-Blocks nicht ausgeführt werden. Dies tritt auf, wenn die **Bedingung** der Schleife nicht zutrifft. Der wichtigste Bestandteil einer Schleife ist nichts anderes als eine *if*-Abfrage. Trifft die Bedingung zu, werden die Befehle im Block ausgeführt. Trifft die Bedingung nicht zu, wird die Schleife beendet. C# bietet 4 verschiedene Schleifen: *for*, *foreach*, *while* und *do-while*. Die *foreach*-Schleife werden wir erst später behandeln, wenn wir beim Thema Arrays sind.
Die **for-Schleife** eignet sich hervorragend für das n Mal ausführen der Schleifen-Befehle. Die *for*-Schleife kennzeichnet sich durch das Schlüsselwort *for* und die runden Klammern. Innerhalb der runden Klammern werden **3 Argumente** notiert, welche durch ein Semikolon getrennt werden. Das 1. Argument dient zur **Variablendeklaration und Variableninitialisierung**. Dieses Argument wird ausgeführt bevor die Schleife durchlaufen wird. Im 2. Argument übergeben wir die **Bedingung**. Die Bedingung wird vor jeder Ausführung der Befehle der Schleife überprüft (inkl. vor der ersten Ausführung). Natürlich können auch hier mehrere Bedingungen geprüft werden (Verundung oder Veroderung). Das 3. Argument ist ein **Befehl**, welcher ausgeführt wird, **bevor die Bedingung** überprüft wird. Bei einer typischen *for*-Schleife wird hier eine Variable hochgezählt (siehe Beispiel). Für diese Variable, den sogenannten **Schleifen-Index** oder Schleifen-Zähler, wird gerne der Name i (für Index) verwendet. Oftmals werden bei verschachtelten Schleifen (also einer Schleife in einer Schleife) der nächste Buchstabe des Alphabets gewählt (also j, k, l, m, …). Die einzelnen Argumente können weggelassen werden. Die Semikolons müssen jedoch bestehen bleiben. Der Vorteil an einer *for*-Schleife ist, dass wir eine Variable direkt im „Schleifen-Kopf“ (erste Zeile der Schleife) initialisieren können. Des Weiteren können bei einer *for*-Schleife, wie bei einer *if*-Abfrage auch, die geschweiften Klammern weggelassen werden, wenn nur ein Befehl untergeordnet ist.
Die **while-Schleife** arbeitet ähnlich wie die *for*-Schleife, jedoch ist hier eine Variableninitialisierung nicht über die Schleife selbst möglich. Auch den Befehl, welcher vor der Bedingungs-Prüfung ausgeführt wird, gibt es hier nicht. Eine *while*-Schleife hat also immer nur **ein Argument** (die Bedingung selbst), weshalb es auch keine Semikolons innerhalb der runden Klammern gibt. Vom Syntax her ist die *while*-Schleife also identisch mit einer **if-Abfrage** (abgesehen vom Schlüsselwort selbst). Die Bedingung bei einer *while*-Schleife wird immer vor der Ausführung des Blocks geprüft. Bei einer *while*-Schleife dürfen die geschweiften Klammern ebenfalls weggelassen werden, wenn nur ein Befehl untergeordnet ist.
Die **do-while-Schleife** ist von der Arbeitsweise fast gleich wie die *while*-Schleife. Der einzige Unterschied ist, dass bei der *do-while*-Schleife die Bedingung erst immer am **Ende der Schleife** geprüft wird. Trifft die Bedingung zu, werden die Befehle im Block erneut ausgeführt. Dies hat zur Folge, dass die Befehle der *do-while*-Schleife **mindestens 1-mal** ausgeführt werden. Zum Syntax der *do-while*-Schleife gilt zu sagen, dass das Schlüsselwort *do* einen Block bildet. Hinter der geschweiften Klammer notieren wir nun das Schlüsselwort *while* und die Bedingung, welche in runde Klammern gestellt wird. Das **Semikolon** hinter der schließenden runden Klammer der Bedingung darf **nicht vergessen** werden.
Um die Funktionalität von Schleifen zu **steuern**, gibt es die Schlüsselwörter *break* und *continue*. Mit *break* können Sie eine Schleife verlassen. *continue* führt dazu, dass die aktuelle Ausführung des Schleifen-Blocks beendet wird und die Schleifen-Prüfung erneut ausgeführt wird. Trifft die Bedingung zu, wird der Schleifen-Block erneut ausgeführt. Bei einer *for*-Schleife wird vor der Prüfung der Bedingung der Schleifen-Befehl ausgeführt.
Im unteren Beispiel sehen Sie, wie wir mit allen 3 Schleifen die gleiche Ausgabe erzeugen können. Wie am Programmcode zu erkennen ist, ist hier die for-Schleife am übersichtlichsten. Dies liegt daran, dass wir eine Art **Zähl-Vorgang** durchführen, wofür die *for*-Schleife grundsätzlich am besten geeignet ist. In einigen anderen Fällen ist jedoch eine *while*- oder *do-while*-Schleife eher angemessen. Welche Schleife Sie verwenden sollten, müssen Sie immer dann überlegen, wenn Sie eine Schleife benötigen.

Program.cs

```
1  // Variablen werden für while und do-while benötigt
2  int j, k;
3
4  Console.WriteLine("for-Schleife:");
5  for (int i = 0; i < 10; i++)
6  {
7      Console.WriteLine(i);
8  }
9
10 Console.WriteLine();
11
12 Console.WriteLine("while-Schleife:");
13 j = 0;
14 while (j < 10)
15 {
16     Console.WriteLine(j);
17     j++;
18 }
19
20 Console.WriteLine();
```

```
21
22  Console.WriteLine("do-while-Schleife:");
23  k = 0;
24  do
25  {
26      Console.WriteLine(k);
27      k++;
28  } while (k < 10);
29
30  Console.ReadKey();
```

Das große Computer ABC

C# lernen

Grundlagen: Arrays

Arrays sind eine **Ansammlung** von Werten des gleichen Datentyps. Dabei werden die Werte innerhalb einer Variablen gespeichert. Man könnte sagen, dass ein Array eine **Liste** ist. Wir werden jedoch später ein Objekt namens *List* kennenlernen, mit welchem wir ein „besseres“ Array erzeugen können. Wichtig zu wissen ist, dass ein Array **nicht vergrößert** werden kann. Ein Array hat immer die Anzahl x an Werten, weshalb es auch als **statisches Array** bezeichnet wird.
Bei den zumeist eingesetzten Arrays handelt es sich um sogenannte **eindimensionale Arrays**, d. h. sie besitzen nur eine Dimension (eine Spalte mit mehreren Zeilen). Um ein Array zu deklarieren, müssen wir bei der Variablendeklaration hinter dem Datentyp die **eckigen Klammern** ([]) notieren. Bei der Variableninitialisierung benötigen wir nun das Schlüsselwort *new*, welches vor dem Datentyp notiert wird. Die genaue Bedeutung von *new* werden wir im Thema Objektorientierung genauer kennenlernen. In den Klammern notieren wir nun die **Anzahl der Elemente**. Um auf die einzelnen Elemente zugreifen zu können, benötigen wir die **Indexierung**. Hierbei wird der Variablenname mit den eckigen Klammern notiert, in welcher der Index angeben wird. Der Index ist also ein Zähler. Die Zählweise beginnt bei 0 und entspricht dem 1. Element. Über die Eigenschaft *Length* können wir die Anzahl der Elemente abrufen.
Um ein Array zu **verändern** oder darin etwas zu **suchen**, gibt es die Klasse *Array*. Mit der Funktion *Sort()* können wir ein Array sortieren. Dabei wird es mit Hilfe des sogenannten Standard-Vergleichs aufsteigend sortiert (numerisch bei Zahlen, alphabetisch bei Zeichenketten). Die Funktion *Reverse()* kehrt die Reihenfolge aller Array-Elemente um. Mit der Funktion *IndexOf()* und *LastIndexOf()* können wir das erste bzw. letzte Vorkommen einer Zahl, eines Zeichens oder einer Zeichenkette im Array suchen. Die Funktionen geben den Index zurück und sind somit ebenfalls null-basierend.
Ein **zweidimensionales Array** hat x Spalten und y Zeilen. Diese Angabe wird bei der Variableninitialisierung in den eckigen Klammern angegeben. Dabei werden die Werte durch ein **Komma** getrennt. Bei Variablendeklaration muss innerhalb der eckigen Klammern ebenfalls das Komma notiert werden. Die Anzahl darf jedoch, wie beim eindimensionalen Array auch, nicht angegeben werden. Bei der Indexierung muss nun ebenfalls die Spalte und Zeile angegeben werden (durch Komma getrennt). Das zweidimensionale Array erinnert also an eine **Tabelle**. Über die Funktion *GetLength()* können wir die Anzahl der Elemente der gewünschten Ebene abrufen. Dazu müssen wir als Parameter den Index der Ebene übergeben: 0 = 1. Ebene = Spalte, 1 = 2. Ebene = Zeile.
Natürlich ist es auch möglich, Arrays mit mehr als 2 Dimensionen zu erstellen. Grundsätzlich spricht man von **mehrdimensionalen Arrays**. Es müssen dementsprechend weitere Kommas und Zahlen bei der Deklaration, Initialisierung und Indexierung angegeben werden. Meistens werden Arrays mit mehr als 3 Dimensionen nicht eingesetzt, da diese sehr unübersichtlich sind.
In C# gibt es noch einen weiteren Array-Typ: **verzweigte Arrays**. Ein verzweigtes Array ähnelt einem zweidimensionalen Array. Jedoch gibt es einen entscheidenden Unterschied: Die Anzahl der Elemente der 2. Ebene müssen nicht gleich groß sein. Dies kommt daher, dass ein verzweigtes Array nicht mehrere Dimensionen hat, sondern **verschaltet** ist, d. h. das Array besteht z. B. aus 10 untergeordneten Arrays. Die Anzahl der Elemente dieser 10 ungeordneten Arrays interessiert das ursprüngliche Array daher nicht. Bei der Deklaration werden 2 eckige Klammern-Paare nacheinander notiert. Bei der Initialisierung wird in den eckigen Klammern der 2. Ebene keine Anzahl festgelegt. Dies ist notwendig, da wir bei der Wertezuweisung (im Beispiel durch die Schleifen) den Arrays der 2. Ebene unterschiedliche Größen zuweisen. Bei der Indexierung müssen wir ebenfalls die 2 Klammern-Paare nacheinander notieren.
Nun wollen wir noch eine weitere Schleife kennenlernen: die *foreach*-Schleife. Die **foreach-Schleife** wird eingesetzt, wenn wir auf die Elemente eines Arrays zugreifen wollen. Die *foreach*-Schleife kennzeichnet sich durch das Schlüsselwort *foreach*, gefolgt von runden Klammern. In den runden Klammern deklarieren wir nun eine Variable (ohne Semikolon). Dahinter folgt das Schlüsselwort *in* und dann der Name des Arrays. Über die **„temporäre“ Variable**, welche in der Schleife deklariert wurde, können wir nun den Wert abrufen. Eine Wertzuweisung ist mit der *foreach*-Schleife nicht möglich. Die *foreach*-Schleife kann auch dazu verwendet werden, ein verzweigtes Array „durchzugehen“ (siehe Beispiel).

Program.cs

```
1  int[] aZahlenreihe = new int[10];
2  int[,] aZahlenreihe2Dimensionen = new int[3, 9];        // dieses Array ist zweidimensional
   (3 Spalten, 9 Zeilen)
3  int[][] aZahlenreiheVerzweigt = new int[10][];          // dieses Array ist "verzweigt"
4
5  for (int i = 0; i < aZahlenreihe.Length; i++)
6  {
7      // cast (Typumwandlung) notwendig, da Math.Pow() standardmäßig mit Kommazahlen (double)
   arbeitet
8      aZahlenreihe[i] = (int)Math.Pow(i, 2);
9  }
10
11 // Variable i wird neu deklariert (1. Deklaration in 1. Schleife oben)
12 //  --> dies ist nur möglich, da i nur innerhalb der Schleife verfügbar ist
13 for (int i = 0; i < aZahlenreihe.Length; i++)
14     Console.WriteLine(i + "² = " + aZahlenreihe[i]);
15
```

```
16  Console.WriteLine();
17
18  // Index des angegebenen Wertes ausgeben
19  Console.WriteLine(Array.IndexOf(aZahlenreihe, 9));
20  Console.WriteLine(Array.LastIndexOf(aZahlenreihe, 9));      // gibt das Gleiche wie IndexOf()
    auf, da 9 nur einmal im Array vorhanden ist
21
22  Console.WriteLine();
23
24  // Array umkehren
25  Array.Reverse(aZahlenreihe);
26
27  // Zahlen erneut ausgeben
28  for (int i = 0; i < aZahlenreihe.Length; i++)
29      Console.WriteLine("index " + i + ": " + aZahlenreihe[i]);
30
31  Console.WriteLine();
32
33  // Werte des zweidimensionalen Arrray füllen
34  for (int i = 0; i < aZahlenreihe2Dimensionen.GetLength(0); i++)
35      for (int j = 0; j < aZahlenreihe2Dimensionen.GetLength(1); j++)
36          aZahlenreihe2Dimensionen[i, j] = (i * 10) + j;
37
38  // Werte des zweidimensionalen Array ausgeben
39  for (int i = 0; i < aZahlenreihe2Dimensionen.GetLength(0); i++)
40      for (int j = 0; j < aZahlenreihe2Dimensionen.GetLength(1); j++)
41          Console.WriteLine("array[{0},{1}] = {2}", i, j, aZahlenreihe2Dimensionen[i, j]);
42
43  Console.WriteLine();
44
45  // Werte des verzweigten Arrays füllen
46  for (int i = 0; i < aZahlenreiheVerzweigt.Length; i++)
47  {
48      aZahlenreiheVerzweigt[i] = new int[i * 2];  // die Länge der untergeordneten Array kann
    sich unterscheiden
49      for (int j = 0; j < aZahlenreiheVerzweigt[i].Length; j++)
50          aZahlenreiheVerzweigt[i][j] = (i * 10) + j;
51  }
52
53  // Werte des verzweigten Arrays ausgeben (mit foreach)
54  foreach (int[] aZahlen in aZahlenreiheVerzweigt)
55  {
56      foreach (int iZahl in aZahlen)
57          Console.Write(iZahl + " ");
58      Console.WriteLine();
59  }
60
61  Console.ReadKey();
```

Das große Computer ABC

C# lernen

Grundlagen: Funktionen

Eine Funktion oder auch Methode oder Routine ist ein Teil einer Klasse, um Programmcode **auszulagern**. Oft handelt es sich hierbei auch um Code, der **öfters gebraucht** wird. Dies ist jedoch nicht zwingend notwendig. Bisher haben wir den kompletten Programmcode in der Main-Methode notiert, welche den **Haupteinstiegspunkt** einer Applikation darstellt. Eine Funktion bildet immer einen Block mit geschweiften Klammern, jedoch können die Klammern hier nicht weggelassen werden. Die Zeile über den geschweiften Klammern wird auch als **Funktions-Kopf** bezeichnet: Hier werden Zugriffsmodifizierer, Rückgabetyp, Name und Parameter angegeben.
Der **Zugriffsmodifizierer** ist ein Schlüsselwort, welches angibt, von wo aus der Zugriff auf die Funktion erlaubt ist. Durch den Zugriffsmodifizierer *private* grenzen wir den Zugriff auf die eigene Klasse ein. Mit *protected* grenzen wir den Zugriff auf die eigene und die davon abgeleitete Klasse ein. *public* erlaubt einen Zugriff von außerhalb der Klasse. Zugriffsmodifizierer können auch für **globale Variablen**, also Membervariablen, verwendet werden. Wird der Zugriffsmodifizierer weggelassen, handelt es sich standardmäßig um *private*.
Der **Rückgabetyp** kann jeder Datentyp sein. Jedoch kann eine Funktion immer nur **einen Wert** zurückgeben. Falls eine Funktion **keinen Wert** zurückgeben soll, müssen wir das Schlüsselwort *void* angeben. Die Rückgabe eines Wertes erfolgt mit dem Schlüsselwort *return* und dem Wert. Das Schlüsselwort *return* (ohne Wert) wird auch dazu eingesetzt, um eine Funktion ohne Rückgabetyp frühzeitig zu verlassen.
Der **Name** einer Funktion darf **nur einmal** innerhalb einer Klasse existieren. Eine Ausnahme bildet die Methodenüberladung. Dazu nachher mehr.
Die **Parameter** einer Funktion können ebenfalls jeden Datentyp haben und werden nach dem Funktions-Namen in runden Klammern notiert. Die Anzahl der Parameter ist unbegrenzt, wobei die einzelnen Parameter mit **Kommas** getrennt werden. Es ist auch möglich, dass eine Funktion **keine Parameter** haben. In diesem Fall werden die runden Klammern leer gelassen. Als Parameter können ebenfalls auch Arrays übergeben werden. Ein Parameter wird im Funktions-Kopf mit dem **Datentyp und dem Namen** angegeben. Der Name des Parameters kann dann innerhalb der Funktion verwendet werden.
Bei der **Methodenüberladung** können wir mehrere Funktionen mit demselben Namen erstellen, welche sich durch die Parameter unterscheiden müssen. Eine **Unterscheidung** der Parameter erfolgt an Hand **der Datentypen oder der Anzahl**. Eine Unterscheidung durch den Rückgabetyp ist nicht möglich. Als Beispiel für eine Methodenüberladung könnten wir eine Additions-Funktion nennen, welche einmal mit 2 Zahlen und einmal mit 3 Zahlen arbeitet. Dies wäre eine Unterscheidung an Hand der Parameter-Typen. Eine andere Möglichkeit wäre, eine Funktion zur Addition, welche einmal int-Typen erwartet und ein anderes Mal *double*-Typen. Dies wäre eine Unterscheidung durch die Datentypen der Parameter. Grundsätzlich könnten zwei verschiedene Funktionen der Methodenüberladung komplett andere Anweisungen durchführen, da beide Funktionen einen eigenen Code-Block besitzen. Normalerweise ist dies jedoch nicht geläufig, da dadurch die Struktur des Programms „zerstört“ wird.
Das Schlüsselwort *static* kann bei Funktionen oder Variablen eingesetzt werden, um eine Funktion oder Variable ohne Objekt-Bindung zu erstellen, d. h. die Funktion oder Variable existiert **unabhängig von einem Objektverweis**. Wir können also auf die Funktion oder Variable zugreifen, ohne ein Objekt der Klasse erstellt zu haben. Der Zugriff erfolgt dabei direkt über den Klassennamen. Wie Ihnen vielleicht schon aufgefallen ist, ist die Main-Routine mit diesem Schlüsselwort versehen. Da wir im Beispiel alle Routinen direkt über die Main-Funktion aufrufen und kein Objekt der Klasse Programm erstellt haben, benötigen wir auch bei allen Beispiel-Funktionen das Schlüsselwort static. Dies kommt daher, dass statische Funktionen keine nicht-statische Funktionen aufrufen können (da kein Objektverweis besteht). Eingeordnet wird das Schlüsselwort zwischen Zugriffsmodifizierer und Rückgabetyp.

Program.cs

```
1  using System;
2
3  namespace CSV20.Funktionen
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Sie haben \"{0}\" eingegeben!", TestEingabe());
10
11             Console.WriteLine();
12
13             Console.WriteLine("Rechnung mit 2 Zahlen: 1 + 2     = {0}", Addition(1, 2));
14             Console.WriteLine("Rechnung mit 3 Zahlen: 1 + 2 + 3 = {0}", Addition(1, 2, 3));
15
16             Console.ReadKey();
17         }
18
19         private static string TestEingabe()
20         {
21             Console.Write("Bitte etwas eingeben: ");
```

```
22
23              return Console.ReadLine();
24          }
25
26          private static int Addition(int iZahl1, int iZahl2)
27          {
28              return iZahl1 + iZahl2;
29          }
30
31          // Funktion Addition() wird überladen, wir können also die Funktion mit 2 oder 3
    Parametern aufrufen
32          // Methoden-Überladungen müssen sich in den Parametern unterscheiden (Anzahl oder Typ)
33          //  eine Unterscheidung durch den Rückgabetyp ist nicht möglich
34          private static int Addition(int iZahl1, int iZahl2, int iZahl3)
35          {
36              return iZahl1 + iZahl2 + iZahl3;
37          }
38      }
39  }
```

Das große Computer ABC

C# lernen

Objektorientierung: Klassen und Objekte

Eine **Klasse** stellt einen **Bauplan** dar, welche Variablen, Eigenschaften (dazu später mehr) und Funktionen enthalten. Eine Klasse kann also als Ansammlung von Variablen, Eigenschaften und Funktionen bezeichnet werden. Klassen werden dazu eingesetzt, einen zusammenhängenden Programm-Teil zu kapseln. Die Klasse befindet sich innerhalb des namespace-Blocks. Auch die Klasse selbst bildet einen Block, welche sich durch das Schlüsselwort *class* kennzeichnet.
Von **Objekten** spricht man, wenn eine Variable dieser Klasse **initialisiert** wird. Um eine Variable einer Klasse zu initialisieren, benötigen wir das Schlüsselwort *new*, den Klassennamen und runde Klammern (direkt hinter dem Klassennamen, also wie bei einer Funktion). Die Anzahl der Objektinitialisierungen unserer Klasse ist unbegrenzt, somit können wir mehrere Objekt erzeugen, benötigen jedoch immer nur eine Klasse.
In der „objektorientierten Programmierung“ (kurz OOP) gibt es einige **stilistische Regeln**. Hierzu zählt z. B. dass globale Variablen nicht von außen zugänglich gemacht werden sollten. Um Werte einer internen Variable auszulesen oder zu setzen, wird daher entweder eine Funktion geschrieben oder Eigenschaften verwendet. Der Vorteil von Funktionen und Eigenschaften ist, dass hier ein Programmcode hinterlegt werden kann, mit dem z. B. der Wertebereich (bei Zahlen) überprüft werden kann.
Hierzu ein Beispiel: Wir haben einen Bauplan (Klasse) für ein Auto. Von diesem Auto können wir jetzt eine unbegrenzte Anzahl bauen, die jederzeit erweitert werden kann. Die einzelnen Autos stellen also die Objekte dar.
In unserem Programm-Beispiel erstellen wir eine Klasse namens *Computer*. Diese besitzt die (privaten, also von außerhalb unzugänglichen) Variablen *bHdmiVorhanden* und *iFestplattenGröße*. Des Weiteren verfügt die Klasse über die Funktionen *SetzeHDMI()*, *SetzeFestplattenGröße()* und *ZeigeInfo()*.
In einigen weiteren Beispielen werden Sie sehen, dass hier das Schlüsselwort *this* verwendet wird. *this* greift auf das eigene Objekt zu. So können wir z. B. eine Unterscheidung herstellen, wenn eine globale Variable der Klasse und der Name eines Parameters einer Funktion den gleichen Namen haben.

Program.cs

```
using System;

namespace CSV20.Klassen_Objekte
{
    class Program
    {
        static void Main(string[] args)
        {
            Computer oComputer1, oComputer2;

            Console.WriteLine("Computer 1:");
            oComputer1 = new Computer();
            Console.WriteLine("Setze Festplattengröße auf 8 TB");
            if (!oComputer1.SetzeFestplattenGröße(8192))   // schlägt fehl (max. 4 TB)
                Console.WriteLine("Festplattengröße konnte nicht gesetzt werden");
            oComputer1.ZeigeInfo();
            Console.WriteLine("Setze Festplattengröße auf 512 GB");
            if (!oComputer1.SetzeFestplattenGröße(512))
                Console.WriteLine("Festplattengröße konnte nicht gesetzt werden");
            oComputer1.ZeigeInfo();

            Console.WriteLine();

            Console.WriteLine("Computer 2:");
            oComputer2 = new Computer();
            oComputer2.SetzeFestplattenGröße(1024);
            oComputer2.SetzeHDMI(true);
            oComputer2.ZeigeInfo();

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

Computer.cs

```
using System;

namespace CSV20.Klassen_Objekte
{
    public class Computer
```

```
    {
        private bool bHdmiVorhanden = false;        // standardmäßig ist kein HDMI-Anschluss
vorhanden
        private int iFestplattenGröße = 256;        // in Gigabyte, Standard-Wert ist 256 GB

        public void SetzeHDMI(bool bHatHDMI)
        {
            // bHdmiVorhanden könnte auch auf public gesetzt werden, dadurch ist
            // die Objektorientierung jedoch nicht gegeben
            bHdmiVorhanden = true;
        }

        public bool SetzeFestplattenGröße(int iGröße)
        {
            // nur Größen von 32 GB bis 4 TB erlaubt
            if (iGröße >= 32 && iGröße <= 4096)
            {
                iFestplattenGröße = iGröße;
                return true;
            }
            else
                return false;
        }

        public void ZeigeInfo()
        {
            Console.WriteLine("Computer-Info:");
            Console.WriteLine(" > Die Festplatte ist {0} GB groß", iFestplattenGröße);
            if (bHdmiVorhanden)
                Console.WriteLine(" > HDMI-Anschluss ist vorhanden");
        }
    }
}
```

Das große Computer ABC

C# lernen

Objektorientierung: Objekteigenschaften

Objekteigenschaften (oder auch nur Eigenschaften) sind „bessere“ Variablen, hinter welchen ein Programmcode hinterlegt werden kann (also wie bei Funktionen). Eigenschaften sehen von der Deklaration **ähnlich wie Variablen** aus, bilden jedoch immer einen Block.
Bei **einfachen Eigenschaften** (ohne Werte-Prüfung oder anderen Programmcode) müssen innerhalb des Blocks die Schlüsselwörter *get* und *set* mit jeweils einem Semikolon am Ende notiert werden. Dadurch entspricht die Eigenschaft eigentlich einer Variablen. Trotzdem sollten Sie immer Eigenschaften anstatt Variablen verwenden, wenn der Wert von „außen“ (also von außerhalb der Klasse) zugänglich sein soll.
Die **komplexeren Eigenschaften** besitzen einen Programmcode, welche zur Überprüfung des Wertes genutzt werden kann. Natürlich könnten auch andere Aktionen ausgeführt werden, wie z. B. der Aufruf einer internen Funktion o. Ä.. Bei den komplexeren Eigenschaften bilden die get- und set-Schlüsselwörter eigene Blöcke. Innerhalb des get-Blocks muss ein Wert durch *return* zurückgegeben werden. Im set-Block dagegen kann der Wert z. B. in einer internen Variablen gespeichert werden. Um auf den „übergebenen“ oder zu setzenden Wert zuzugreifen, nutzen wir das Schlüsselwort *value*. Wie wir also sehen können, ist bei den komplexeren Eigenschaften immer eine **interne Variable** von Nöten, wenn der Wert intern gespeichert werden soll. Der Variablenname besteht dabei oft aus einem Unterstrich und dem Eigenschaftsnamen.
Stellen wir uns einmal vor, was wir mit solchen Eigenschaften alles machen können und wie wir diese adaptieren können. Es ist z. B. möglich, den *set*-Block wegzulassen: Dadurch können wir das Setzen eines Wertes unterbinden. Um das Setzen innerhalb der Klasse zuzulassen, jedoch von außen zu unterbinden, könnten wir auch vor das Schlüsselwort *set* einen Zugriffsmodifizierer platzieren (z. B. *private* oder *protected*).
In der objektorientierten Programmierung ist es übrigens üblich, dass die Eigenschaften nicht mit einem Suffix passend zum Daten-Typ versehen werden.

Program.cs

```
1  using System;
2
3  namespace CSV20.Objekteigenschaften
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Computer oComputer = new Computer();
10
11             oComputer.HdmiVorhanden = true;
12             oComputer.FestplattenGröße = 8192;  // funktioniert nicht > vorherige Größe
   bleibt erhalten
13             oComputer.ZeigeInfo();
14             oComputer.FestplattenGröße = 512;
15             oComputer.ZeigeInfo();
16             oComputer.HdmiVorhanden = false;
17             oComputer.ZeigeInfo();
18
19             Console.ReadKey();
20         }
21     }
22 }
```

Computer.cs

```
1  using System;
2
3  namespace CSV20.Objekteigenschaften
4  {
5      public class Computer
6      {
7          private int iFestplattenGröße = 256;        // interner Wert für die Festplattengröße
8
9          public bool HdmiVorhanden { get; set; }
10         public int FestplattenGröße
11         {
12             get
13             {
14                 return iFestplattenGröße;
15             }
16             set
```

```
17              {
18                  // nur Größen von 32 GB bis 4 TB erlaubt
19                  if (value >= 32 && value <= 4096)
20                      iFestplattenGröße = value;
21              }
22          }
23
24          public void ZeigeInfo()
25          {
26              Console.WriteLine("Computer-Info:");
27              Console.WriteLine(" > Die Festplatte ist {0} GB groß", iFestplattenGröße);
28              if (HdmiVorhanden)
29                  Console.WriteLine(" > HDMI-Anschluss ist vorhanden");
30          }
31      }
32  }
```

Das große Computer ABC

C# lernen

Objektorientierung: Konstruktor und Destruktor

Der **Konstruktor** ist eine **spezielle Funktion**, welche bei der **Initialisierung** (also dem Erstellen) **einer Klasse** aufgerufen wird. Der Name der Konstruktor-Funktion entspricht immer dem Klassennamen. Die Funktion besitzt **keinen Rückgabewert** (auch nicht *void*), da der Konstruktor sozusagen das neu initialisierte Objekt zurückgibt. Wird kein Konstruktor in der Klasse erstellt, gibt es immer einen sogenannten **Standard-Konstruktor**, welcher keine weiteren Aktionen ausführt. Der Standard-Konstruktor besitzt keine Parameter. Jedoch kann dies bei der Implementierung eigener Konstruktoren geändert werden. Des Weiteren ist auch die Überladung des Konstruktors möglich. Wichtig zu wissen ist, dass sobald Konstruktoren in der Klasse deklariert werden, es den Standard-Konstruktor den C# automatisch erstellt, nicht mehr gibt. Der Zugriffsmodifizierer bei Konstruktoren ist normalerweise *public*. Wird als Zugriffsmodifzierer *private* verwendet, ist die Objektinitialisierung dieser Klasse nicht mehr möglich. Der **private Konstruktor** wird deshalb verwendet, um die Objektinitialisierung einer Klasse, welche lediglich statische Funktionen enthält, zu unterbinden.
Der **Destruktor** ist das Gegenteil des Konstruktors, da der Destruktor aufgerufen wird, wenn das **Objekt zerstört / gelöscht** wird. Wie bereits angesprochen, wird ein Objekt (welches auf dem Heap gespeichert ist) automatisch gelöscht, sobald das .NET Framework **keine Referenzierung** mehr findet. Dieses Löschen der Objekte geschieht automatisch durch den **Garbage Collector** (kurz GC), wenn das Programm aktuell keine Vorgänge zu bearbeiten hat oder der Speicherplatz kritisch wird. Des Weiteren ist es auch möglich, über die statische Funktion *Collect()* der Klasse *GC* das Löschen unreferenzierter Objekte manuell auszulösen. Der Name des Destruktors entspricht ebenfalls dem Klassennamen, jedoch mit einem vorangestellten Tilde-Zeichen (~). Beim Destruktor darf des Weiteren kein Zugriffsmodifizierer und Rückgabewert angegeben werden.

Program.cs

```
1  using System;
2
3  namespace CSV20.Konstruktor_Destruktor
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Main: Programm gestartet -> rufe Funktion auf");
10             TestFunktion();
11             Console.WriteLine("Main: Funktion verlassen -> Programm zu Ende");
12
13             Console.ReadKey();
14         }
15
16         private static void TestFunktion()
17         {
18             Computer oComputer;
19
20             Console.WriteLine("Fuktion: Funktion aufgerufen -> Erstelle Objekt");
21             oComputer = new Computer();
22             Console.WriteLine("Fuktion: Objekt erstellt -> ändere Eigenschaften");
23             oComputer.HdmiVorhanden = true;
24             Console.WriteLine("Fuktion: Eigenschaften geändert -> zeige Info");
25             oComputer.ZeigeInfo();
26             Console.WriteLine("Fuktion: Info angezeigt -> zerstöre Objekt (Taste drücken, für
   weiter)");
27             // Referenz entfernen, dadurch wird das Objekt nicht mehr adressiert
28             // und kann unten über den Funktionsaufruf gelöscht werden
29             oComputer = null;
30
31             // Variablen würden beim Verlassen der Funktion gelöscht werden,
32             // Objekte werden jedoch nicht automatisch sofort gelöscht
33             // Um das sofortige Löschen zu veranlassen, kann folgendes aufgerufen werden
34             GC.Collect();
35
36             Console.ReadKey();
37             Console.WriteLine("Fuktion: Objekt zerstört -> verlasse Funktion");
38
39             // Info: die Refernz zu entfernen ist normalerweise nur bei globalen Variablen
   notwendig,
40             // im Beispiel ist es jedoch ebenfalls notwendig, da sonst die Funktion für den
   GC das Objekt nicht löscht
41         }
42     }
```

```
}
```

Computer.cs

```
using System;

namespace CSV20.Konstruktor_Destruktor
{
    public class Computer
    {
        private int iFestplattenGröße;      // Variable wird im Konstruktor initialisiert

        public bool HdmiVorhanden { get; set; }
        public int FestplattenGröße
        {
            get
            {
                return iFestplattenGröße;
            }
            set
            {
                // nur Größen von 32 GB bis 4 TB erlaubt
                if (value >= 32 && value <= 4096)
                    iFestplattenGröße = value;
            }
        }

        public Computer()
        {
            iFestplattenGröße = 256;   // auch "this.iFestplattenGröße = 256;" möglich
            Console.WriteLine("Objekt: Objekt erstellt");
        }

        ~Computer()
        {
            Console.WriteLine("Objekt: Objekt zerstört");
        }

        public void ZeigeInfo()
        {
            Console.WriteLine("Computer-Info:");
            Console.WriteLine(" > Die Festplatte ist {0} GB groß", iFestplattenGröße);
            if (HdmiVorhanden)
                Console.WriteLine(" > HDMI-Anschluss ist vorhanden");
        }
    }
}
```

Das große Computer ABC

C# lernen

Objektorientierung: Statische Klassen

Eine statische Klasse ist eine Klasse, die lediglich **statische Funktionen, Variablen und Eigenschaften** enthalten. Ein Beispiel für eine solche Klasse ist die Klasse *Math*. Durch das Schlüsselwort *static* vor dem Schlüsselwort *class* darf die Klasse keine nicht-statische Funktionen, Variablen und Eigenschaften enthalten. Dies ist hilfreich, um eine Initialisierung der Klasse als Objekt (dies wird auch als **Objekt-Instanziierung** bezeichnet) zu verbieten. Statische Klassen werden gerne verwendet, um Hilfs-Funktionen in einer Klasse zu kapseln.

Program.cs

```
1  using System;
2
3  namespace CSV20.Statische_Klassen
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine(CSHelper.WillkommensNachricht());
10
11             Console.WriteLine();
12
13             Console.WriteLine(CSHelper.Addieren(12, 24));
14             Console.WriteLine(CSHelper.Subtrahiere(78, 31));
15
16             Console.ReadKey();
17         }
18     }
19 }
```

CSHelper.cs

```
1  namespace CSV20.Statische_Klassen
2  {
3      public static class CSHelper
4      {
5          public static string WillkommensNachricht()
6          {
7              return "Hallo und Willkommen!";
8          }
9
10         public static int Addieren(int a, int b)
11         {
12             return a + b;
13         }
14
15         public static int Subtrahiere(int a, int b)
16         {
17             return a - b;
18         }
19     }
20 }
```

Das große Computer ABC

C# lernen

Objektorientierung: Vererbung

Bei der Vererbung werden, wie der Name schon vermuten lässt, Funktionen, Variablen und Eigenschaften von der einen Klasse (Basisklasse) zu einer anderen Klasse (abgeleitete Klasse) vererbt. Die **Basisklasse** (auch Elternklasse oder vererbende Klasse genannt) stellt also Funktionen, Variablen und Eigenschaften zur Verfügung, welche in einer oder mehreren **abgeleiteten Klasse(n)** (auch Kindklasse(n) genannt) zur Verfügung stehen. Grundsätzlich kann jede Klasse als Basisklasse verwendet werden. Bei der abgeleiteten Klasse muss hinter dem Klassennamen ein Doppelpunkt und dahinter die Basisklasse notiert werden. Die abgeleitete Klasse kann auf alle Funktionen, Variablen und Eigenschaften der Basisklasse zugreifen, welche den Zugriffsmodifizierer *protected* oder *public* besitzen.
Nun zu unserem Beispiel: Als Basisklasse verwenden wir die Klasse *Computer*. Diese enthält die Eigenschaften *HdmiVorhanden* und *FestplattenGröße*. Die Klasse *Notebook* ist eine Kindklasse, die von der Klasse *Computer* erbt. Des Weiteren enthält die Klasse *Notebook* eine weitere Eigenschaft namens *BildschirmDiagonale*. Im Konstruktor der Klasse *Notebook* wird die Festplatten-Größe auf 512 GB gesetzt. Hier könnten wir nun auf die Eigenschaft *FestplattenGröße* zugreifen, zu Demonstrations-Zwecken greifen wir jedoch direkt auf die Variable *iFestplattenGröße* zu. Der Zugriff auf die Variable ist von der Kindklasse nur möglich, da der Zugriffsmodifizierer der Variable auf *protected* gesetzt wurde. Hierdurch ist ein Zugriff von der Klasse selbst und der abgeleiteten Klasse möglich.

Program.cs

```
1  using System;
2
3  namespace CSV20.Vererbung
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Notebook oNotebook = new Notebook();
10
11             // oNotebook.iFestplattenGröße = 1024; > nicht möglich da Zugriff nur von der
   Klasse selbst oder der geerbten Klasse möglich ist
12             oNotebook.FestplattenGröße = 1024;
13             // oNotebook.iBildschirmdiagonale = 13; > nicht möglich da Zugriff nur von der
   Klasse selbst erlaubt ist
14             oNotebook.BildschirmDiagonale = 13;
15             Console.WriteLine(oNotebook.BildschirmDiagonale);
16
17             Console.ReadKey();
18         }
19     }
20 }
```

Computer.cs

```
1  namespace CSV20.Vererbung
2  {
3      public class Computer
4      {
5          // auf diese Variable, können auch geerbte Klassen zugreifen
6          protected int iFestplattenGröße = 256;
7
8          public bool HdmiVorhanden { get; set; }
9          public int FestplattenGröße
10         {
11             get
12             {
13                 return iFestplattenGröße;
14             }
15             set
16             {
17                 if (value >= 32 && value <= 4096)
18                     iFestplattenGröße = value;
19             }
20         }
21     }
22 }
```

Notebook.cs

```
namespace CSV20.Vererbung
{
    public class Notebook : Computer
    {
        private int iBildschirmDiagonale = 15;

        public int BildschirmDiagonale
        {
            get
            {
                return iBildschirmDiagonale;
            }
            set
            {
                // Wert validieren
                if (value >= 10 && value <= 17)
                    iBildschirmDiagonale = value;
            }
        }

        public Notebook()
        {
            // Notebooks bekommen standardmäßig eine Größe von 512 GB zugewiesen
            // an dieser Stelle könnte auch "FestplattenGröße = 512;" genutzt werden,
            // jedoch wollen wir hier die Verwendung von protected demonstrieren
            iFestplattenGröße = 512;
        }
    }
}
```

Das große Computer ABC

C# lernen

Objektorientierung: Namensräume

Namensräume werden benutzt, um Klassen und / oder Enumerationen (dazu später mehr) zu **gruppieren**. Namensräume kennzeichnen sich durch das Schlüsselwort *namespace* und den Namen des Namensraums und bilden dabei immer einen Block.
Normalerweise haben Programme immer nur einen Namensraum. Bei sehr großen Projekten, bei denen es viele Klassen gibt, macht es jedoch schon Sinn, mehrere Namensräume zu erstellen. Bei sehr großen Projekten ist es des Weiteren u. U. notwendig, Namensräume zu verschachteln. Eine solche **Verschachtelung** kann auf 2 Arten erzeugt werden: Die 1. Möglichkeit wäre die hierarchische Verschachtelung von namespace-Blöcken. Bei der 2. Möglichkeit werden die einzelnen Namen in einem namespace-Block durch einen Punkt getrennt. Im Beispiel haben wir die 2. Möglichkeit verwendet.
Nachdem wir nun Namensräume erstellen können, müssen wir noch wissen, wie wir von einem anderen Namensraum darauf zugreifen können. Hierfür benötigen wir das Schlüsselwort *using*. Mit *using* können wir einen Namensraum **einbinden**. Bei dieser Einbindung müssen wir den Namen notieren, welcher in der Deklaration des Namensraums angegeben wurde.

Program.cs

```
1  using System;
2  using CSV20.Namensräume.Hilfsklassen;
3
4  namespace CSV20.Namensräume
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             Console.WriteLine(CSHelper.WillkommensNachricht());
11
12             // falls nicht der komplette Namensraum eingebunden wird (also ohne "using
   CSV20.Namensräume.Hilfsklassen;")
13             // könnten wir auch wie folgt auf den Namensraum zugreifen;
14             //  - CSV20.Namensräume.Hilfsklassen.CSHelper.WillkommensNachricht()
15             //  - Namensräume.Hilfsklassen.CSHelper.WillkommensNachricht();
16             //  - Hilfsklassen.CSHelper.WillkommensNachricht();
17
18             Console.ReadKey();
19         }
20     }
21 }
```

CSHelper.cs

```
1  using System;
2
3  namespace CSV20.Namensräume.Hilfsklassen
4  {
5      public class CSHelper
6      {
7          public static string WillkommensNachricht()
8          {
9              return "Hallo und Willkommen!";
10         }
11     }
12 }
```

Das große Computer ABC

C# lernen

Fortgeschrittene Grundlagen: Konvertierungen

Konvertierungen werden benötigt, um den Wert eines Datentyps in einen Wert eines anderen Datentyps **umzuwandeln**. Für den Benutzer handelt es sich dabei oft um den gleichen Wert. Dies ist jedoch in der Programmierung nicht immer der Fall. Ein Beispiel dazu: "15" ist nicht das Gleiche wie 15. Bei dem ersten Wert handelt es sich um den Datentyp *string* und bei dem zweiten Wert um den Datentyp *int*.
Einige Werte können direkt in einen anderen Datentyp konvertiert werden. Hier spricht man auch von **Typumwandlungen**. Hierbei unterscheidet man zwischen der impliziten Typumwandlung und der expliziten Typumwandlung. Die **implizite Typumwandlung** führt der Compiler automatisch aus. Diese Umwandlung erfolgt z. B. bei der Zuweisung einer *int*-Variablen auf eine *long*-Variable. Diese „automatische“ Umwandlung ist möglich, da keine Daten dabei verloren gehen. Die **explizite Typumwandlung** muss durch den Programmierer im Programmcode ausgelöst werden. Hierfür ist der sogenannte **cast** notwendig. Beim cast werden vor dem zuzuweisenden Wert runde Klammern notiert, in welche der Ziel-Datentyp eingetragen wird. Eine solche Konvertierung ist z. B. notwendig, wenn wir den Wert einer double-Variable in eine Ganzzahl-Variable (z. B. vom Datentyp int) zuweisen wollen, da hierbei Informationen verloren gehen (in diesem Fall der Gleitkomma-Anteil).
Wenn wir nun bspw. eine Zeichenkette in eine Zahl umwandeln möchten, reichen die oben beschriebenen Typ-Umwandlungen jedoch nicht aus. Dies liegt daran, da eine Zeichenkette sozusagen eine spezielle Art von Array ist. Des Weiteren entspricht ein einzelnes Zeichen (*char*) nicht der (für den Menschen sichtbaren) Zahl selbst, sondern der dazugehörigen ASCII- / Unicode-Zahl ('0' != 0, sondern '0' == 48). Für solche **Konvertierungen** gibt es die statische Klasse *Convert* (Namensraum *System*). Diese Klasse besitzt einige Funktionen, welche mehrfach überladen sind. Als Beispiel zu nennen sind *ConvertToInt32()* und *ConvertToDouble()*. Natürlich existieren auch Konvertierungs-Funktionen für die anderen Datentypen. Im ersten Moment erscheint der Funktionsname *ConvertToInt32()* etwas verwirrend. Die 32 im Funktionsname entspricht der Anzahl der Bits des dazugehörigen Ganzzahl-Typs (in diesem Fall also *int*). Für *short* gibt es damit also die Funktion *ConvertToInt16()*, für *long* die Funktion *ConvertToInt64()*, für *uint* die Funktion *ConvertToUInt32()* usw. Das „Problem“ mit diesen Funktionen ist, dass eine **Exception** (dazu später mehr) ausgelöst wird, wenn der übergebene Wert ungültig ist. Wird eine solche Exception im Programmcode nicht abgefangen, stürzt das Programm ab.
Eine **elegantere Variante** ist die *TryParse()*-Funktion, die immer dann eingesetzt werden kann, wenn eine Zeichenkette in einen anderen Datentyp umgewandelt werden soll. Die *TryParse()*-Funktion befindet sich in der Klasse des jeweiligen Datentyps (für den Datentyp *int* also *int.TryParse()*). *TryParse()* erwartet als 1. Parameter eine Zeichenkette und als 2. Parameter eine Referenz (*out*-Schlüsselwort, dazu später mehr) auf eine Variable des jeweiligen Datentyps. Die Rückgabe der *TryParse()* ist ein *bool*-Wert. Ist die Konvertierung erfolgreich, wird der konvertierte Wert in der referenzierten Variable (2. Parameter) gespeichert und *true* zurückgegeben. Bei fehlerhafter Konvertierung wird der Wert auf 0 gesetzt und *false* zurückgegeben.

Program.cs

```
1  int iZahl1, iZahl2;
2
3  // falls bei den ersten zwei Eingaben, etwas ungültiges eingegeben wird, wird eine Exception
   ausgelöst
4  Console.Write("Bitte geben Sie die erste Zahl ein: ");
5  iZahl1 = Convert.ToInt32(Console.ReadLine());
6  Console.Write("Bitte geben Sie die zweite Zahl ein: ");
7  iZahl2 = Convert.ToInt32(Console.ReadLine());
8  Console.WriteLine("Das Ergebnis von {0} + {1} ist {2}", iZahl1, iZahl2, iZahl1 + iZahl2);
9
10 Console.WriteLine();
11
12 // falls bei den nächsten zwei Eingaben, etwas ungültiges eingegeben wird, wird eine
   Fehlermeldung ausgegeben
13 Console.Write("Bitte geben Sie die erste Zahl ein: ");
14 if (!int.TryParse(Console.ReadLine(), out iZahl1))
15     Console.WriteLine("Die eingegebene Zahl war ungültig. iZahl1 wurde automatisch auf 0
   gesetzt!");
16 Console.Write("Bitte geben Sie die zweite Zahl ein: ");
17 if (!int.TryParse(Console.ReadLine(), out iZahl2))
18     Console.WriteLine("Die eingegebene Zahl war ungültig. iZahl2 wurde automatisch auf 0
   gesetzt!");
19 Console.WriteLine("Das Ergebnis von {0} + {1} ist {2}", iZahl1, iZahl2, iZahl1 + iZahl2);
20
21 Console.ReadKey();
```

Das große Computer ABC

C# lernen

Fortgeschrittene Grundlagen: Fehlerbehandlung

Die Fehlerbehandlung dient zum **Abfangen von Exceptions**. Solche Exceptions können von aufgerufenen Funktionen ausgelöst werden. Ein häufiger Einsatz solcher Exceptions (oder auch Ausnahmen genannt) findet sich bei **Zugriffen auf Dateien und Streams** (dazu später mehr). Wenn solche Exceptions im Programm nicht abgefangen werden, **stürzt das Programm u. U. komplett ab**. Deshalb wollen wir uns in diesem Thema damit beschäftigen, wie wir solche Exceptions abfangen können.
Grundlegend gilt zu sagen, dass alle Exceptions der Basisklasse *Exception* (Namensraum *System*) angehören. Für verschiedene Anwendungseinsätze werden jedoch von *Exception* **abgeleitete Klassen** verwendet (z. B. *FormatException*, *OverflowException* oder *NullReferenceException*). Der Programmcode, welcher eine Exception auslösen kann, sollte daher mit dem **try-Block** umschlossen werden. Darin können natürlich auch Programmzeilen notiert werden, welche keine Exception auslösen können. Um nun den Fehlerfall zu behandeln, benötigen wir den **catch-Block**. Der Kopf des catch-Blocks bildet sich aus dem Schlüsselwort *catch* und runden Klammern, in welcher wir eine Variable deklarieren (z. B. Exception ex). Dabei entspricht der Datentyp der Variable der Exception-Klasse. Der Variablenname kann wie immer frei gewählt werden. Die Basisklasse *Exception* kann für alle Exceptions verwendet werden und ist vor allem dann hilfreich, wenn es egal ist, welche exakte Form der Exception ausgelöst wurde. Der Code des catch-Blocks wird also ausgeführt, sobald eine Exception im try-Block auftritt. Der restliche Code des try-Blocks (wenn vorhanden) wird nicht mehr ausgeführt. catch-Blöcke können untereinander gestapelt werden. Hierbei ist die Reihenfolge entscheidend: Wird im ersten catch-Block der Typ *Exception* abgefangen und im zweiten Block der Typ *NullReferenceException,* so wird das Programm niemals im zweiten Block landen, da die *NullReferenceException* bereits durch den Typ *Exception* abgefangen wird.
Bei Stream-Zugriffen ist es des Weiteren oftmals wichtig, den Stream zu schließen, unabhängig davon, ob eine Exception aufgetreten ist oder nicht. Dafür gibt es den **finally-Block**. Der finally-Block wird unterhalb der catch-Blöcke notiert. Der Code innerhalb des finally-Blocks wird ausgeführt, nachdem der Code im try-Block oder der des catch-Blocks (falls eine Exception ausgelöst wurde) ausgeführt wurde.
Exceptions sind keine Fehler, welche vom Prozessor ausgelöst werden. Im Gegenteil, es sind **Fehler, die im .NET-Framework abgefangen sind**. Genau aus diesem Grund ist es auch möglich, solche Exceptions im C#-Programmcode als Programmierer selbst auszulösen. Hierfür benötigen wir ein Objekt der gewünschten Exception-Klasse. Über das Schlüsselwort *throw* können wir dann eine Exception auslösen. Hinter dem Schlüsselwort *throw* müssen wir dann das erstellte Objekt notieren. Aufgrund der Bezeichnung des Schlüsselworts wird auch oft davon gesprochen, dass eine Exception **„geworfen“** (engl. throw) wird. Natürlich können Sie auch eigene Exceptions erstellen, indem Sie eine Klasse erstellen, welche die Klasse Exception als Basisklasse verwendet. Den Exception-Klassen können im Konstruktor standardmäßig die Fehlermitteilung übergeben werden (Eigenschaft *Message*). Mit der *ToString()*-Methode kann des Weiteren eine detaillierte Fehlermeldung angezeigt werden. Hierfür sollten jedoch noch weitere Eigenschaften gesetzt werden.

Program.cs

```
1  int iZahl;
2
3  try
4  {
5      Console.Write("Bitte geben Sie eine Zahl ein: ");
6      // diese Funktion kann eine FormatException oder eine OverflowException auslösen
7      iZahl = Convert.ToInt32(Console.ReadLine());
8  }
9  catch (FormatException fex)
10 {
11     Console.WriteLine("Ein Formatierungs-Fehler ist aufgetreten:");
12     Console.WriteLine(fex.ToString());
13 }
14 catch (OverflowException oex)
15 {
16     Console.WriteLine("Ein Überlauf-Fehler ist aufgetreten:");
17     Console.WriteLine(oex.ToString());
18 }
19 finally
20 {
21     Console.WriteLine("Falls wir Stream-Zugriffe o. Ä. im try-Block vornehmen können wir");
22     Console.WriteLine("hier unabhängig ob der Vorgang erfolgreich war oder nicht,");
23     Console.WriteLine("Ressourcen wieder freigeben (z. B. Streams schließen)");
24 }
25
26 Console.WriteLine();
27
28 try
29 {
```

```
30      Console.WriteLine("Wir lösen gleich einen Fehler aus!");
31      // das Auslösen von Exception's wird hauptsächlich in größeren Projekten verwendet,
32      // kann jedoch in einigen Fällen sehr hilfreich sein
33      throw new Exception("Ein Fehler wurde mit Absicht ausgelöst!");
34  }
35  // Exception ist eine Basisklasse und kann daher für eine alle Fehlerbehandlungen verwendet
    werden
36  catch (Exception ex)
37  {
38      Console.WriteLine("Ein Fehler ist aufgetreten!");
39      Console.WriteLine("Die übergebene Fehler-Meldung ist: " + ex.Message);
40  }
41
42  Console.ReadKey();
```

Das große Computer ABC

C# lernen

Fortgeschrittene Grundlagen: Enumeration

Enumerationen (auch **Enumerationstypen**) sind ein spezieller Datentyp, mit welchem es möglich ist, einem **konstanten Text einen numerischen Wert zuzuweisen**. Dies soll vor allem dem Programmierer eine **Hilfestellung** sein. Des Weiteren macht es einen Programmcode **übersichtlicher**.
Ein Beispiel dazu: Wir benötigen in unserem Programm die Wochentage. Diesen Wochentagen liegen intern die Werte 0 für Sonntag, 1 für Montag, … und 6 für Samstag zu Grunde. Wenn wir nun intern z. B. den Wert 4 haben, müssen wir als Programmierer erst überlegen, um welchen Tag es sich handelt. Wollen wir eine Abfrage machen, ob der Wert dem Freitag entspricht, müssten wir also if (iTag == 5) notieren. Dies ist weder übersichtlich noch guter Programmierstil. In einem solchen Fall könnten wir eine Enumeration erstellen.
Um eine eigene Enumeration zu verwenden, müssen wir diese erst definieren. Dazu benötigen wir das Schlüsselwort *enum* gefolgt von dem Namen, welchen wir der Enumeration geben wollen. Vor dem Schlüsselwort kann des Weiteren ein Zugriffsmodifzierer angegeben werden. Wird dieser nicht angegeben, wird *private* als Sicherheitsstufe verwendet. Die Definition einer Enumeration bildet immer einen **Block**. Innerhalb des Blocks werden die einzelnen Werte (konstante Texte) angegeben und **durch Komma getrennt**. Den numerischen Wert müssen wir nicht zwangsläufig angeben. Werden keine Werte angegeben, entspricht der **erste Wert der Zahl 0**. Die weiteren Elemente werden **durchgezählt**. Über das Gleichheitszeichen lassen sich die **Werte ändern**. Wird ein Wert nicht angegeben, setzt die normale Zählung (vorheriger Wert + 1) wieder in Kraft.
Sobald eine Enumeration definiert wurde, können wir diese als Datentyp einer Variablen verwenden. Die Zuweisung einer Enumerations-Variable erfolgt durch den Namen der Enumeration und den konstanten Text eines einzelnen Elements (getrennt durch einen Punkt). Eine Enumeration besitzt standardmäßig einen Wertebereich des Datentyps *int*. Dieser kann über den Doppelpunkt gefolgt vom Datentyp hinter dem Namen der Enumerations-Definition geändert werden. Der gewählte Datentyp muss jedoch immer ein numerischer Wert sein.
Die statische Klasse *Enum* besitzt einige **Hilfsfunktionen,** um Konvertierungen durchzuführen. Wollen wir z. B. den Namen (konstanten Text) des dazugehörigen numerischen Wertes ermitteln, kann uns die Funktion *GetName()* behilflich sein. Als erster Parameter wird der Typ der Enumeration benötigt. Hierfür benötigen wir das Schlüsselwort *typeof* gefolgt von runden Klammern, in welcher der Datentyp der Ziel-Enumeration angegeben wird. Der zweite Parameter entspricht dem numerischen Wert. Über die Funktion *IsDefined()* können wir ermitteln, ob der angegebene Name (konstante Text) in der Enumeration vorhanden ist. Die Rückgabe der Funktion ist ein Wert des Datentyps *bool*. Über die Funktion *Parse()* können wir den Text in den dazugehörigen numerischen Wert umwandeln. Um ein Enumerations-Element in den zu Grund liegenden, numerischen Wert umzuwandeln, reicht es aus, eine explizite Typumwandlung (cast) durchzuführen.

Program.cs

```
1  using System;
2
3  namespace CSV20.Enumeration
4  {
5      class Program
6      {
7          private enum Monate
8          {
9              Januar = 1,
10             Februar,
11             März,
12             April,
13             Mai,
14             Juni,
15             Juli,
16             August,
17             September,
18             Oktober,
19             November,
20             Dezember
21         }
22
23         private enum Wochentage : byte  // mit ": byte", weisen wir der Enumeration einen
   Wertebereich von 0 - 255 zu
24         {
25             Sonntag,        // Sonntag ist automatisch 0
26             Montag,
27             Dienstag,
28             Mittwoch,
29             Donnerstag,
30             Freitag,
31             Samstag
```

```
32          }
33
34          static void Main(string[] args)
35          {
36              int iMonat;
37              Wochentage eWochentag = Wochentage.Freitag;
38              string sWochentagEingabe;
39
40              Console.Write("Bitte geben Sie eine Monats-Nummer ein: ");
41              if (!int.TryParse(Console.ReadLine(), out iMonat) || iMonat < 1 || iMonat > 12)
42                  Console.WriteLine("Der eingegebene Monat war ungültig!");
43              else
44                  Console.WriteLine(Enum.GetName(typeof(Monate), iMonat));
45
46              Console.Write("Bitte geben Sie einen Wochentag ein: ");
47              sWochentagEingabe = Console.ReadLine();
48              if (!Enum.IsDefined(typeof(Wochentage), sWochentagEingabe))
49                  Console.WriteLine("Der eingegebene Wochentag war ungültig, Freitag wird
    stattdessen verwendet!");
50              else
51                  eWochentag = (Wochentage)Enum.Parse(typeof(Wochentage), sWochentagEingabe);
52              Console.WriteLine("Wert von Wochentage." + eWochentag + ": " + (int)eWochentag);
53
54              Console.ReadKey();
55          }
56      }
57  }
```

Das große Computer ABC

C# lernen

Fortgeschrittene Grundlagen: Listen

Listen sind eine bessere Form von Arrays. Der Vorteil von Listen besteht hauptsächlich darin, dass diese **nach Belieben vergrößert** werden und **einzelne Elemente auch wieder gelöscht** werden können, ohne das Array komplett neu zu erstellen. In C# gibt es die *ArrayList* und die *List*. *ArrayList* ist an keinen bestimmten Typ gebunden, d. h. hier können sowohl *int*-Werte als auch *string*-Werte im selben Array gespeichert werden. Neben einigen Problemen bei der Verarbeitung dieser Werte ist die *ArrayList* des Weiteren ein schlechter Programmierstil, weshalb wir uns nur mit der *List* genauer auseinandersetzen werden.
Die *List* ist an einen **generischen Typ** gebunden, d. h. hier können nur Werte desselben Datentyps gespeichert werden. Der Datentyp der hier verwendet wird, ist jedoch unabhängig und muss lediglich bei der Deklaration angegeben werden, d. h. wir können eine solche Liste auch mit selbst deklarierten Objekten erstellen. Der Programmcode der sich hinter der List-Klasse befindet, existiert deshalb nur einmal, wodurch man von generischen Typen spricht. Wie Ihnen vielleicht schon aufgefallen ist, kann mit der Funktion *Sort()* der Klasse *Array* ebenfalls ein statisches Array jedes Datentyps sortiert werden. Dies liegt daran, dass die *Sort()*-Funktion als Parameter ein Array eines generischen Typs (und somit unabhängig vom Datentyp) erwartet.
Eine Liste ist ein Klasse, weshalb wir das Schlüsselwort *new* benötigen, um ein Objekt davon zu erstellen. Der generische Typ wird in **spitzen Klammern** direkt hinter dem Klassennamen notiert. Mit der Funktion *Add()* können wir der Liste ein neues Element hinzufügen. Das Element wird dabei am Ende der Liste angehängt. Mit *Insert()* können wir ein Element am gewünschten Index einfügen. Die dahinter folgenden Elemente werden dabei um einen Index weiter verschoben. Die Funktion *Remove()* entfernt das gewünschte Element. Falls das Element mehrmals in der Liste vorhanden ist, wird lediglich das erste gefundene Element entfernt. Die Funktionen *IndexOf()* und *LastIndexOf()* dienen zur Suche nach einem Element, bei dem der dazugehörige Index zurückgegeben wird. Bleibt die Suche erfolglos, wird -1 zurückgegeben. Auch für die Sortierung gibt es eine Funktion: Mit *Sort()* erfolgt eine Sortierung an Hand des Standardvergleichs, d. h. bei einzelnen Zeichen bzw. Zeichenketten alphabetisch und bei Zahlen numerisch.

Program.cs

```
1  List<string> lZeichenketten = new List<string>();
2
3  lZeichenketten.Add("C#-Buch");
4  lZeichenketten.Add(".NET Framework");
5  lZeichenketten.Add("Visual Studio");
6
7  // da die Liste Zeichenketten enthält, erfolgt eine alphabetische Sortierung
8  // bei nummerischen Typen würde eine nummerische Sortierung erfolgen
9  // Objekt können nicht ohne weiteres sortiert werden, hierfür wird LINQ benötigt
10 lZeichenketten.Sort();
11
12 Console.WriteLine("Sortierte Liste:");
13 foreach (string sZeichenkette in lZeichenketten)
14     Console.WriteLine(sZeichenkette);
15
16 Console.WriteLine();
17
18 // falls C#-Buch mehrmals vorkommen würde, kann der Index des letzten Vorkommens mit
   LastIndexOf() ermittelt werden
19 Console.WriteLine("Die Zeichenkette \"C#-Buch\" befindet sich am Index {0}.",
   lZeichenketten.IndexOf("C#-Buch"));
20
21 lZeichenketten.Insert(1, "Microsoft");      // einfügen an Index 1 (2. Element)
22 lZeichenketten.Remove("Visual Studio");     // falls "Visual Studio" mehrmals vorkommt, wird
   nur der 1. Eintrag gelöscht
23
24 Console.WriteLine();
25
26 Console.Write("Geben Sie einen Eintrag für die Liste ein: ");
27 lZeichenketten.Add(Console.ReadLine());
28
29 Console.WriteLine("aktuelle Liste:");
30 foreach (string sZeichenkette in lZeichenketten)
31     Console.WriteLine(sZeichenkette);
32
33 Console.ReadKey();
```

Das große Computer ABC

C# lernen

Fortgeschrittene Grundlagen: Referenzen und Verweise

Referenzen und Verweise werden in Funktionen dazu verwendet, einen Übergabeparameter innerhalb einer Funktion **änderbar zu machen**. Hierzu gilt zu sagen, dass alle Datentypen außer Objekte an die Funktion übergeben werden, so dass eine Änderung innerhalb der Funktion **nicht die ursprüngliche Variable ändert**. Da bei Objekten lediglich eine Referenz übergeben wird (da die eigentlichen Daten ja auf dem Heap liegen), wirken sich Änderungen am Übergabeparameter immer auf das ursprüngliche Objekt aus. Um eine solche Änderung der Parameter auch für die anderen Datentypen zu ermöglichen, gibt es die Referenzen und Verweise.
Für Referenzen und Verweise benötigen wir die Schlüsselwörter *ref* und *out*. Eines dieser Schlüsselwörter muss (falls gewünscht) bei der **Funktions-Deklaration** vor dem Datentyp angegeben werden. Des Weiteren muss das Schlüsselwort beim **Funktionsaufruf** vor dem Variablenname angegeben werden. Wir haben eine solche Referenz bereits bei den *TryParse()*-Funktionen für die Konvertierung einer Zeichenkette in eine Zahl verwendet. Referenzen und Verweise werden grundsätzlich gerne dazu verwendet, wenn eine Funktion mehrere Variablen ändern oder zurückgeben muss. Die *TryParse()*-Funktion ist hierbei das beste Beispiel: Wir wollen als Programmierer wissen, ob die Konvertierung erfolgreich war und wollen aber gleichzeitig den konvertierten Wert erhalten. Da eine Funktion jedoch immer nur einen Wert zurückgeben kann, wurde hier eine Referenz eingesetzt. Der Unterschied zwischen *ref* und *out* lässt sich wie folgt erklären: Wird eine Variable mit *out* übergeben, muss die Variable unbedingt in der Funktion initialisiert werden, d. h. die ursprüngliche Variable, darf vor dem Funktionsaufruf uninitialisiert sein. Eine Variable die mit *ref* übergeben wird, muss vor dem Funktionsaufruf initialisiert sein und muss nicht unbedingt in der Funktion zugewiesen werden.
Arrays und Objekte (und somit auch Listen) werden immer als Referenz übergeben, d. h. Änderungen an einem Wert in dem Array oder an einer Objekteigenschaft des Objektes wirken sich auf das Ursprungs-Objekt aus. Der Grund hierfür ist, dass Arrays und Objekte auf dem Heap gespeichert sind. Der Funktion wird also nicht das vollständige Array oder Objekt übergeben, sondern lediglich die Adresse auf dem Heap. Man spricht hierbei von **reference passed by value**. Wird das Array oder Objekt innerhalb der Funktion neu initialisiert / instanziiert, so wirken sich die Änderungen nicht auf das Ursprungs-Objekt aus (da hierdurch die Referenz-Adresse nur lokal geändert wird). Ist dieser Fall jedoch erwünscht (z. B. für die Initialisierung eines Objektes von einer anderen Funktion heraus) so benötigen wir das *ref*- bzw. *out*-Schlüsselwort.

Program.cs

```
1  using System;
2
3  namespace CSV20.Referenzen_Verweise
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              int iZahl = 17;
10             int iZahl1 = 12;
11             int iZahl2 = 54;
12             int iMultiplikator = 3;
13             int iErgebnis;              // unzugewiesene Variable
14
15             Console.WriteLine("iZahl = " + iZahl);
16             MultipliziereZahl(ref iZahl, iMultiplikator);
17             Console.WriteLine("iZahl = " + iZahl);
18
19             Console.WriteLine();
20
21             Console.WriteLine("iErgebnis ist nicht zugewiesen");
22             AddiereZahlen(iZahl1, iZahl2, out iErgebnis);
23             Console.WriteLine("iErgebnis = " + iErgebnis);
24
25             Console.ReadKey();
26         }
27
28         private static void MultipliziereZahl(ref int iZahl, int iMultiplikator)
29         {
30             Console.WriteLine("Multiplikation wird durchgeführt ...");
31             // iZahl muss nicht zwangsläufig verändert werden
32             iZahl = iZahl * iMultiplikator;
33             // mathematische Operationen können auch abgekürzt werden
34             //  > iZahl *= iMultiplikator;
35         }
36
37         private static void AddiereZahlen(int iZahl1, int iZahl2, out int iErgebnis)
38         {
```

```
39              Console.WriteLine("Addition wird durchgeführt ...");
40              // iErgebnis muss immer zugewiesen werden, da die Variable
41              // andernfalls u. U. uninitialisiert wäre, was nicht passieren darf
42              iErgebnis = iZahl1 + iZahl2;
43          }
44      }
45  }
```

Das große Computer ABC

C# lernen

Fortgeschrittene Grundlagen: Ereignisse

Ereignisse oder auch Events genannt, sind eine Möglichkeit, Daten **von einer Klasse zu einer anderen zu übertragen**. Dabei gibt es immer die Klasse, welche das Event besitzt und auslöst (**Herausgeber**) und die Klasse, die das Event registriert hat (**Abonnent**). Ereignisse werden hauptsächlich in Windows-Anwendungen mit grafischer Oberfläche (Windows Forms und WPF) eingesetzt. Jedoch lohnt es sich schon jetzt, die Grundlagen zu lernen. In einer Konsolen-Anwendung gibt es das Event *CancelKeyPress*, welches ausgelöst wird, wenn Ctrl^C bzw. Strg^C gedrückt wird.
Um ein Event zu abonnieren, benötigen wir eine Anweisung, um eine Registrierung durchzuführen und eine Event-Funktion. Hierfür benötigen wir einen Event-Handler und Event-Argumente. Mit dem += Operator können wir ein **Event registrieren**. Hierfür muss ein Objekt der Klasse *EventHandler* dem Event zugewiesen werden. Im Konstruktor wird die Event-Funktion übergeben. Eine **Event-Funktion** besitzt immer zwei Übergabeparameter und keinen Rückgabewert. Die Übergabeparameter sind das Objekt, welches das Event ausgelöst (oftmals wird hier der Variablenname „sender“ verwendet) hat und die Event-Argumente (Klasse *EventArgs*, oftmals wird hier der Variablenname „e“ verwendet). Über die Basisklasse *EventArgs* und *EventHandler* können keine zusätzliche Informationen an die Event-Funktion über die Event-Argumente weitergegeben werden. Deshalb werden, wenn **zusätzliche Informationen** erforderlich sind, von *EventArgs* und *EventHandler* abgeleitete Klassen erstellt und in der Event-Funktion verwendet (z. B. *ConsoleCancelEventArgs* und *ConsoleCancelEventHandler*). Über den -= Operator kann ein Event auch wieder de-registriert werden. Wichtig zu wissen ist, dass **ein Herausgeber mehrere Abonnenten** haben kann.

Program.cs

```
1  using System;
2
3  namespace CSV20.Ereignisse
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.CancelKeyPress += new ConsoleCancelEventHandler(AbbruchKommado);
10
11             Console.WriteLine("Bitte drücken Sie Ctrl^C bzw. Strg^C ...");
12
13             // auf Auslösung des Events warten
14             Console.ReadKey();
15         }
16
17         private static void AbbruchKommado(object sender, ConsoleCancelEventArgs e)
18         {
19             Console.WriteLine("Event von Ctrl^C bzw. Strg^C erhalten!");
20
21             // Konsole offen halten, sodass der User die Nachricht lesen kann
22             Console.ReadKey();
23         }
24     }
25 }
```

Das große Computer ABC

C# lernen

Fortgeschrittene Grundlagen: LINQ

LINQ (Language Integrated Query) ist eine Erweiterung im .NET Framework, mit welchem **spezielle Abfrageausdrücke** erstellt werden können. Das Thema LINQ ist sehr komplex und wird Ihnen bei verschiedenen erweiterten Programmierthemen immer wieder begegnen. Die Erweiterung LINQ befindet sich im Namensraum *System.Linq*.
Hier wollen wir Ihnen ein Beispiel vorstellen, mit welchem eine **generische Liste erweitert sortiert** werden kann. Diese spezielle Sortierung eignet sich hervorragend für die Sortierung von Objekten, da hier eine Sortierung an Hand einzelner Eigenschaften oder Funktionsrückgabewerten sinnvoll ist. Für eine solche Sortierung benötigen wir die Funktionen *OrderBy()* und *OrderByDescending()*. Um eine Liste noch weiter zu sortieren, können wir die Funktionen *ThenBy()* und *ThenByDescending()* verwenden. *OrderByDescending()* und *ThenByDescending()* führt die Sortierung absteigend durch. Als Übergabeparameter müssen wir das sogenannte **Query** übergeben. Hierfür notieren wir einen frei wählbaren Variablennamen, gefolgt von einem **Pfeiloperator** (=>). Hinter dem Pfeiloperator müssen wir nun die Funktion oder Eigenschaft des zu prüfenden (und damit den zu sortierenden) Wert festlegen. Da die Sortierfunktionen ein Objekt des Typs *IOrderedEnumerable* zurückgibt, muss die Funktion *ToList()* aufgerufen werden, um wieder eine generische Liste zu erhalten. Im Gegensatz zu der *Sort()*-Funktion liefert die LINQ-Sortierung eine neue Liste zurück und überschreibt die ursprüngliche Liste nicht.

Program.cs

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

namespace CSV20.LINQ
{
    class Program
    {
        static void Main(string[] args)
        {
            List<Computer> lComputer = new List<Computer>();

            for (int i = 0; i < 5; i++)
                lComputer.Add(new Computer());

            lComputer[0].FestplattenGröße = 2048;
            lComputer[0].HdmiVorhanden = true;
            lComputer[1].FestplattenGröße = 128;
            lComputer[3].FestplattenGröße = 512;
            lComputer[3].HdmiVorhanden = true;
            lComputer[4].FestplattenGröße = 512;

            // OrderBy() bzw. OrderByDescending() gibt eine neue Liste zurück und verändert die Quell-Liste nicht
            // Sort() dagegen sortiert und überschreibt die Quell-Liste
            lComputer = lComputer.OrderByDescending(o => o.FestplattenGröße).ThenByDescending(o => o.HdmiVorhanden).ToList();

            foreach (Computer oComputer in lComputer)
                Console.WriteLine(oComputer.Seriennummer);

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

Computer.cs

```
using System;

namespace CSV20.LINQ
{
    public class Computer
    {
        private static int iSeriennummerLast = 0;   // letzte Seriennummer, diese wird bei jeder Objekterzeugung hochgezählt (wichtig: "static")
        private int iSeriennummer;                  // aktuelle Seriennumer dieses Objektes
        private int iFestplattenGröße = 256;

```

```
11          // Eigenschaft kann nicht verändert werden
12          public int Seriennummer
13          {
14              get
15              {
16                  return iSeriennummer;
17              }
18          }
19          public bool HdmiVorhanden { get; set; }
20          public int FestplattenGröße
21          {
22              get
23              {
24                  return iFestplattenGröße;
25              }
26              set
27              {
28                  if (value >= 32 && value <= 4096)
29                      iFestplattenGröße = value;
30              }
31          }
32
33          public Computer()
34          {
35              // iSeriennummer wird erhöht und dann (mit dem neuen Wert) der Variable
    iSerienummer zugewiesen
36              iSeriennummer = ++iSeriennummerLast;
37              // iSeriennummer = iSeriennummerLast++; würde den alten Wert in der Variable
    iSeriennumer ablegen
38          }
39      }
40  }
```

Das große Computer ABC

C# lernen

Dateien und Streams: Dateizugriffe

Für **einfache Dateizugriffe** können wir die statische Klasse *File* verwenden. Diese befindet sich so wie alle anderen Klassen für Datei- und Ordneroperationen im Namensraum *System.IO*. Um erweiterte Dateizugriffe vornehmen zu können werden wir später die Klassen *StreamWriter* und *StreamReader* kennenlernen.
Um **Dateien zu schreiben**, können wir die Funktionen *WriteAllText()* und *WriteAllLines()* verwenden. Beide Funktionen erwarten zwei Parameter: Zum einen den Dateinamen (als ersten Parameter) und zum andern die Daten, welche geschrieben werden sollen (als zweiten Parameter). *WriteAllText()* schreibt den angegebenen *string* in die Datei. *WriteAllLines()* benötigt dagegen als zweiten Parameter ein Array aus Zeichenketten. Mit der Funktion *AppendAllText()* kann die angegebene Zeichenkette an die **angegebene Datei angehängt** werden. Natürlich gibt es auch Funktionen zum **Lesen von Dateien**: *ReadAllText()* und *ReadAllLines()*. Beide Funktionen erwarten als Parameter den Dateinamen. Die Funktionen geben eine Zeichenkette bzw. ein Array aus Zeichenketten zurück.
Es gibt noch weitere Funktionen, um verschiedene Datei-Aktionen auszuführen. Die Funktion *Copy()* erstellt eine **Kopie einer Datei**. Die Funktion ist überladen und kann mit zwei oder drei Parametern aufgerufen werden. Die ersten zwei Parameter sind der Quell- und Zieldateiname. Der dritte (optionale) Parameter gibt an, ob die Ziel-Datei überschrieben werden soll oder nicht. Bei einem Aufruf mit zwei Parametern wird die Ziel-Datei nicht überschrieben. Natürlich ist es auch möglich, eine **Datei zu verschieben**. Hierfür dient die Funktion *Move()*. Die *Move()*-Funktion besitzt keine Überladungen, wodurch die Ziel-Datei nie überschrieben wird. Das **Löschen einer Datei** ist über die Funktion *Delete()* möglich, welche als Parameter den Dateinamen benötigt. Mit der Funktion *Exists()* kann geprüft werden, ob die angegebene **Datei existiert**. Als Rückgabewert verwendet die Funktion ein *bool*.
Grundlegend gilt: So gut wie alle Datei- und Ordner-Funktionen können Exceptions auslösen, weshalb es sich empfiehlt, einen try-catch-Block einzusetzen. Gängige Exceptions für solche Funktionen sind *IOException*, *DirectoryNotFoundException*, *FileNotFoundException* und *UnauthorizedAccessException*.

Program.cs

```
1  const string sDateinameTest = "test.txt";
2  const string sDateinameTestCopy = "test2.txt";
3  const string sDateinameLog = "logfile.txt";
4
5  string[] sDateiInhaltSchreiben = new string[] {
6      "C#-Buch V 2.0",
7      "",
8      "Kapitel 5 (Dateien und Streams)",
9      "Thema 1 (Dateizugriffe)"
10 };
11 string[] sDateiInhaltLesen;
12
13 try
14 {
15     // Daten in Datei schreiben und danach lesen
16     Console.WriteLine("Test-Datei wird geschrieben ...");
17     File.WriteAllLines(sDateinameTest, sDateiInhaltSchreiben);
18     Console.WriteLine("Test-Datei wurde geschrieben!");
19     Console.WriteLine("Test-Datei wird gelesen ...");
20     sDateiInhaltLesen = File.ReadAllLines(sDateinameTest);
21     if (sDateiInhaltSchreiben.Length == sDateiInhaltLesen.Length)
22     {
23         for (int i = 0; i < sDateiInhaltSchreiben.Length; i++)
24             if (sDateiInhaltSchreiben[i] != sDateiInhaltLesen[i])
25                 Console.WriteLine("In Zeile {0} wurde ein Unterschied gefunden!", i);
26     }
27     else
28         Console.WriteLine("Die Länge des gelesenen und geschrieben Arrays stimmt nicht
   überein!");
29     Console.WriteLine("Test-Datei wurde gelesen!");
30
31     Console.WriteLine();
32
33     // Logfile-Eintrag schreiben
34     Console.WriteLine("Logfile wird geschrieben ...");
35     File.AppendAllText(sDateinameLog, "Das Programm wurde erfolgreich ausgeführt!\r\n");
   // \r\n ist der Zeilenumbruch für Windows
36     Console.WriteLine("Logfile wurde geschrieben!");
37
38     Console.WriteLine();
39
```

```
    Console.WriteLine("Führe Dateiaktionen aus ...");
    File.Copy(sDateinameTest, sDateinameTestCopy, true);    // true = überschreiben erlaubt
    File.Delete(sDateinameTest);
    File.Move(sDateinameTestCopy, sDateinameTest);
    if (!File.Exists(sDateinameTestCopy))                   // Bedingung sollte immer
zutreffen
        File.Create(sDateinameTestCopy);
    else
        Console.WriteLine("Kopie der Test-Datei existiert bereits!");
    Console.WriteLine("Dateiaktionen wurden ausgeführt!");
}
catch (Exception ex)
{
    Console.WriteLine(ex.ToString());
}

Console.ReadKey();
```

Das große Computer ABC

C# lernen

Dateien und Streams: Ordnerzugriffe

Die Funktionen für Ordnerzugriffe befinden sich in der statische Klasse *Directory*. Einige Funktionen für Ordnerzugriffe sind vom Namen und deren Bedeutung gleich oder ähnlich zu Funktionen für Dateizugriffe. Zum **Erstellen eines Verzeichnisses** kann die Funktion *CreateDirectory()* verwendet werden. Die Prüfung ob ein **Verzeichnis existiert,** kann mit der Funktion *Exists()* geprüft werden. Das **Löschen eines Verzeichnisses** erfolgt mit der Funktion *Delete()*. Die Funktion erwartet wie die anderen Funktionen auch nur einen Parameter: den Ordnernamen. Die *Delete()*-Funktion ist überladen und kann auch mit einem weiteren Parameter aufgerufen werden. Hier wird ein *bool*-Wert übergeben, der angibt, ob alle untergeordneten Dateien und Verzeichnisse ebenfalls gelöscht werden sollen. Beim Aufruf mit einem Parameter kann das Verzeichnis nur gelöscht werden, wenn der Ordner leer ist. Über die *Move()*-Funktion kann ein **Verzeichnis verschoben** werden. Dabei spielt es keine Rolle, ob das Verzeichnis leer ist oder nicht.
Ein Programm arbeitet immer im sogenannten **Arbeitsverzeichnis**. Dies ist standardmäßig das Verzeichnis, in welchem die exe-Datei liegt bzw. von wo diese aufgerufen wird. Über die Funktion *GetCurrentDirectory()* und *SetCurrentDirectory()* kann das aktuelle Verzeichnis ausgelesen oder geändert werden.
In C# ist es auch möglich, alle untergeordneten Verzeichnisse oder Dateien eines Verzeichnisses zu ermitteln. Hierfür gibt es die Funktionen *GetDirectories()* und *GetFiles()*. Beide Funktionen erwarten als Parameter das zu prüfende Verzeichnis und geben ein Zeichenketten-Array zurück. Wichtig: Beide Funktionen geben den kompletten Pfad des Verzeichnisses bzw. der Datei zurück, weshalb wir im Beispiel Zeichenketten-Funktionen aufrufen, um lediglich den einzelnen Verzeichnis- bzw. Dateinamen in der Konsole auszugeben.
Für das Kopieren eines Verzeichnisses bietet C# keine eigene Funktion. Eine solche Funktion kann jedoch mit den oben genannten Funktionen wie *CreateDirectory()*, *GetDirectories()* und *GetFiles()* sowie der Funktion *File.Copy()* realisiert werden.

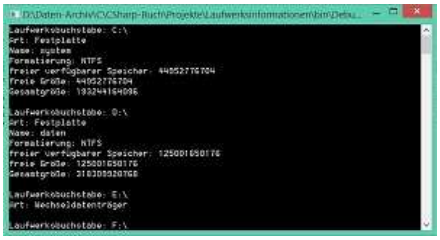
Program.cs

```
1  const string sTestOrdner = "Test";
2  const string sTestOrdner2 = "Test-2";
3  string[] aOrdner;
4  string[] aDateien;
5
6  try
7  {
8      Console.WriteLine("Das aktuelle Arbeitsverzeichnis ist:");
9      Console.WriteLine(Directory.GetCurrentDirectory());
10
11     Console.WriteLine();
12
13     Directory.CreateDirectory(sTestOrdner);
14     Directory.Move(sTestOrdner, sTestOrdner2);
15     Directory.CreateDirectory(sTestOrdner);
16
17     Console.WriteLine("Ordnerliste des Arbeitsverzeichnises:");
18     aOrdner = Directory.GetDirectories(Directory.GetCurrentDirectory());
19     foreach (string sOrdner in aOrdner)
20         Console.WriteLine(sOrdner.Substring(sOrdner.LastIndexOf('\\') + 1));
21
22     Console.WriteLine();
23
24     Console.WriteLine("Dateiliste des Arbeitsverzeichnises:");
25     aDateien = Directory.GetFiles(Directory.GetCurrentDirectory());
26     foreach (string sDatei in aDateien)
27         Console.WriteLine(sDatei.Substring(sDatei.LastIndexOf('\\') + 1));
28
29     // Alle Ordner wieder löschen
30     if (Directory.Exists(sTestOrdner))                  // Bedingung sollte immer zutreffen
31         Directory.Delete(sTestOrdner);
32     if (Directory.Exists(sTestOrdner2))                 // Bedingung sollte immer zutreffen
33         Directory.Delete(sTestOrdner2);
34 }
35 catch (Exception ex)
36 {
37     Console.WriteLine(ex.ToString());
38 }
39
40 Console.ReadKey();
```

Das große Computer ABC

C# lernen

Dateien und Streams: Laufwerksinformationen



Um Informationen über ein Laufwerk bzw. alle Laufwerke eines Computers zu erhalten, gibt es die Klasse *DriveInfo*. Über die statische Funktion *GetDrives()* erhalten Sie ein Array von *DriveInfo*-Objekten. Über einige Objekteigenschaften können nun **Informationen über dieses Laufwerk** abgerufen werden.
Über die Eigenschaft *Name* erhalten Sie den **Laufwerksbuchstaben**. Die Eigenschaft *DriveType* gibt den **Typ des Laufwerks** zurück: also CD-/DVD-Laufwerk, Festplatte, Wechseldatenträger oder Netzlaufwerk. Diese verschiedenen Typen sind in der Enumeration *DriveType* deklariert.
Die Klasse enthält noch einige weitere Eigenschaften, die jedoch nur dann abgerufen werden dürfen, wenn das Laufwerk bereits ist. Diese Eigenschaften werden wir in diesem Abschnitt genauer erläutern. Der Wert, ob das Laufwerk bereit ist, kann über die Eigenschaft *IsReady* abgerufen werden. Die Eigenschaft *VolumeLabel* gibt den **Namen des Laufwerks** zurück. Über die Eigenschaft *AvailableFreeSpace*, *TotalFreeSpace* und *TotalSize* kann die **Größe des Laufwerks** abgerufen werden. Die Eigenschaft *AvailableFreeSpace* gibt den tatsächlich verfügbaren Speicherplatz für den Benutzer zurück, wohingegen *TotalFreeSpace* den verfügbaren Speicherplatz des Laufwerks zurückgibt (Größe des Laufwerks - Größe des belegten Speichers). Bei diesen Werten handelt es sich um den Datentyp *long*. Die *DriveFormat*-Eigenschaft gibt die Formatierung des Laufwerks als Zeichenkette zurück (also FAT32, NTFS, etc.).
Um Informationen eines **einzelnen Laufwerks** abzurufen, kann dem Konstruktor der Klasse *DriveInfo* als Parameter die Zeichenkette eines Laufwerksbuchstaben gefolgt von einem Doppelpunkt übergeben werden.

Program.cs

```
1  // Alle Laufwerke laden
2  DriveInfo[] aDrives = DriveInfo.GetDrives();
3
4  // Alle Laufwerke durcharbeiten
5  foreach (DriveInfo oDrive in aDrives)
6  {
7      Console.WriteLine("Laufwerksbuchstabe: " + oDrive.Name);
8      // Typ des Laufwerkes bestimmen
9      switch (oDrive.DriveType)
10     {
11         case DriveType.CDRom:
12             Console.WriteLine("Art: CD-Laufwerk");
13             break;
14         case DriveType.Fixed:
15             Console.WriteLine("Art: Festplatte");
16             break;
17         case DriveType.Removable:
18             Console.WriteLine("Art: Wechseldatenträger");
19             break;
20         case DriveType.Network:
21             Console.WriteLine("Art: Netzlaufwerk");
22             break;
23         default:
24             Console.WriteLine("Art: Unbekannt");
25             break;
26     }
27     // Einige Informationen dürfen nur abgerufen werden, wenn das Laufwerk bereit ist
28     if (oDrive.IsReady)
29     {
30         Console.WriteLine("Name: " + oDrive.VolumeLabel);
31         Console.WriteLine("Formatierung: " + oDrive.DriveFormat);
32         Console.WriteLine("freier verfügbarer Speicher: " + oDrive.AvailableFreeSpace);
33         Console.WriteLine("freie Größe: " + oDrive.TotalFreeSpace);
34         Console.WriteLine("Gesamtgröße: " + oDrive.TotalSize);
35     }
36
37     Console.WriteLine();
38 }
39
40 Console.ReadKey();
```

Das große Computer ABC

C# lernen

Dateien und Streams: FileStream

Die *FileStream*-Klasse ist eine Klasse, mit welcher eine Datei gelesen oder geschrieben werden kann. Hierbei ist es im Gegensatz zu den einfachen Funktionen der statischen File-Klasse möglich, die Datei in **mehreren Einzelschritten** zu schreiben oder zu lesen, ohne die Datei währenddessen zu schließen. Beim Schreiben und Lesen von einem *FileStream* werden **einzelne Bytes verarbeitet**, d. h. eine Zeichenkette muss erst in ein byte-Array konvertiert werden bevor dieses geschrieben werden kann. Deshalb wird auch von einem Stream gesprochen.
Ein Stream muss als erstes **geöffnet werden**, bevor Daten gelesen oder geschrieben werden können. In C# existieren solche Funktionen zum Öffnen meistens nicht, da das Öffnen bereits beim Erstellen des Objekts erfolgt. Nach dem Dateizugriff ist es notwendig, den **Stream wieder zu schließen**. Hierfür gibt es die Funktion *Close()*. Im Beispiel haben wir einen try-catch-finally-Block eingesetzt, welcher gerne bei solchen Dateizugriffen eingesetzt wird.
In den nächsten Abschnitten wollen wir nun genauer auf die *FileStream*-Klasse selbst eingehen. Der Konstruktor der Klasse ist mehrfach überladen und wird oftmals mit drei Parametern aufgerufen: Hierbei ist der erste Parameter der Dateiname, der zweite Parameter der **Öffnungs-Modus** und der letzte Parameter der **Zugriffs-Modus**. Beim Öffnungs-Modus können wir angeben, ob wir die Datei erstellen (*CreateNew*), erstellen oder überschreiben (*CreateNew*), öffnen (*Open*), öffnen oder erstellen (*OpenOrCreate*) oder Daten anhängen (*Append*) wollen. Hier wird die Enumeration FileMode verwendet. Mit der Enumeration *FileAccess* können wir den Zugriffs-Modus angeben: lesen (*Read*), schreiben (*Write*) oder lesen und schreiben (*ReadWrite*). Die Datei im selben Moment zu lesen und zu schreiben, ist eher untypisch und wird nur selten benötigt.
Um die Datei **schreiben** zu können, gibt es die Funktionen *WriteByte()* und *Write()*. *WriteByte()* schreibt ein einzelnes Byte und erwartet dieses als Parameter. Die *Write()*-Funktion schreibt ein ganzes Byte-Array bzw. einen Teil davon. Die Funktion erwartet als Parameter ein Byte-Array, ein Offset (Index im Byte-Array, ab welchem geschrieben werden soll, zumeist 0) und die Länge (der zu schreibenden Bytes). Zum **Lesen** gibt es die Funktionen *ReadByte()* und *Read()*. Die *ReadByte()*-Funktion erwartet keine Parameter und gibt lediglich ein einzelnes Byte zurück. Die *Read()*-Funktion ist vom Aufbau identisch zur *Write()*-Funktion.
Um innerhalb einer Datei springen zu können, gibt es die Funktion *Seek()*, welche den sogenannten Lese- bzw. Schreibzeiger der Datei setzt. Die Funktion erwartet zwei Parameter: Die Position und die **Positionierungs-Art** (Enumeration *SeekOrigin*). Bei der Angabe der Positionierungs-Art können Sie angeben, von wo aus der Dateizeiger an Hand des ersten Parameters gesetzt werden soll: *Begin* (ab dem Anfang der Datei), *Current* (ab der aktuellen Position) und *End* (ab dem Ende der Datei, hierbei sollte der erste Parameter einen negativen Wert aufweisen).
Am Ende des Dateizugriffs sollte die Datei / der Stream **stets geschlossen werden**. Hierfür ist die Funktion *Close()* zuständig, welche zudem auch den Inhalt einer Datei aus dem internen Datenpuffer schreibt und diesen anschließend leert. Um das Schreiben der Daten manuell auszuführen, können Sie die Funktion *Flush()* aufrufen, welche kein Parameter zur Übergabe benötigt. Meistens ist dieser Aufruf jedoch nicht notwendig.
Im Beispiel wird eine Zeichenkette über die Klasse *Encoding* in ein Byte-Array umgewandelt. Dabei muss die Zeichenkodierung (im Beispiel ASCII) angegeben werden.

Program.cs

```
1  const string sDateiname = "test.txt";
2  const string sInhalt = "Hallo Welt!";
3  FileStream oStream = null;
4
5  try
6  {
7      oStream = new FileStream(sDateiname, FileMode.OpenOrCreate, FileAccess.Write);
8      oStream.Write(Encoding.ASCII.GetBytes(sInhalt), 0, sInhalt.Length);
9
10     Console.WriteLine("Die Datei {0} ist nun {1} Bytes groß!", sDateiname, oStream.Length);
11 }
12 catch (Exception ex)
13 {
14     Console.WriteLine(ex.ToString());
15 }
16 finally
17 {
18     // mit Close() werden die Daten automatisch geschrieben und die Datei geschlossen, wir
   könnten
19     // auch Flush() aufrufen, um die Datei manuell zu schreiben
20     if (oStream != null)
21         oStream.Close();
22 }
23
24 Console.ReadKey();
```

Das große Computer ABC

C# lernen

Dateien und Streams: StreamWriter und StreamReader

In C# gibt es zusätzlich zum *FileStream* noch einige andere Klassen, mit welchem ein Dateizugriff möglich ist. Hier wollen wir nun noch die Klassen *StreamWriter* und *StreamReader* vorstellen, welche in C# sehr gerne eingesetzt werden. Dabei kann der *StreamWriter* ausschließlich zum Schreiben verwendet werden, wohingegen der *StreamReader* nur für Lesezugriffe geeignet ist.
Der Konstruktor der *StreamWriter*-Klasse benötigt als Übergabeparameter den **Dateinamen**. Als zweiter Parameter kann ein *bool*-Wert übergeben werden, der angibt, ob **Daten an die Datei angehängt** werden sollen (*true* = anhängen, *false* = überschreiben). Unabhängig vom zweiten Parameter (also egal ob *true*, *false* oder nicht angegeben), wird die Datei immer erstellt, falls diese nicht vorhanden ist. Die Funktionen zum Schreiben sind ähnlich aufgebaut wie die Ein- und Ausgabe-Funktionen der Konsole, d. h. es gibt die *Write()*- und *WriteLine()*-Funktion, welche mehrfach überladen ist. Dadurch fällt der **Zugriff auf den Inhalt der Datei einfacher** als mit dem *FileStream*, da dort erst eine Konvertierung in ein *byte*-Array notwendig ist. Der *StreamWriter* verfügt wie der *FileStream* auch über die Funktionen *Flush()* und *Close()*.
Dem *StreamReader* wird im Konstruktor der Dateiname übergeben. Um Daten aus der Datei zu lesen, gibt es die Funktionen *Peek()*, *Read()*, *ReadLine()* und *ReadToEnd()*. *Peek()* und *Read()* geben den Code eines einzelnen eingelesenen Zeichens zurück (abhängig von der Codierung). Dieser Code kann dann z. B. in ein *char* gecastet werden. Der Unterschied zwischen *Peek()* und *Read()* kann wie folgt erklärt werden: Beim Lesen aus einer Datei wird normalerweise der Lesezeiger um eine Position vorgesetzt. *Peek()* hingegen verändert die **Position des Lesezeigers** jedoch nicht. *ReadLine()* liest bis zu einem Zeilenumbruch und liefert eine Zeichenkette zurück. *ReadToEnd()* gibt ebenfalls eine Zeichenkette zurück, liest jedoch die komplette Datei ein. Da es sich beim *StreamReader,* wie der Name schon verrät, um einen Stream handelt, muss dieser ebenfalls mit *Close()* geschlossen werden.
Nachdem Sie nun **drei verschiedene Möglichkeiten für Dateizugriffe** kennen, müssen Sie selbst entscheiden, welche Methode am einfachsten für den spezifischen Anwendungsfall Ihres eigenen Programms geeignet ist. Um eine Datei am Stück zu schreiben oder zu lesen, eignet sich die statische Klasse *File* am besten. Für einen etwas erweiterten Text basierenden Dateizugriff eignet sich *StreamWriter* und *StreamReader*. Falls der Dateizugriff komplexer wird (z. B. Dateizeiger setzen) sollte der *FileStream* verwendet werden.

Program.cs

```
1  const string sDateiname = "test.txt";
2  StreamWriter oWrite = null;
3  StreamReader oRead = null;
4
5  try
6  {
7      // Daten in Datei schreiben
8      oWrite = new StreamWriter(sDateiname, false);  // false = kein Append (anhängen)
9      oWrite.WriteLine("Hallo-Welt!");
10     oWrite.WriteLine();
11     oWrite.Write("C#-Buch V2.0 5.5");       // Write() erzeugt keinen Zeilenumbruch am Ende
12     oWrite.Close();
13 }
14 catch (Exception ex)
15 {
16     Console.WriteLine(ex.ToString());
17     Console.ReadKey();
18     return;
19 }
20 finally
21 {
22     // Daten schreiben und Stream schließen
23     if (oWrite != null)
24         oWrite.Close();
25 }
26
27 try
28 {
29     // Datei auslesen und auf der Konsole ausgeben
30     oRead = new StreamReader(sDateiname);
31     while (oRead.Peek() != -1)
32         Console.WriteLine(oRead.ReadLine());
33 }
34 catch (Exception ex)
35 {
36     Console.WriteLine(ex.ToString());
37 }
38 finally
```

```
39  {
40      // Stream schließen
41      if (oRead != null)
42          oRead.Close();
43  }
44
45  Console.ReadKey();
```

Das große Computer ABC

C# lernen

XML-Verarbeitung: Grundlagen zu XML

XML (Abkürzung für Extensible Markup Language) ist eine sogenannte **Auszeichnungssprache**, welche Daten **strukturiert darstellen** soll. XML-Dokumente können von Menschen ohne weiteres gelesen werden, da diese meist aus ASCII-Zeichen bestehen. Viele moderne Auszeichnungssprachen nutzen als Basis XML. So handelt es sich bei XHTML, SVG, RSS und vielen mehr um XML-Sprachen, welche über sogenannte **Dokumenttypdefinitionen** (kurz DTD) oder **XML Schemen** (kurz XSD) eine Regel zugewiesen wurde. Bei solchen Regeln wird angegeben, welche Elementnamen, Attribute und deren Werte erlaubt sind.

Wenn wir später zum Thema WPF kommen, werden Sie mit der XML-Sprache XAML arbeiten, um das Layout Ihres Formulars zu gestalten. Einige Programme nutzen heutzutage XML, um Informationen in einer Datei / einem Dokument zu speichern (z. B. Eagle von CadSoft). Gerne verwenden auch Hobby-Programmierer ein XML-Format, um Daten abzuspeichern. Dies kommt nicht nur daher, dass das Format gut **leserlich** und **übersichtlich** ist, sondern dass C# Schnittstellen zur einfachen Verarbeitung von XML-Dokumenten anbietet. Diese Schnittstellen werden wir in diesem Kapitel kennenlernen.

Meistens enthält ein XML-Dokument in der ersten Zeile eine XML-Deklaration. Unterhalb davon folgen nun sogenannte Elemente. **Elemente** werden in spitzen Klammern (< und >) notiert. Diese Elemente können beliebig verschaltet werden, sofern es sich um ein zweiteiliges Element handelt. Einteilige Elemente werden am Ende mit /> gekennzeichnet. Bei zweiteiligen Elementen gibt es immer einen öffnenden Tag und einen schließenden Tag. Dabei müssen Elemente in der umgekehrten Reihenfolge geschlossen werden, im Vergleich zu, wie diese geöffnet wurden. Die Elementnamen dürfen, sofern kein DTD oder XSD vorhanden, frei gewählt werden. Ein Element kann neben weiteren untergeordneten Elementen auch einen Text enthalten. In XML können im Start-Tag zusätzlich sogenannte **Attribute** notiert werden. Dabei werden hinter dem Attributname ein Gleichheitszeichen und anschließend der dazugehörige Wert in doppelten Anführungszeichen notiert. Auch die Attributnamen und dessen Werte sind, sofern kein DTD oder XSD vorhanden, frei wählbar. Das erste Element im XML-Dokument wird als Wurzelelement bezeichnet. Es kann immer nur ein Wurzelelement geben. Alle weiteren Verschachtelungen finden ausschließlich in den davon untergeordneten Elementen statt.

Eine ausführliche Erklärung zu XML und deren basierenden Sprachen finden Sie auf unserer Partnerwebseite unter http://www.homepage-webhilfe.de/XML/.

Das große Computer ABC

C# lernen

XML-Verarbeitung: Dateien lesen

Um eine XML-Datei einzulesen und zu verarbeiten, gibt es die Klasse *XmlReader* im Namensraum *System.Xml*. Ein Objekt der Klasse wird normalerweise nicht über den Konstruktor, sondern über die statische Funktion *Create()* erzeugt. Der Funktion wird als Parameter der Dateiname übergeben. Ein *XmlReader* ist ebenfalls ein Stream, weshalb dieser am Ende mit *Close()* geschlossen werden sollte.
Mit der Funktion *Read()* wird ein sogenannter **Knoten** eingelesen. Dabei kann es sich um ein Element, ein Kommentar, einen Text oder anderes handeln. Hierbei arbeitet die *Read()*-Funktion das XML-Dokument an Hand der **Baumstruktur** ab. Nun können über verschiedene Eigenschaften Informationen über den aktuellen Knoten abgerufen werden. Die Eigenschaft *NodeType* gibt einen Wert der Enumeration *XmlNodeType* zurück, welcher angibt, um was für eine Art von Knoten es sich handelt (z. B. *Element*, Comment und *Text*). Die Eigenschaft *Name* gibt den Namen des aktuellen Knotens zurück. Bei Elementen ist dies also der Elementname, bei Attributen der Name des Attributs usw.. Über die Eigenschaft *Value* können wir den aktuellen Wert abrufen, dieser enthält bei Attributen den Attribut-Wert, einen Kommentar oder auch einen Text. Über die Eigenschaft *Depth* können wir die aktuelle Tiefe der Baumstruktur ermitteln. **Attribute** können nicht über die *Read()*-Funktion eingelesen werden, jedoch können wir, wenn wir ein Element eingelesen haben über die Eigenschaft *HasAttributes* ermitteln, ob das Element Attribute enthält. Ist dies der Fall, so können wir diese mit der Funktion *MoveToNextAttribute()* auslesen.

Program.cs

```
1  XmlReader oXmlReader = null;
2
3  try
4  {
5      oXmlReader = XmlReader.Create("test.xml");
6
7      while (oXmlReader.Read())
8      {
9          // Einrückung um 2 Leerzeichen pro Baumstruktur-Tiefe
10         for (int i = 0; i < oXmlReader.Depth; i++)
11             Console.Write("  ");
12
13         // Unterscheiden, anhand des XML-Typs (Element, Attribut etc.)
14         switch (oXmlReader.NodeType)
15         {
16             case XmlNodeType.Element:
17                 Console.WriteLine("Element (Tag) \"{0}\"", oXmlReader.Name);
18                 // Prüfen ob Element Attribute hat
19                 if (oXmlReader.HasAttributes)
20                 {
21                     while (oXmlReader.MoveToNextAttribute())
22                     {
23                         // Einrückung um 2 Leerzeichen pro Baumstruktur-Tiefe
24                         for (int i = 0; i < oXmlReader.Depth; i++)
25                             Console.Write("  ");
26
27                         // Informationen über das Attribut ausgeben
28                         Console.WriteLine("Attribut \"{0}\" = \"{1}\"", oXmlReader.Name,
   oXmlReader.Value);
29                     }
30                 }
31                 break;
32             case XmlNodeType.EndElement:
33                 Console.WriteLine("Element-Ende (Tag) \"{0}\"", oXmlReader.Name);
34                 break;
35             case XmlNodeType.Comment:
36                 Console.WriteLine("Kommentar \"{0}\"", oXmlReader.Value);
37                 break;
38             case XmlNodeType.Text:
39                 Console.WriteLine("Text \"{0}\"", oXmlReader.Value);
40                 break;
41             default:
42                 // Wichtig: Cursor wird um die Anzahl an aufgefüllten Leerzeichen
   zurückgesetzt, wenn keine Ausgabe erfolgen soll
43                 Console.SetCursorPosition(Console.CursorLeft - (oXmlReader.Depth * 2),
   Console.CursorTop);
44                 break;
45         }
46     }
```

```
}
catch (Exception ex)
{
    Console.WriteLine(ex.ToString());
}
finally
{
    if (oXmlReader != null)
        oXmlReader.Close();
}

Console.ReadKey();
```

test.xml

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>

<datenbank>
  <stadt stadtstaat="ja">
    <name>Berlin</name>
    <bundesland>Berlin</bundesland>
    <flaeche>891,85 km²</flaeche>
    <einwohnerzahl>3.543.676</einwohnerzahl>
  </stadt>
  <stadt landeshauptstadt="ja">
    <name>München</name>
    <bundesland>Bayern</bundesland>
    <flaeche>310,43 km²</flaeche>
    <einwohnerzahl>1.378.176</einwohnerzahl>
  </stadt>
  <stadt stadtstaat="ja">
    <name>Hamburg</name>
    <bundesland>Hamburg</bundesland>
    <flaeche>755,26 km²</flaeche>
    <einwohnerzahl>1.813.587</einwohnerzahl>
  </stadt>
  <!-- Frankfurt am Main ist nicht die Landeshauptstadt! -->
  <stadt landeshauptstadt="nein">
    <name>Frankfurt am Main</name>
    <bundesland>Hessen</bundesland>
    <flaeche>248,31 km²</flaeche>
    <einwohnerzahl>691.518</einwohnerzahl>
  </stadt>
  <stadt landeshauptstadt="nein">
    <name>Köln</name>
    <bundesland>Nordrhein-Westfalen</bundesland>
    <flaeche>405,17 km²</flaeche>
    <einwohnerzahl>1.017.155</einwohnerzahl>
  </stadt>
</datenbank>
```

Das große Computer ABC

C# lernen

XML-Verarbeitung: Dateien schreiben

Um eine XML-Datei zu schreiben, benötigen wir die Klasse *XmlWriter*. Auch hier verwenden wir zur Objekterzeugung die statische Funktion *Create()*. Die Funktion ist überladen und wird normalerweise mit ein oder zwei Parametern aufgerufen. Dabei ist der erste Parameter der Dateiname. Der zweite Parameter ist ein Objekt, welches Einstellungen für den Schreib-Vorgang enthält. Hierfür benötigen wir die Klasse *XmlWriterSettings*.
Im Beispiel verwenden wir diese Klasse, um einige **Einstellungen festzulegen**. Mit den Eigenschaften *Indent* und *IndentChars* können wir die Einrückung in der Ziel-Datei steuern. Die Eigenschaft *NewLineChars* legt die Zeichenkette für den Zeilenumbruch fest (zumeist "
"). Wird *Indent* auf *false* gesetzt, werden untergeordnete Elemente nicht eingerückt und befinden sich immer am Zeilenanfang.
Nun aber zu den Funktionen der *XmlWriter*-Klasse. Am Anfang sollten wir die Funktion *WriteStartDocument()* aufrufen. Dadurch wird die XML-Deklaration in das Dokument geschrieben. Der Funktion wird ein *bool*-Wert als Parameter übergeben, welcher besagt, ob das *standalone*-Attribut in der XML-Deklaration auf *yes* oder *no* gesetzt werden soll. Über die Funktionen *WriteStartElement()* und *WriteEndElement()* können wir die Element-Tags in die Datei schreiben. Beiden Funktionen wird der Elementname als Parameter übergeben. Mit Hilfe der Funktion *WriteAttributeString()* können wir ein Attribut in das aktuelle Element schreiben. Hierfür werden der Funktion zwei Zeichenketten als Parameter übergeben: der Attributname und dessen Wert. Über die Funktion *WriteString()* ist es möglich, einen Text in die Datei zu schreiben. Mit Hilfe der Funktion *WriteComment()* können wir einen XML-Kommentar in die Ziel-Datei schreiben. Die Reihenfolge der Funktionsaufrufe erfolgt dabei chronologisch der Baumstruktur nach. Dabei bildet die Klasse das perfekte komplementäre Stück zum *XmlReader*. Wie auch beim *XmlReader* dürfen wir beim *XmlWriter* nicht vergessen, am Ende des Dateizugriffs die Datei zu schließen.

Program.cs

```
1  XmlWriter oXmlWriter = null;
2  XmlWriterSettings oXmlWriterSettings = new XmlWriterSettings();
3
4  try
5  {
6      // Eigenschaften / Einstellungen festlegen
7      oXmlWriterSettings.Indent = true;
8      oXmlWriterSettings.IndentChars = "  ";
9      oXmlWriterSettings.NewLineChars = "\r\n";
10
11     oXmlWriter = XmlWriter.Create("test.xml", oXmlWriterSettings);
12
13     // XML-Validierungs-Kopf
14     oXmlWriter.WriteStartDocument(true);    // true = standalone="yes"
15
16     // Wurzel-Element
17     oXmlWriter.WriteStartElement("liste");
18
19     // Kommentar
20     oXmlWriter.WriteComment("Personen-Liste");
21
22     // 1. Person in XML-Datei schreiben (mit Attributen und Text)
23     oXmlWriter.WriteStartElement("person");
24     oXmlWriter.WriteAttributeString("alter", "42");
25     oXmlWriter.WriteAttributeString("geschlecht", "m");
26     oXmlWriter.WriteString("Max Mustermann");
27     oXmlWriter.WriteEndElement();
28
29     // 2. Person in XML-Datei schreiben (mit Attributen und Text)
30     oXmlWriter.WriteStartElement("person");
31     oXmlWriter.WriteAttributeString("alter", "39");
32     oXmlWriter.WriteAttributeString("geschlecht", "w");
33     oXmlWriter.WriteString("Maria Musterfrau");
34     oXmlWriter.WriteEndElement();
35
36     // Wurzel-Element schließen
37     oXmlWriter.WriteEndElement();
38
39     Console.WriteLine("Die XML-Datei wurde geschrieben!");
40 }
41 catch (Exception ex)
42 {
43     Console.WriteLine(ex.ToString());
```

```
44  }
45  finally
46  {
47      // Daten in Datei schreiben und Stream schließen
48      oXmlWriter.Close();
49  }
50
51  Console.ReadKey();
```

test.xml

```
1  <?xml version="1.0" encoding="utf-8" standalone="yes"?>
2  <liste>
3    <!--Personen-Liste-->
4    <person alter="42" geschlecht="m">Max Mustermann</person>
5    <person alter="39" geschlecht="w">Maria Musterfrau</person>
6  </liste>
```

Das große Computer ABC

C# lernen

XML-Verarbeitung: Navigierung

Um innerhalb eines XML-Dokuments zu navigieren, bietet uns C# eine **XPath-Schnittstelle**. Hierfür benötigen wir die Klassen *XPathDocument* und *XPathNavigator*. Dem Konstruktor der *XPathDocument*-Klasse wird als Parameter der Dateiname übergeben. Über die Funktion *CreateNavigator()* des *XPathDocument*-Objekts erhalten wir ein Objekt der *XPathNavigator*-Klasse.
Um einen XPath-Ausdruck zu verarbeiten, kann die Funktion *Evaluate()* verwendet werden. Diese erwartet eine Zeichenkette als Parameter. Der Rückgabewert ist ein Objekt, welcher in den meisten Fällen mit Hilfe der *ToString()*-Funktion in eine Zeichenkette umgewandelt werden kann, um den gewünschten Wert zu erhalten. Die Funktion *MoveToFirstChild()* navigiert zum ersten Element in der nächsten Ebene. Zu den nächsten Elementen in der gleichen Ebene können wir mit Hilfe der Funktion *MoveToNext()* navigieren. Hier ist schon deutlich erkennbar, dass der XPath-Navigator **durch die einzelnen Ebenen navigiert**. Über die Eigenschaft *NodeType* können wir den Typ des aktuellen Knotens abfragen. Hierfür dient die Enumeration *XPathNodeType*, die mit der Enumeration *XmlNodeType* vergleichbar ist. Auch die Eigenschaften *Name* und *Value* sind so wie in der *XmlReader*-Klasse auch verfügbar. Über die Eigenschaft *HasChildren* können wir abfragen, ob das Element weitere untergeordnete Elemente hat. Für die **Navigation durch Attribute** dienen die Funktionen *MoveToFirstAttribute()* und *MoveToNextAttribute()*. Sowohl *MoveToFirstChild()* und *MoveToNext()* als auch *MoveToFirstAttribute()* und *MoveToNextAttribute()* geben einen boolschen Wert zurück, mit welchem festgestellt werden kann, ob die Navigation möglich war. Wenn wir wieder in die übergeordnete Ebene wechseln wollen, können wir die Funktion *MoveToParent()* verwenden. Das Beispiel wird den Zusammenhang der einzelnen Funktionen und das Konzept der Navigierung etwas verständlicher erklären.

Program.cs

```
1   XPathDocument oDocument;
2   XPathNavigator oNavigator;
3
4   try
5   {
6       oDocument = new XPathDocument("test.xml");
7       oNavigator = oDocument.CreateNavigator();
8
9       // zum (ersten) Wurzel-Element navigieren
10      oNavigator.MoveToFirstChild();
11
12      // versuchen in die untegeordnete Elementen-Ebene zu wechseln
13      if (oNavigator.MoveToFirstChild())
14      {
15          // mit einer Schleife durch alle direkt untergeordneten Elemente (2. Ebene) gehen
16          do
17          {
18              if (oNavigator.NodeType == XPathNodeType.Comment)
19                  Console.WriteLine("Kommentar: " + oNavigator.Value);
20              else if (oNavigator.NodeType == XPathNodeType.Element)
21              {
22                  Console.WriteLine("Element: " + oNavigator.Name);
23                  if (oNavigator.HasChildren)
24                      Console.WriteLine(" -> weitere Unterelemente vorhanden");
25                  // versuchen in die Attribut-Ebene zu wechseln
26                  if (oNavigator.MoveToFirstAttribute())
27                  {
28                      Console.WriteLine(" -> Attribute vorhanden:");
29                      // mit einer Schleife durch alle Attribute des Elementes gehen
30                      do
31                      {
32                          Console.WriteLine("    + {0} = {1}", oNavigator.Name,
    oNavigator.Value);
33                      } while (oNavigator.MoveToNextAttribute());
34                      // wieder zurück in die Elementen-Ebene wechseln
35                      oNavigator.MoveToParent();
36                  }
37              }
38          }
39          while (oNavigator.MoveToNext());
40      }
41
42      // oDocument und oNavigator müssen nicht geschlossen werden
43  }
44  catch (Exception ex)
```

```
{
    Console.WriteLine(ex.ToString());
}

Console.ReadKey();
```

test.xml

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>

<datenbank>
  <stadt stadtstaat="ja">
    <name>Berlin</name>
    <bundesland>Berlin</bundesland>
    <flaeche>891,85 km</flaeche>
    <einwohnerzahl>3.543.676</einwohnerzahl>
  </stadt>
  <stadt landeshauptstadt="ja">
    <name>München</name>
    <bundesland>Bayern</bundesland>
    <flaeche>310,43 km</flaeche>
    <einwohnerzahl>1.378.176</einwohnerzahl>
  </stadt>
  <stadt stadtstaat="ja">
    <name>Hamburg</name>
    <bundesland>Hamburg</bundesland>
    <flaeche>755,26 km¦</flaeche>
    <einwohnerzahl>1.813.587</einwohnerzahl>
  </stadt>
  <!-- Frankfurt am Main ist nicht die Landeshauptstadt! -->
  <stadt landeshauptstadt="nein">
    <name>Frankfurt am Main</name>
    <bundesland>Hessen</bundesland>
    <flaeche>248,31 km</flaeche>
    <einwohnerzahl>691.518</einwohnerzahl>
  </stadt>
  <stadt landeshauptstadt="nein">
    <name>Köln</name>
    <bundesland>Nordrhein-Westfalen</bundesland>
    <flaeche>405,17 km</flaeche>
    <einwohnerzahl>1.017.155</einwohnerzahl>
  </stadt>
</datenbank>
```

Das große Computer ABC

C# lernen

Wichtige C#-Klassen: DateTime

DateTime ist eine spezielle Klasse (bzw. eine sogenannte Struktur) in C#, mit welcher ein Datum und Uhrzeit gespeichert und verwaltet werden kann. Um ein Objekt für **das aktuelle Datum und die aktuelle Uhrzeit** zu erhalten, verwenden wir die statische Eigenschaft *Now*. Die dazugehörige aktuelle UTC-Zeit erhalten Sie durch die Eigenschaft *UtcNow*. Grundsätzlich kann ein solches Datum-Objekt auf verschiedene Arten formatiert werden. Hierfür gibt es bereits vier vordefinierte **Formatierungs-Funktionen**: *ToLongDateString()*, *ToShortDateString()*, *ToLongTimeString()* und *ToShortTimeString()*. Mit der Funktion *ToString()* kann des Weiteren eine **manuelle Formatierung** durchgeführt werden. Hierfür sind verschiedene Kürzel notwendig, welche durch die Anzahl der Zeichen und deren englischen Begriffe abzuleiten sind (z. B. Tag einstellig = d, Tag zweistellig = dd).
Für die einzelnen **Bestandteile eines Datusm und einer Uhrzeit** gibt es verschiedene Eigenschaften wie z. B. *Hour*, *Minute*, *Second*, *Day*, *Month*, *Year* etc.. Um vom Datum bzw. der Uhrzeit Stunden, Minuten, Sekunden, Tage, Monate, Jahre etc. hinzuzuzählen oder auch abzuziehen, gibt es die Funktionen *AddHours()*, *AddMinutes()*, *AddSeconds()*, *AddDays()*, *AddMonths()*, *AddYears()* etc.. Hier können sowohl positive als auch negative Werte angegeben werden.
Wie bei den Datentypen in C# auch, gibt es bei der *DateTime*-Struktur eine *TryParse()*-Funktion, mit welcher die Zeichenkette eines Datum und / oder einer Uhrzeit in ein *DateTime*-Objekt umgewandelt werden kann.

Program.cs

```
1  DateTime sDatumUhrzeit = DateTime.Now;
2
3  // einige vorgefertigte Funktionen zur Formatierung
4  Console.WriteLine(".ToLongDateString(): " + sDatumUhrzeit.ToLongDateString());
5  Console.WriteLine(".ToShortDateString(): " + sDatumUhrzeit.ToShortDateString());
6  Console.WriteLine(".ToLongTimeString(): " + sDatumUhrzeit.ToLongTimeString());
7  Console.WriteLine(".ToShortTimeString(): " + sDatumUhrzeit.ToShortTimeString());
8
9  Console.WriteLine();
10
11 // eigene Formate sind mit Hilfe der ToString()-Funktion anwendbar
12 Console.WriteLine("eigenes Format: " + sDatumUhrzeit.ToString("dddd, dd.MM.yyyy
   HH:mm:ss,fff"));
13
14 Console.WriteLine();
15
16 // einzelne Bestandteile von Datum und Uhrzeit können ebenfalls abgerufen werden
17 Console.WriteLine(".Second = " + sDatumUhrzeit.Second);
18 Console.WriteLine(".Month = " + sDatumUhrzeit.Month);
19 Console.WriteLine(".DayOfYear = " + sDatumUhrzeit.DayOfYear);
20
21 Console.WriteLine();
22
23 // Sommer- / Winterzeit prüfen
24 if (sDatumUhrzeit.IsDaylightSavingTime())
25     Console.WriteLine("Sommerzeit");
26 else
27     Console.WriteLine("Winterzeit");
28
29 Console.WriteLine();
30
31 // Datum und Uhrzeit verändern
32 sDatumUhrzeit = sDatumUhrzeit.AddDays(19);
33 sDatumUhrzeit = sDatumUhrzeit.AddYears(3);
34 sDatumUhrzeit = sDatumUhrzeit.AddHours(-13);
35 sDatumUhrzeit = sDatumUhrzeit.AddSeconds(-278);
36 Console.WriteLine("geänderte Zeit: " + sDatumUhrzeit.ToShortDateString() + " " +
   sDatumUhrzeit.ToLongTimeString());
37
38 Console.WriteLine();
39
40 // Eingabe eines Datums und einer Uhrzeit durch den User
41 Console.Write("Bitte geben Sie Datum und / oder Uhrzeit ein: ");
42 if (DateTime.TryParse(Console.ReadLine(), out sDatumUhrzeit))
43 {
44     Console.WriteLine("eingegebene Zeit: " + sDatumUhrzeit.ToShortDateString() + " " +
   sDatumUhrzeit.ToLongTimeString());
45 }
46 else
```

```
47      Console.WriteLine("Das eingegebene Datum bzw. die eingegebene Uhrzeit war ungültig!");
48
49  Console.ReadKey();
```

Das große Computer ABC

C# lernen

Wichtige C#-Klassen: Process

Mit der Klasse *Process* können Sie ein **fremdes Programm starten** oder **eine Datei öffnen**. Beim Öffnen einer Datei verhält sich die Klasse so, wie wenn Sie die Datei über den Explorer starten würden, d. h. entweder wird das dafür vorgesehene **Standardprogramm** mit der Datei oder der **Dialog für unbekannte Dateien** geöffnet. Die Klasse *Process* befindet sich im Namensraum *System.Diagnostics*.
Über die Eigenschaft *StartInfo* kann das *ProcessStartInfo*-Objekt abgerufen werden, welches **Informationen über die auszuführende Datei** enthält. Über die Eigenschaft *FileName* wird der Dateiname der zu ausführenden Datei angegeben. Hierbei muss auch der Pfad angegeben werden, sofern sich die Datei nicht im aktuellen Arbeitsverzeichnis oder im Windows „system32"-Ordner befindet. Die Eigenschaft *Arguments* legt die Argumente für das Programm fest. Mit Hilfe der Eigenschaft *WindowStyle* kann festgelegt werden, in welchem Stil / Aussehen das Programm geöffnet werden soll: Normal-Modus (*Normal*) minimiert (*Minimized*) oder maximiert (*Maximized*). Diese Werte befinden sich in der Enumeration *ProcessWindowStyle*.
Die Funktion *Start()* der *Process*-Klasse ruft den „Prozess" auf (oder einfach gesagt: Startet die Datei). Mit Hilfe der Eigenschaft *HasExited* kann abgefragt werden, ob das Programm bereits wieder geschlossen wurde. Natürlich können wir den Prozess auch beenden: Hierfür benötigen wir die Funktion *Kill()*. Falls Sie ein Ereignis auslösen wollen sobald das Programm beendet wurde, können Sie ein Ereignis auf das Event *Exited* registrieren.
Bei einfachen Aufrufen zum Starten eines Programms erscheint der oben genannte Weg etwas kompliziert. Deshalb können wir auch die statische Funktion *Start()* der *Process*-Klasse aufrufen, welcher wir als Parameter den Dateinamen und bei Bedarf noch zusätzlich die Argumente übergeben. Die Funktion gibt das erstellte *Process*-Objekt zurück, wodurch eine Steuerung des Prozesses (z. B. Beenden oder prüfen ob die Anwendung noch geöffnet ist) immer noch möglich ist.

Program.cs

```
1  Process oProcess = new Process();
2
3  oProcess.StartInfo.FileName = "cmd.exe";
4  oProcess.StartInfo.WindowStyle = ProcessWindowStyle.Maximized;
5  oProcess.StartInfo.Arguments = "/K ping localhost";
6
7  oProcess.Start();
8
9  Console.WriteLine("Bitte drücken Sie eine Taste um das Programm zu beenden ...   ");
10 Console.ReadKey();
11 if (oProcess.HasExited)
12     Console.WriteLine("Das Programm wurde bereits beendet!");
13 else
14 {
15     oProcess.Kill();
16     Console.WriteLine("Das Programm wurde beendet!");
17 }
18
19 // Kurz-Form möglich (für einfache Start-Vorgänge): Process.Start("cmd.exe", "/K ping
   localhost");
20
21 Console.ReadKey();
```

Das große Computer ABC

C# lernen

Wichtige C#-Klassen: Regex

Regex ist eine Klasse in C# für die Bearbeitung und Prüfung sogenannter **regulärer Ausdrücke**. Die Klasse befindet sich im Namensraum *System.Text.RegularExpressions* und enthält einige statische Funktionen, welche meistens auch genutzt werden, da eine Objektinstanzierung zumeist nicht notwendig ist. Die einzelnen Zeichenliterale, Quantoren, Gruppierungen usw. werden wir hier nicht erklären, da es ein sehr komplexes Thema ist. Aber natürlich wollen wir Ihnen nicht vorenthalten, wie solche reguläre Ausdrücke in C# eingebaut werden können.
Über die statische Funktion *Replace()* können wir mit Hilfe eines **regulären Ausdrucks als Suchmuster** eine **Ersetzung** durchführen. Dabei ist der erste Parameter die Zeichenkette, die bearbeitet werden soll. Der zweite Parameter stellt das Suchmuster dar und der dritte Parameter gibt die zu ersetzende Zeichenkette an. Die Funktion gibt als Rückgabewert die bearbeitete Zeichenkette zurück.
Mit Hilfe der statischen Funktion *IsMatch()* können wir prüfen, ob die im ersten Parameter übergebene Zeichenkette der **Bedingung** des regulären Ausdrucks (zweiter Parameter) zutrifft.
Bei regulären Ausdrücken kommen oft Backslashes vor. Da bei Zeichenketten Backslashes **escapt** (auch maskiert genannt) werden müssen (mit Hilfe eines doppelten Backslashes), ist dies bei regulären Ausdrücken oft unübersichtlich. Um diesem Problem entgegenzugehen, können wir vor der Zeichenkette ein at-Zeichen (Klammeraffe) notieren. Dadurch werden die Backslashes automatisch escapt. Dieser Vorteil wird auch oft bei Pfadangaben verwendet. Natürlich ist die Funktion des at-Zeichens nicht immer von Vorteil, da man dann keine Zeichenkette mehr erstellen kann, welche doppelte Anführungszeichen als Text enthalten. Deshalb wird der Klammeraffe meist nur bei Pfadangaben und regulären Ausdrücken verwendet.

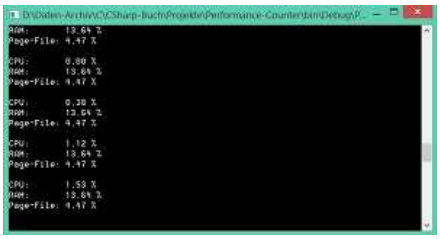
Program.cs

```
1  // Info: durch das @-Zeichen vor einer Zeichenkette ist es nicht notwendig einen Schrägstrich
   zu "escapen" (also statt \\ -> \)
2  Console.WriteLine(Regex.Replace("Hallo Welt  vom C#-Buch Version    2.0 vom   Jahr  2015!",
   @"\s+", " "));
3
4  Console.WriteLine();
5
6  Console.Write("Bitte geben Sie eine IP-Adresse ein: ");
7  if (Regex.IsMatch(Console.ReadLine(), @"^((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.)
   {3}(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9])$"))
8      Console.WriteLine("Die eingegebene IP-Adresse ist gültig!");
9  else
10     Console.WriteLine("Ihre Eingabe entspricht keiner gültigen IP-Adresse.");
11
12 Console.ReadKey();
```

Das große Computer ABC

C# lernen

Wichtige C#-Klassen: PerformanceCounter



Der *PerformanceCounter* ist eine Klasse, mit welcher Informationen über die **aktuelle Auslastung oder Nutzung des Computers** (Leistungsindikator) abgerufen werden können und sich im Namensraum *System.Diagnostics* befindet. Dem Konstruktor kann dabei der Kategorie-Name (*CategoryName*) und der Zähler-Name (*CounterName*) sowie den optionalen Instanz-Namen (*InstanceName*) übergeben werden. Die *InstanceName*-Eigenschaft wird nicht immer benötigt und wird daher oft weggelassen. Über die Funktion *NextValue()* können wir den aktuellen Wert abrufen.
Im Beispiel geben wir die aktuelle Auslastung des Prozessors und die Nutzung des Arbeitsspeichers und der Page-File (Auslagerungsdatei von Windows) aus. Mit der statischen Funktion *Sleep()* der *Thread*-Klasse (Namensraum *System.Threading*) können wir den aktuellen Thread für x Millisekunden blockieren. Dies ist notwendig, um eine übersichtliche Ausgabe zu erhalten. Weitere Funktionen und das Thema Multitasking werden wir später ausführlich behandeln.

Program.cs

```
1  // Übersicht über Konstruktor-Parameter                    CategoryName
   CounterName                InstanceName (nicht immer benötigt)
2  PerformanceCounter oPerformanceCpu = new PerformanceCounter("Processor",    "% Processor
   Time",         "_Total");
3  PerformanceCounter oPerformanceRam = new PerformanceCounter("Memory",       "% Committed
   Bytes In Use");
4  PerformanceCounter oPerformancePf = new PerformanceCounter( "Paging File",  "%
   Usage",                  "_Total");
5
6  while (!Console.KeyAvailable)       // warten bis Taste gedrückt wird
7  {
8      Console.WriteLine("CPU:       {0:F2} %", oPerformanceCpu.NextValue());
9      Console.WriteLine("RAM:       {0:F2} %", oPerformanceRam.NextValue());
10     Console.WriteLine("Page-File: {0:F2} %", oPerformancePf.NextValue());
11     Console.WriteLine();
12     Thread.Sleep(500);              // aktuellen Thread (dazu später mehr) 500 ms pausieren
13 }
```

Das große Computer ABC

C# lernen

Wichtige C#-Klassen: Random

Die Klasse *Random* stellt einen **Zufallsgenerator** zur Verfügung, mit welcher Zufallszahlen generiert werden können. Nachdem wir ein Objekt der Klasse erzeugt haben, können wir mit den Funktionen *Next()*, *NextBytes()* und *NextDouble()* Zufallszahlen generieren. Die Funktion *Next()* generiert einen Zufallswert in einem **gewünschten Bereich**. Dabei ist der erste Parameter die inklusive Untergrenze und der zweite Parameter stellt die exklusive Obergrenze dar. Mit *NextBytes()* können Sie ein ganzes *byte*-**Array** mit Zufallszahlen füllen. Die Zufallszahlen können nicht eingegrenzt werden und liegen auf Grund des Datentyps immer im Bereich von 0 bis 255. Als Parameter der Funktion übergeben wir das *byte*-Array. Die Funktion *NextDouble()* generiert eine **Gleitkommazahl** die größer oder gleich 0,0 ist und kleiner als 1,0.

Program.cs

```
1  Random oZufall = new Random();
2  byte[] aPuffer = new byte[5];
3
4  Console.WriteLine("Zufallszahlen (Ganzzahlen 0 bis 1000):");
5  for (int i = 0; i < 5; i++)
6      Console.WriteLine(oZufall.Next(0, 1001));       // Zahlen von 0 bis 1000 (!)
7
8  Console.WriteLine();
9
10 Console.WriteLine("Array mit byte-Zufallszahlen:");
11 oZufall.NextBytes(aPuffer);
12 foreach (byte bPufferEintrag in aPuffer)
13     Console.WriteLine(bPufferEintrag);
14
15 Console.WriteLine();
16
17 Console.WriteLine("Zufallszahlen (Kommazahl 0.0 bis 1.0):");
18 for (int i = 0; i < 5; i++)
19     Console.WriteLine(oZufall.NextDouble());
20
21 Console.ReadKey();
```

Das große Computer ABC

C# lernen

Wichtige C#-Klassen: Environment

Environment ist eine statische Klasse, die einige Eigenschaften und Funktionen für das Abrufen und Setzen von **Informationen der Computer-Umgebung** enthält. Die Eigenschaft *CommandLine* gibt die **Befehlszeile** (Pfad der Datei, Dateiname und Argumente) des Programms zurück. Die Eigenschaft *CurrentDirectory* gibt das aktuelle **Arbeitsverzeichnis** zurück oder legt dieses fest. Standardmäßig entspricht dies dem Ordner, in welchem die ausführbare Datei liegt. Das Arbeitsverzeichnis kann jedoch durch innere und äußere Einflüsse vor oder während der Laufzeit des Programms geändert werden. Mit Hilfe der Funktion *GetCommandLineArgs()* erhalten Sie die **Befehlszeilenargumente**. Diese sind bereits in der Eigenschaft *CommandLine* enthalten, jedoch bietet die Funktion den Vorteil, dass hier die einzelnen Argumente bereits zerteilt wurden, sodass wir ein *string*-Array erhalten. Das erste Argument entspricht immer dem Pfad mit Dateiname. Alle weiteren Argumente sind übergebene Parameter, welche beim „einfachen“ Aufrufen der ausführbaren Datei nicht vorhanden sind. Dabei entspricht das von der Funktion zurückgegebene Array dem Array, welches als Parameter in der *Main()*-Funktion übergeben wurde. Mit der Eigenschaft *MachineName* können Sie den **Computer-Namen** abrufen. *UserName* gibt den **Namen des aktuellen Benutzers** zurück. Um die aktuelle **Version des Betriebssystems** zu ermitteln, kann die Eigenschaft *OSVersion* abgerufen werden. Die Eigenschaft ist ein Objekt der Klasse *OperatingSystem*. Im Beispiel wird der Inhalt des Objekts automatisch in eine Zeichenkette umgewandelt, welche alle notwendigen Informationen enthält. Über die Eigenschaft *SystemDirectory* erhalten wir den **Pfad des Windows-System-Verzeichnisses** (normalerweise C:Windows\system32). Die Eigenschaft *ProcessorCount* gibt die **Anzahl der logischen Prozessoren** zurück. Die Funktion *Exit()* erlaubt es, das aktuelle Programm mit einem gewissen Code zu beenden. Bei diesem Code handelt es sich um den sogenannten **Exitcode**, welcher als Parameter übergeben wird. Der Exitcode ist eine Ganzzahl und wird unter anderem in der Ereignisprotokollierung von Windows protokolliert. Windows bzw. Microsoft empfiehlt, den Exitcode 0, um ein erfolgreiches Beenden des Programms zu kennzeichnen. Meistens werden negative Werte für Fehlercodes verwendet. Die meisten Eigenschaften der *Environment*-Klasse können nur gelesen werden und nicht gesetzt werden.

Program.cs

```
1  Console.WriteLine("Befehlszeile: " + Environment.CommandLine);
2  Console.WriteLine("Arbeitsverzeichnis: " + Environment.CurrentDirectory);
3
4  Console.WriteLine();
5
6  Console.WriteLine("Argumente:");
7  // Environment.GetCommandLineArgs() entspricht den args-Parametern der Main-Funktion
8  foreach (string sArg in Environment.GetCommandLineArgs())
9      Console.WriteLine(sArg);
10
11 Console.WriteLine();
12
13 Console.WriteLine("Computer-Name: " + Environment.MachineName);
14 Console.WriteLine("Benutzername: " + Environment.UserName);
15
16 Console.WriteLine();
17
18 Console.WriteLine("Betriebssystem-Version: " + Environment.OSVersion);      // OS-Version ist
   eine Klasse, über welche weitere
19                                                                              // Informationen
   einzeln abgerufen werden können
20 Console.WriteLine("Systemverzeichnis: " + Environment.SystemDirectory);
21
22 Console.WriteLine();
23
24 Console.WriteLine("Prozessoren-Kern-Anzahl: " + Environment.ProcessorCount);
25
26 Console.WriteLine();
27 Console.WriteLine();
28
29 Console.Write("Geben Sie \"exit\" ein, um das Programm zu beenden: ");
30 if (Console.ReadLine() == "exit")
31     Environment.Exit(0);
32
33 Console.WriteLine("Sie haben das Programm nicht über Environment.Exit() beendet!");
34
35 Console.ReadKey();
```

Das große Computer ABC

C# lernen

Multithreading: Einführung

Beim **Multithreading** werden mehrere sogenannte **Threads** (auch Ausführungsstrang genannt) innerhalb eines einzelnen **Tasks** (auch Prozess genannt) gleichzeitig abgearbeitet. Dieses Verfahren ist nur bei Mehrkernprozessoren möglich.
Doch trotzdem scheint es so, dass selbst bei Einkernprozessoren verschiedene Aufgaben quasi gleichzeitig erledigt werden. Dies kommt daher, dass die einzelnen Prozesse nur eine bestimmte Zeit laufen dürfen. Läuft diese Zeit ab, so wird der nächste Prozess ausgeführt. Dieser Vorgang läuft im Kreis weiter, wodurch es auch kein Problem ist, wenn ein Prozess beendet wird bzw. ein neuer hinzukommt. Da die Zeiten wie lange ein Prozess laufen darf extrem klein sind, entsteht das Bild, als würden die Aufgaben gleichzeitig ausgeführt werden. Bei diesem Verfahren spricht man von **Multitasking**.
Das Multithreading-Verfahren ist wie bereits oben angesprochen nur möglich, wenn mehrere Prozessorkerne vorhanden sind. Diese Regel gilt jedoch nur für das **hardwareseitige Multithreading**. Beim **softwareseitigen Multithreading** genügt es, wenn nur ein Prozessorkern vorhanden ist, da hier die Aufteilung der Threads softwareseitig passiert. Da jedoch nur ein Prozesskern vorhanden ist, ist eine echtes gleichzeitiges abarbeiten nicht mehr möglich. Das softwareseitige Multithreading ist also von der Arbeitsweise vergleichbar mit dem Multitasking. Der Unterschied kann wie folgt erklärt werden: Beim Multithreading werden Threads aufgeteilt, wohingegen beim Multitasking Tasks (vereinfacht gesagt also mehrere Programme) aufgeteilt werden. Der Vorteil bei der Aufteilung von Threads ist, dass sich die Threads innerhalb eines Tasks befinden und somit eine **Kommunikation zwischen den Threads** möglich ist.
In C# ist es möglich, Threads zu erstellen, dadurch können wir also unseren Prozess unterteilen. In Windows-Forms-Anwendungen wird hier gerne die Klasse *BackgroundWorker* verwendet, auf welchen wir später genauer eingehen werden. In diesem Kapitel wollen wir uns mit der Klasse *Thread* und dem Delegat *ThreadStart* sowie dem Delegat *AsynCallback* und dem Interface *IAsynResult* beschäftigen.

Das große Computer ABC

C# lernen

Multithreading: Erstellen von Threads

Um einen Thread zu erstellen, müssen wir ein Objekt der Klasse *Thread* erzeugen. Der Konstruktor erwartet als Parameter einen Delegat des Typs *ThreadStart*. Ein **Delegat** (Mehrzahl Delegaten) kennzeichnet sich durch das Schlüsselwort *delegate* und stellt einen Prototyp einer Funktion dar. Delegaten kommen in Form von Funktionszeigern vor, welche eine Art **Referenz auf eine Funktion** darstellt. Delegaten werden unter anderem bei Events verwendet. Dem *ThreadStart*-Konstruktor wird als Parameter die Funktion, welche in einem neuen Thread arbeiten soll, übergeben. Die Funktion wird dabei nicht mit runden Klammern notiert, da die Funktion nicht aufgerufen werden soll, sondern die Referenz der Funktion übergeben werden soll. Natürlich arbeiten auch alle Funktionen, die von der „Start-Funktion“ (welche dem *ThreadStart*-Konstruktor übergeben wird) aufgerufen werden in dem neu erstellen Thread. Die Start-Funktion besitzt keine Übergabeparameter und darf keinen Rückgabewert haben. Mit der statischen Funktion *Sleep()* können wir den Thread, von welchem die Programmzeile ausgeführt wurde, für die übergebene Zahl an Millisekunden anhalten. Mit der Funktion *Abort()* können wir einen Thread abbrechen. Dabei wird der Programmcode, der im Thread aktuell ausgeführt wird, sofort beendet.

Program.cs

```
1  using System;
2  using System.Threading;
3
4  namespace CSV20.Mehrere_Threads
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             Thread oThreadGroßbuchstaben = new Thread(new ThreadStart(ThreadGroßbuchstaben));
11             Thread oThreadKleinbuchstaben = new Thread(new
   ThreadStart(ThreadKleinbuchstaben));
12
13             oThreadGroßbuchstaben.Start();
14             oThreadKleinbuchstaben.Start();
15
16             for (int i = 0; i < 5; i++)
17             {
18                 Console.WriteLine(i + 1);
19                 // Beim 3. Durchlauf Klein-Buchstaben-Thread abbrechen
20                 if (i == 2)
21                     oThreadKleinbuchstaben.Abort();
22                 Thread.Sleep(250);
23             }
24
25             Console.ReadKey();
26         }
27
28         private static void ThreadGroßbuchstaben()
29         {
30             for (int i = 0; i < 5; i++)
31             {
32                 Console.WriteLine((char)('A' + i));
33                 Thread.Sleep(250);
34             }
35         }
36
37         private static void ThreadKleinbuchstaben()
38         {
39             for (int i = 0; i < 5; i++)
40             {
41                 Console.WriteLine((char)('a' + i));
42                 Thread.Sleep(250);
43             }
44         }
45     }
46 }
```

Das große Computer ABC

C# lernen

Multithreading: Asynchrone Programmierung

Eine weitere Möglichkeit für eine spezielle Art von Multithreading in C# ist die asynchrone Programmierung. Bei Verwendung von asynchronen Programmierungen gibt es immer ein **UI-Thread** (Thread des User Interfaces, für die grafische Oberfläche) und den **Arbeitsthread** (für langwierige Aufgaben). Vor allem bei Anwendungen mit grafischer Oberfläche kommen wir in das Problem, dass wenn wir langwierige Aufgaben ausführen, das Fenster nicht mehr reagiert. Deshalb eignet sich die Auslagerung in einen neuen Thread oder in den Arbeitsthread (mit Hilfe der asynchronen Programmierung).
Für die asynchrone Programmierung benötigen wir den Delegat *AsyncCallback*. Eine Funktion des Delegaten *AsyncCallback* besitzt keinen Rückgabetyp und als Übergabeparameter das Interface (vergleichbar mit einer Klasse) *IAsyncResult*. Diese **Callback-Funktion** wird aufgerufen, sobald die Arbeitsfunktion abgeschlossen ist. Um dieses Beispiel gut erklären zu können, benötigen wir eine Klasse, in welcher wir eine Start- und Stopp-Funktion sowie die eigentliche Arbeitsfunktion integrieren. In der Klasse wird intern (*private*) ein eigener Delegat deklariert. Dieser Delegat wird für die Arbeitsfunktion verwendet. Der Vorteil bei der Deklaration eines eigenen Delegaten ist, dass wir den Rückgabetyp und die Übergabeparameter der Arbeitsfunktion selbst bestimmten können.
Im Hauptprogramm rufen wir nun die Start-Funktion unserer Klasse auf, nachdem wir ein Objekt unserer Klasse erzeugt haben. Diese Start-Funktion erstellt nun einen neuen Delegat auf die interne Arbeitsfunktion und startet die „Arbeit“ mit Hilfe der Funktion *BeginInvoke()*. Der *BeginInvoke()* werden die Parameter, welche im Delegat notiert werden, übergeben. Des Weiteren wird der Funktion noch die Callback-Funktion (oder auch *null*, falls nicht verwendet) und ein Objekt-Parameter (meistens *null*) übergeben. Sobald die Arbeitsfunktion beendet wurde, wird falls vorhanden, die Callback-Funktion aufgerufen. In der Callback-Funktion wird dann meistens die Stopp-Funktion aufgerufen, welche die *EndInvoke()*-Funktion aufruft, um den Rückgabewert (falls vorhanden) der Arbeitsfunktion zu erhalten. Das Beispiel wird diese Thematik genauer erläutern.

Program.cs

```
using System;
using System.Threading;

namespace CSV20.Asynchroner_Thread
{
    class Program
    {
        private static AsyncClass oClass;

        static void Main(string[] args)
        {
            AsyncCallback oCallback;
            IAsyncResult oResult;

            // Objekte vorbereiten
            oClass = new AsyncClass();
            oCallback = new AsyncCallback(AsynchroneAntwort);
            oResult = oClass.StartePotenzRechnung(2, 32, oCallback);

            // Main-Thread beschäftigen (Zahlen von 1 bis 25 ausgeben)
            for (int i = 1; i <= 25; i++)
            {
                Console.WriteLine("MAIN: {0}", i);
                Thread.Sleep(150);
            }

            Console.ReadKey();
        }

        private static void AsynchroneAntwort(IAsyncResult oAsync)
        {
            Console.WriteLine("MAIN: Endwert des asynchronen Handlers {0}",
oClass.BeendePotenzRechnung(oAsync));
        }
    }
}
```

AsyncClass.cs

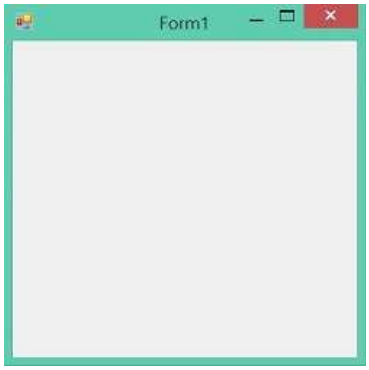
```
using System;
using System.Threading;

namespace CSV20.Asynchroner_Thread
```

```
{
    public class AsyncClass
    {
        private delegate long PotenzHandler(int x, int yMax);
        private PotenzHandler aktuellerHandler;

        // asynchrone Start-Funktion: ruft BeginInvoke() auf
        public IAsyncResult StartePotenzRechnung(int x, int yMain, AsyncCallback oCallback)
        {
            aktuellerHandler = new PotenzHandler(BerechnePotenzen);
            return aktuellerHandler.BeginInvoke(x, yMain, oCallback, null);
        }

        // asynchrone Stop-Funktion: ruft EndInvoke() auf
        public long BeendePotenzRechnung(IAsyncResult oAsynResult)
        {
            return aktuellerHandler.EndInvoke(oAsynResult);
        }

        // intere Funktion (wird synchron ausgeführt!)
        private long BerechnePotenzen(int x, int yMain)
        {
            long lPotenz = 0;

            for (int y = 0; y < yMain; y++)
            {
                lPotenz = (long)Math.Pow(x, y);
                Console.WriteLine("ASYNC: {0}^{1} = {2}", x, y, lPotenz);
                Thread.Sleep(100);
            }

            return lPotenz;
        }
    }
}
```

Das große Computer ABC

C# lernen

Windows Forms: Grundlagen



Um eine Windows Forms Applikation (kurz WinForm) zu erstellen, wählen wir bei der Erstellung eines Projekts **„Windows Forms-Anwendung“**. Durch das Erstellen eines solchen Projekts erhalten wir bereits einige C#-Dateien: Program.cs, Form1.cs und Form1.Designer.cs.
In der **Program.cs**-Datei befindet sich die *Main()*-Funktion, welche wir bereits aus den Konsolen-Anwendungen kennen. Dort werden die statischen Funktionen *EnableVisualStyles()*, *SetCompatibleTextRenderingDefault()* und *Run()* der Klasse *Application* aus dem Namensraum *System.Windows.Forms* (in welchem sich auch weitere Klassen für WinForm-Applikationen befinden) aufgerufen. Die ersten zwei Funktionen sind dabei für die Darstellung des Formulars zuständig. Der *Run()*-Funktion wird als Parameter ein Objekt unseres Formulars übergeben. Falls das Programm mehrere Formulare besitzt, wird der *Run()*-Funktion lediglich das **Start-Formular** übergeben. Weitere Formulare oder auch Dialoge werden dann vom Start-Formular aufgerufen (z. B. durch einen Button).
Die **Form1.cs**-Datei besitzt den eigentlichen Programmcode, welcher vom Programmierer geändert wird. Im Formular selbst werden vor allem Event-Funktionen (z. B. für den Klick auf ein Button) eingesetzt. Natürlich können Sie auch eigene Funktion in der Datei definieren. Bei großen Programmen ist es jedoch üblich, dass sich in den C#-Dateien eines Formulars ausschließlich Event-Funktionen befinden, welche dann Funktionen von eigen erstellten Klassen aufrufen. Hierdurch erhalten wir einen guten Programmierstil und eine ordentliche Kapselung von Funktionen in Klassen.
In der **Form1.Designer.cs**-Datei befindet sich der Code der Funktion *InitializeComponent()*, welche die einzelnen Steuerelemente des Formulars initialisiert und vom Konstruktor in der Form1.cs-Datei aufgerufen wird. Die Datei sollte nicht bearbeitet werden, da diese automatisch durch den Designer generiert wird. Der **Designer** ist ein Teil von Visual Studio, mit welchem einzelne Steuerelemente (mit Hilfe des Werkzeugkastens) auf das Formular gezogen, verschoben und editiert werden können. Die Eigenschaften eines Steuerelements (bei welchem es sich ebenfalls um ein Objekt handelt) können über das **Eigenschaftsfenster** editiert werden. Events bzw. deren Funktionen können durch einen Doppelklick auf das jeweilige Event eines Steuerelements, welche sich ebenfalls in dem Eigenschaftsfenster befinden, erstellt werden. Durch einen Doppelklick auf das Steuerelement selbst wird automatisch die Funktion des Standard-Events erstellt. Das einfache Anklicken eines Steuerelements selektiert das Steuerelement nur, sodass wir es editieren oder skalieren können.
In den Dateien Form1.cs und Form1.Designer.cs ist Ihnen vielleicht aufgefallen, dass sich vor dem Schlüsselwort *class* das Schlüsselwort *partial* befindet. Mit dem Schlüsselwort *partial* kennzeichnen wir, dass sich die Klasse über mehrere Dateien erstreckt. Für Programmierer ist es ganz interessant, den Quellcode der *InitializeComponent()*-Funktion zu sehen, weshalb wir empfehlen, bei einigen von unseren Programmen sich die Funktion anzuschauen, um C# besser zu verstehen.

Program.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Windows.Forms;
5
6  namespace CSV20.WinForm_Grundlagen
7  {
8      static class Program
9      {
10         /// <summary>
11         /// Der Haupteinstiegspunkt für die Anwendung.
12         /// </summary>
13         [STAThread]
14         static void Main()
15         {
16             Application.EnableVisualStyles();
17             Application.SetCompatibleTextRenderingDefault(false);
18             Application.Run(new Form1());
19         }
20     }
21 }
```

Form1.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.ComponentModel;
4  using System.Data;
5  using System.Drawing;
```

```
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace CSV20.WinForm_Grundlagen
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

Form1.Designer.cs

```
namespace CSV20.WinForm_Grundlagen
{
    partial class Form1
    {
        /// <summary>
        /// Erforderliche Designervariable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Verwendete Ressourcen bereinigen.
        /// </summary>
        /// <param name="disposing">True, wenn verwaltete Ressourcen gelöscht werden sollen;
andernfalls False.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Vom Windows Form-Designer generierter Code

        /// <summary>
        /// Erforderliche Methode für die Designerunterstützung.
        /// Der Inhalt der Methode darf nicht mit dem Code-Editor geändert werden.
        /// </summary>
        private void InitializeComponent()
        {
            this.components = new System.ComponentModel.Container();
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.Text = "Form1";
        }

        #endregion
    }
}
```

Das große Computer ABC

C# lernen

Windows Forms: Eigenschaften und Ereignisse eines Formulars

Ein Formular (also die Klasse *Form*) hat einige Eigenschaften und Ereignisse, um dessen Aussehen zu steuern und zu beeinflussen. Natürlich werden wir hier nicht alle Eigenschaften und Ereignisse vorstellen. Dies gilt auch für die weiteren Themen zu den Windows Forms Steuerelementen. Viele der Eigenschaften können durch den englischen Namen abgeleitet werden. Des Weiteren hilft uns das Eigenschaftsfenster bei der Editierung von Eigenschaften. Alle Eigenschaften können über den Designer und zur Programmlaufzeit über den Programmcode geändert werden. Die Eigenschaft *Name* ist eine spezielle Eigenschaft, die im Designer zusätzlich zum internen Namen des Steuerelements dazu verwendet wird, den Variablennamen festzulegen (standardmäßig werden Namen wie label1, label2 etc. verwendet, für eine bessere Übersicht sollten diese jedoch geändert werden).

Wollen wir uns am Anfang einmal **allgemeingültige Eigenschaften** anschauen, welche auch für andere Steuerelemente verwendet werden können. Die Eigenschaft *BackColor* und *ForeColor* ändert die **Hintergrund- bzw. Vordergrundfarbe** (oftmals Schriftfarbe). Mit *Enabled* kann das Steuerelement aktiviert bzw. deaktiviert werden. *Width* und *Height* sind Eigenschaften für die **Größe**. Mit *AutoSize* kann festgelegt werden, ob die Größe automatisch ermittelt werden soll. Die *Location*-Eigenschaft besteht aus einer x- und y-Koordinate, welche zur Platzierung des Steuerelements zuständig ist. Ähnlich zur *Location*-Eigenschaft gibt es als Alternative zu der *Width*- und *Height*-Eigenschaft, die Eigenschaft *Size* für die Größe. Mit der Eigenschaft *Visible* kann ein Steuerelement ein- und ausgeblendet werden. Um den **Titel** (oder bei anderen Steuerelementen den „Text“ bzw. Inhalt) zu ändern, kann die *Text*-Eigenschaft gesetzt werden. Die Eigenschaft *TabIndex* ist eine sehr wichtige Eigenschaft, um die **Tabreihenfolge** zu verändern. Diese Eigenschaft ist jedoch erst dann wichtig, wenn unser Formular mehrere fokussierbare Steuerelemente (wie z. B. ein Textfeld oder einen Button) besitzt. Die Anordnung der Tabreihenfolge wird automatisch beim Erstellen von Steuerelementen getroffen, d. h. das Element, welches als letztes im Designer eingefügt wurde, besitzt den größten „Tab-Index“ und wird dadurch als letztes per Tabulator-Taste fokussiert, bevor das Ganze wieder von vorne beginnt.

Nun wollen wir uns noch einige **spezielle Eigenschaften der Form-Klasse** anschauen: Mit der Eigenschaft *FormBorderStyle* legen wir einen Wert der gleichnamigen Enumeration fest, mit welcher die **Rahmenart** des Formulars gesteuert werden kann: *FixedSingle* (Formular welches nicht vergrößert werden kann), *FixedDialog* (Formular welches als Dialog-Fenster genutzt wird und nicht vergrößert werden kann), *Sizable* (Formular welches vergrößert werden kann) und *None* (Formular ohne Rahmen). Über die *MinimizeBox*- und *MaximizeBox*-Eigenschaft können die **Steuerungs-Buttons in der Fensterleiste** und somit auch deren Funktion deaktiviert werden, d. h. durch die Zuweisung des Werts *false* an die *MaximizeBox*-Eigenschaft wird der Maximierungs-Button ausgegraut. Zugleich ist es dann nicht mehr möglich, das Fenster zu maximieren. *StartPosition* ist eine Eigenschaft, um die **Position des Fensters** beim Programmstart festzulegen. Hierbei wird ebenfalls ein Wert der gleichnamigen Enumeration zugewiesen: *CenterParent* (zentrierte Ausrichtung zum übergeordneten Fenster), *CenterScreen* (zentrierte Ausrichtung zum Bildschirm), *Manual* (manuelle Festlegung der Position durch die Eigenschaft *Location*) und *WindowsDefaultLocation* (Standard-Ausrichtung von Windows). Die Eigenschaft *ShowIcon* und *ShowInTaskbar* steuert die Anzeige des Icons in der Fensterleiste und in der Taskleiste. *WindowState* ist eine Eigenschaft und Enumeration, welche angibt, in welchem Modus das Fenster beim Starten geöffnet werden soll: Normal (Normal-Modus), Maximized (maximiert) und Minimized (minimiert). Diese Eigenschaft kann auch zur Laufzeit geändert werden, um z. B. das Fenster zu minimieren.

Als nächstes stellen wir einige **allgemeingültige Ereignisse** vor: Hier ist z. B. das *Click*-Event zu nennen, welches bei einem **einfachen Mausklick** eintritt. Dieses Event wird z. B. bei Buttons verwendet. Für die Tastatur- und Maussteuerung gibt es noch einige andere Events, welche wir uns jedoch später genauer anschauen. Das Ereignis *Paint* tritt ein, wenn das **Steuerelement gezeichnet** wird. Bei der grafischen Programmierung wird die Ereignis-Funktion dieses Events dazu verwendet, spezielle Steuerelemente bzw. grafische Teile zu zeichnen. Um ein Steuerelement dazu zu veranlassen, dass es erneut gezeichnet wird, können wir die Funktion *Invalidate()* aufrufen. Hierdurch wird das Steuerelement erneut gezeichnet und anschließend das *Paint*-Ereignis ausgelöst.

Auch die *Form*-Klasse verfügt über einige **spezielle Events**: Das Load-Event tritt ein bevor das Formular angezeigt wird. Das *FormClosing*-Event tritt ein bevor das Formular geschlossen wird. Da dem Ereignis *FormClosingEventArgs* als Event-Argumente übergeben werden, können wir über die Eigenschaft *Cancel* den Schließungs-Vorgang des Formulars abbrechen. Wird der Schließungs-Vorgang nicht abgebrochen, so wird das Formular geschlossen und das Event *FormClosed* ausgelöst.

Eine sehr praktische Funktion ist die statische Funktion *Show()* der Klasse *MessageBox,* über welche wir eine **Nachrichten-Box** (auch Meldungsfenster genannt) anzeigen können. Der Funktion können bis zu fünf Argumente übergeben werden. Dabei können die letzten in absteigender Reihenfolge weggelassen werden. Als erstes Argument übergeben wir den Text, gefolgt vom Titel. Der dritte Parameter ist ein Wert der Enumeration *MessageBoxButtons*, welcher die verschiedenen anzuzeigende Buttons auswählt: *OK*, *OKCancel*, *RetryCancel*, *YesNo*, *YesNoCancel* und *AbortRetryIgnore*. Der vierte Übergabeparameter legt das anzuzeigende Icon an. Hierfür wird ein Wert der Enumeration *MessageBoxIcon* benötigt: *Information*, *Warning*, *Stop* und *Question*. Mit der Enumeration *MessageBoxDefaultButton* können wir den fünften und letzten Übergabeparameter festlegen. Hiermit wird der Button ausgewählt, welcher standardmäßig fokussiert werden soll, festgelegt: *Button1*, *Button2* und *Button3*. Der Rückgabewert der *Show()*-Funktion ist ein Wert der Enumeration *DialogResult*, mit *welcher* angegeben wird, mit welchem Button das Meldungsfenster geschlossen wurde: *OK*, *Cancel*, *Retry*, *Yes*, *No*, *Abort* und *Ignore*. Bei der Abfrage des Werts bzw. dem Speichern des Werts in einer Variablen muss der vollständige Namensraum angegeben werden, da andernfalls keine Unterscheidung zwischen der Enumeration *DialogResult* und der Eigenschaft *DialogResult* des Formulars möglich wäre (siehe Beispiel).

Form1.cs

```
1  private void Form1_FormClosing(object sender, FormClosingEventArgs e)
2  {
3      // Beenden-Meldung anzeigen, falls "Nein" geklickt wird, Schließen-Vorgang des Formulars
   beenden
4      if (MessageBox.Show("Sind Sie sicher, dass Sie das Programm beenden möchten?",
5                          "Programm beenden?",
6                          MessageBoxButtons.YesNo,
7                          MessageBoxIcon.Question,
8                          MessageBoxDefaultButton.Button2) ==
   System.Windows.Forms.DialogResult.No)
9          e.Cancel = true;
10 }
11
12 private void Form1_Load(object sender, EventArgs e)
13 {
14     MessageBox.Show("Das Formular der Anwendung wurde geladen!");
15 }
```

Das große Computer ABC

C# lernen

Windows Forms: Labels und Eingabefelder

Labels (Klasse *Label*) sind Textfelder, die vom Benutzer nicht verändert werden können. Der Haupteinsatz von Labels findet sich in **Beschriftungen** (z. B. Beschriftung vor einem Eingabefeld, sodass der Benutzer weiß, was er in das Eingabefeld eingeben soll). Über die Eigenschaft *Text* können wir den Text festlegen. Die Eigenschaft *AutoSize* ist bei diesem Steuerelement standardmäßig auf *true* eingestellt. Mit der *TextAlign*-Eigenschaft können wir die Ausrichtung des Textes steuern. Beachten Sie, dass diese Eigenschaft nur von Bedeutung ist, wenn *AutoSize* auf *false* gesetzt wird.
Ein Label der Klasse *LinkLabel* ist ein spezielles Label, mit welchem **Links** erstellt werden können. Durch das Event *LinkClicked* können wir das Klicken auf den Link abfangen, um ein Programm oder eine Webseite zu öffnen. Dabei wird die *Process*-Klasse zum Starten eines Prozesses, einer Webseite oder einer Datei verwendet. Über die Eigenschaft *LinkVisited* kann festgelegt werden, ob der Link bereits angeklickt wurde. Hiermit ändert sich die Farbe des Links von der ursprünglichen Farbe, welche in *LinkColor* hinterlegt ist in den Wert der Eigenschaft *VisitedLinkColor*.
Ein **Textfeld** kann mit der Klasse *TextBox* erstellt werden. Der Inhalt eines Textfeldes kann vom Benutzer normalerweise geändert werden. Wird die Eigenschaft *ReadOnly* auf *true* eingestellt, so kann der Inhalt des Textfeldes nicht mehr editiert werden. Die Hintergrundfarbe (*BackColor*) wird dabei vom Visual Studio Designer automatisch auf grau eingestellt. Mit der Eigenschaft *MaxLength* können wir die maximale Anzahl an Zeichen begrenzen. Die Höhe eines Textfeldes ist immer von der gewählten Schriftart und -größe abhängig. Wenn wir ein **mehrzeiliges Textfeld** haben möchten, müssen wir der Eigenschaft *Multiline* den Wert *true* zuweisen. Danach kann auch die Höhe der *TextBox* verändert werden. Das Abrufen oder Zuweisen des Inhalts passiert über die Eigenschaft *Text* oder auch *Lines* (*string*-Array, wird für mehrzeilige Textfelder verwendet). Mit Hilfe des Ereignisses *TextChanged* können wir auf Änderungen am Inhalt einer TextBox reagieren. Im Beispiel wird der vom Benutzer eingegebene Text in einem darunter angeordneten *Label*-Objekt angezeigt. Um ein Passwortfeld zu erstellen, muss der Wert in der Eigenschaft *PasswordChar* auf ein beliebiges **Pseudo-Zeichen** gesetzt werden. Hierdurch wird bei der Eingabe in der *TextBox* nicht das eingegebene Zeichen, sondern das Pseudo-Zeichen angezeigt. Der „reale“ Inhalt kann weiterhin über die Eigenschaft *Text* abgerufen werden.
Ein **numerisches Eingabefeld** können wir mit Hilfe des *NumericUpDown*-Steuerelements erzeugen. Hier kann ein Wert über die Eigenschaft *Value* abgerufen oder zugewiesen werden. Ein Vorteil des numerischen Eingabefeldes ist die Festlegung einer Obergrenze (Eigenschaft *Maximum*) und einer Untergrenze (Eigenschaft *Minimum*). Die Eigenschaft *ThousandsSeparator* legt fest, ob ein Tausendertrennzeichen angezeigt werden soll oder nicht. Mit Hilfe des *ValueChanged*-Events können Änderungen am numerischen Wert abgefragt werden.

Form1.cs

```
1  private void textBoxEingabe_TextChanged(object sender, EventArgs e)
2  {
3      labelEingabe.Text = textBoxEingabe.Text;
4  }
5
6  private void numericUpDownEingabe_ValueChanged(object sender, EventArgs e)
7  {
8      if (numericUpDownEingabe.Value == numericUpDownEingabe.Minimum)
9          MessageBox.Show("Der Minimal-Wert wurde erreicht!");
10     else if (numericUpDownEingabe.Value == numericUpDownEingabe.Maximum)
11         MessageBox.Show("Der Maximal-Wert wurde erreicht!");
12 }
13
14 private void linkLabelWebseite_LinkClicked(object sender, LinkLabelLinkClickedEventArgs e)
15 {
16     Process.Start("http://www.das-grosse-computer-abc.de/CSharp/");
17
18     linkLabelWebseite.LinkVisited = true;
19 }
```

Das große Computer ABC

C# lernen

Windows Forms: Buttons

Ein *Button* ist ein sehr wichtiges und zugleich auch sehr einfaches Steuerelement, welches dazu verwendet wird, Daten zu speichern, ein Formular zu öffnen o. A.. Die Eigenschaft *Text* legt den anzuzeigenden Text innerhalb des Rahmens des Buttons fest. Ein *Button*-Steuerelement kann nach Belieben skaliert werden, da *AutoSize* standardmäßig auf *false* eingestellt ist. Durch das *Click*-Event können wir das Einfachklicken auf den Button durch den Benutzer abfangen. Hierbei handelt es sich um das Standardevent.

Form1.cs

```
1  private void buttonMsgBox_Click(object sender, EventArgs e)
2  {
3      MessageBox.Show("Sie haben auf den Button geklickt!");
4  }
```

Das große Computer ABC

C# lernen

Windows Forms: CheckBox und RadioButton

CheckBox und *RadioButton* sind Steuerelemente, welche Sie vermutlich bereits aus **Formularen** (sowohl auf Papierform, als auch im Internet sowie in Programmen) kennen. Eine *CheckBox* (**Kontrollkästchen**) kennzeichnet sich durch das eckige Kästchen. Ein *RadioButton* (**Auswahlfeld**) kennzeichnet sich durch das runde Kästchen. Bei gruppierten Auswahlfeldern von *RadioButton*-Steuerelementen darf immer nur eines davon gewählt werden. Da wir zum jetzigen Zeitpunkt noch keine Möglichkeit kennen, Steuerelemente zu gruppieren, können wir aktuell nur eine Gruppe von *RadioButton*-Steuerelementen erzeugen (hierbei dient das Formular selbst als ein Steuerelement zur Gruppierung).
Sowohl bei *CheckBox*- als auch bei *RadioButton*-Steuerelementen können wir über die Eigenschaft *Checked* prüfen, ob ein **Häkchen** gesetzt ist oder nicht. Das Setzen des Hakens erfolgt vom Benutzer durch das Klicken auf das Steuerelement (Text oder auch Kästchen). Die Eigenschaft *Text* entspricht bei diesen Steuerelementen der Beschriftung, die zumeist rechts vom Kästchen angezeigt wird. Die Position des Kästchens kann jedoch durch die Eigenschaft *CheckAlign* verändert werden.
Das Ereignis *CheckedChanged* tritt ein, sobald sich der Wert der *Checked*-Eigenschaft ändert. Normalerweise passiert dies ausschließlich über das Setzen oder Wegnehmen eines Hakens auf der grafischen Oberfläche durch den Benutzer.
Im Beispiel können Sie sehen, dass wir in der Funktion *radioButtonAuswahl_CheckedChanged()* den Übergabeparameter *sender* nutzen. Dies liegt daran, dass wir für alle drei Radio-Buttons die gleiche Event-Funktion nutzen. Um nun unterscheiden zu können, welcher Radio-Button das Event ausgelöst hat, können wir das Objekt *sender* nutzen. Dieses Objekt entspricht dem Steuerelement, welches das Ereignis ausgelöst hat. Dies ist gängige Programmierpraxis und sollte immer dann eingesetzt werden, wenn es sich lohnt (z. B. um Programmcode zu sparen).

Form1.cs

```
1  private void checkBoxAktivierung_CheckedChanged(object sender, EventArgs e)
2  {
3      if (checkBoxAktivierung.Checked)
4      {
5          radioButtonAuswahl1.Enabled = radioButtonAuswahl2.Enabled =
   radioButtonAuswahl3.Enabled = true;
6          // Bei Reaktivierung müssen alle RadioButton nacheinander geprüft werden ob diese
7          // gewählt sind, sodass wir den richtigen Text anzeigen können
8          if (radioButtonAuswahl1.Checked)
9              labelInfo.Text = radioButtonAuswahl1.Text + " gewählt!";
10         else if (radioButtonAuswahl2.Checked)
11             labelInfo.Text = radioButtonAuswahl2.Text + " gewählt!";
12         else if (radioButtonAuswahl3.Checked)
13             labelInfo.Text = radioButtonAuswahl3.Text + " gewählt!";
14     }
15     else
16     {
17         labelInfo.Text = "Kein RadioButton ausgewählt!";
18         radioButtonAuswahl1.Enabled = radioButtonAuswahl2.Enabled =
   radioButtonAuswahl3.Enabled = false;
19     }
20 }
21
22 // Event-Funktion wird für alle 3 RadioButtons verwendet, über das sender-Objekt ermitteln
   wir die RadioButton
23 private void radioButtonAuswahl_CheckedChanged(object sender, EventArgs e)
24 {
25     RadioButton oRadioButton = (RadioButton)sender;
26
27     // Ereignis wird mehrmals ausgelöst, uns interessiert jedoch nur das Ereignis von
28     //  dem RadioButton, welcher aktuell ausgewählt ist
29     if (oRadioButton.Checked)
30         labelInfo.Text = oRadioButton.Text + " gewählt!";
31 }
```

Das große Computer ABC

C# lernen

Windows Forms: GroupBox

Die *GroupBox* ist ein Steuerelement, um Steuerelemente zu gruppieren. Oft wird ein solches Steuerelement zur Gruppierung von *RadioButton*-Steuerelementen verwendet. Zur Beschriftung der *GroupBox* kann die *Text*-Eigenschaft verwendet werden. Ein Merkmal der *GroupBox* ist, dass sich um das Steuerelement herum ein Rahmen befindet. Die Ecken dieses Rahmens sind leicht abgerundet. Ist eine Beschriftung vorhanden, so ist der Rahmen an der Stelle, wo sich der Titel befindet, unterbrochen. Beim Bearbeiten eines Formulars im Designer sollte darauf geachtet werden, dass das Element, welches untergeordnet werden soll, sich auch wirklich im *GroupBox*-Steuerelement befindet.

Form1.cs

```
1  private void groupBox1_radioButton_CheckedChanged(object sender, EventArgs e)
2  {
3      RadioButton oRadioButton = (RadioButton)sender;
4
5      if (oRadioButton.Checked)
6          labelG1.Text = oRadioButton.Text;
7  }
8
9  private void groupBox2_radioButton_CheckedChanged(object sender, EventArgs e)
10 {
11     RadioButton oRadioButton = (RadioButton)sender;
12
13     if (oRadioButton.Checked)
14         labelG2.Text = oRadioButton.Text;
15 }
```

Das große Computer ABC

C# lernen

Windows Forms: ListBox und ComboBox

Die *ListBox* ist ein Steuerelement, um **ein oder mehrere Elemente** auszuwählen. Standardmäßig kann in einer *ListBox* nur ein Element gewählt werden. Durch die Eigenschaft *SelectionMode* kann dies geändert werden. Setzen wir diesen Wert auf *MultiExtended*, so können null bis alle Elemente gewählt werden. Die einzelnen Elemente befinden sich in der **Liste** mit dem Eigenschaftsnamen *Items*. Die Eigenschaft *Text* (oder auch *SelectedItem*) gibt den Namen des aktuell ausgewählten Elementes zurück. Bei einer *ListBox* bei der eine Mehrfach-Auswahl möglich ist, können die gerade genannten Eigenschaften nicht verwendet werden. Hier sollte auf die Eigenschaft *SelectedItems* zugegriffen werden, welche eine Liste darstellt. Um Änderungen bei der Auswahl in einer *ListBox* mitzubekommen, können wir das Ereignis *SelectedIndexChanged* verwenden. Die Höhe einer *ListBox* kann verändert werden, jedoch zeigt eine *ListBox* immer nur vollständige Elemente an (und somit mindestens ein Element).
Eine *ComboBox* ist ein Steuerelement, welches teilweise der *ListBox* ähnelt. Wie bei der *ListBox* enthält auch die *ComboBox* eine **Liste an Elementen**. Eine *ComboBox* kann des Weiteren auch für die **Eingabe von Text** genutzt werden. Dieser Effekt ist oft nicht erwünscht, weshalb mit der Eigenschaft *DropDownStyle* und dem Wert *DropDownList* der Enumeration *ComboBoxStyle* die *ComboBox* zu einer **Drop-Down-Liste** umgewandelt werden kann, bei der keine Texteingabe möglich ist. Von der Höhe ist die *ComboBox* immer so hoch wie ein einzelnes Element (was wiederum abhängig von der Schriftart und -größe ist). Auch bei der *ComboBox* kann über die Eigenschaft *Text* und *SelectedItem* der Name des Elements abgerufen werden. Bei einer *ComboBox*, bei welcher eine Texteingabe möglich ist, sollte die Eigenschaft *Text* bevorzugt werden. Das Ereignis *SelectedIndexChanged* ist auch bei einer *ComboBox* verfügbar. Für eine *ComboBox* mit Text-Eingabe sollte jedoch u. U. das Event *TextChanged* bevorzugt werden.

Form1.cs

```
1  private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
2  {
3      labelListBox1.Text = listBox1.Text;
4  }
5
6  private void listBox2_SelectedIndexChanged(object sender, EventArgs e)
7  {
8      labelListBox2.Text = "";
9      for (int i = 0; i < listBox2.SelectedItems.Count; i++)
10         labelListBox2.Text += (string)listBox2.SelectedItems[i] + (i !=
   (listBox2.SelectedItems.Count - 1) ? "; " : "");
11 }
12
13 private void comboBox1_TextChanged(object sender, EventArgs e)
14 {
15     labelComboBox1.Text = comboBox1.Text;
16 }
17
18 private void comboBox2_SelectedIndexChanged(object sender, EventArgs e)
19 {
20     labelComboBox2.Text = comboBox2.Text;
21 }
```

Das große Computer ABC

C# lernen

Windows Forms: Grafiken

Für die **Anzeige von Bildern** bzw. Grafiken können wir das Steuerelement *PictureBox* verwenden. Um die Anzeige des Bildes innerhalb einer *PictureBox* zu steuern, gibt es die Eigenschaft *SizeMode*. Hier wird ein Wert der Enumeration *PictureBoxSizeMode* festgelegt: *AutoSize* (*PictureBox* wird automatisch auf die Originalgröße des Bildes skaliert), *CenterImage* (Bild wird zentriert und abgeschnitten falls es zu groß ist), *StretchImage* (Bild wird vergrößert oder verkleinert, sodass es passend zur Größe der *PictureBox* ist), *Zoom* (Bild wird vergrößert oder verkleinert, sodass es passend zur Größe der *PictureBox* ist, jedoch wird das Seitenverhältnis beibehalten) und *Normal* (Bild wird links oben platziert und abgeschnitten, falls es zu groß ist). Das Bild selbst wird über die Eigenschaft *Image* geladen. Bei statischen *PictureBox*-Steuerelementen wird hier über den Designer ein Bild in die **Ressourcen-Liste** eingebunden und geladen. Wollen wir ein Bild über den Programmcode **dynamisch von einer Datei laden**, können wir die statische Funktion *FromFile()* von der *Image*-Klasse aufrufen. Als Parameter wird der Funktion der Dateiname (mit Pfad) übergeben.

Form1.cs

```
1  private bool bCSharpIconShown = true;
2
3  private void pictureBox1_DoubleClick(object sender, EventArgs e)
4  {
5      if (bCSharpIconShown)
6          pictureBox1.Image = Image.FromFile("Logo.png");
7      else
8          pictureBox1.Image = Resources.CSharp_Icon;
9      bCSharpIconShown = !bCSharpIconShown;
10 }
```

Das große Computer ABC

C# lernen

Windows Forms: Timer und Fortschrittsbalken

Um in grafischen Oberflächen einen Fortschrittsbalken anzuzeigen, benötigen wir das Steuerelement *ProgressBar*. Um den Wert eines Fortschrittsbalkens zu steuern, gibt es die Eigenschaften *Value* sowie die Funktion *PerformStep()*. Mit Hilfe der Eigenschaft *Value* könnten wir den Fortschrittsbalken auch wieder zurücksetzen oder Sprünge durchführen. Die *PerformStep()*-Funktion ändert die *Value*-Eigenschaft immer um den Wert, welcher in der Eigenschaft *Step* hinterlegt ist. Um den Maximalwert eines *ProgressBar*-Steuerelements festzulegen, können wir die Eigenschaft *Maximum* setzen.
Zusätzlich zum normalen **fortlaufenden Balken** in dem Fortschrittsbalken können wir auch einen **Laufbalken** erstellen. Dazu muss die Eigenschaft *Style* auf den Wert *Marquee* der Enumeration *ProgressBarStyle* geändert werden. Die Eigenschaft *MarqueeAnimationSpeed* legt die Geschwindigkeit des Laufbalkens fest. Wird die Eigenschaft auf null gesetzt, so hält die Animation und dadurch auch der Laufbalken an.
Der *Timer* ist ein spezielles Steuerelement, welches sich ebenfalls im Werkzeugkasten befindet und auf das Formular „gezogen“ wird, jedoch **nicht auf dem Formular angezeigt** wird. Das *Timer*-Steuerelement wird in einem Bereich unterhalb des Formulars angezeigt. Der *Timer* dient dazu, ein **Event in einem bestimmten Rhythmus** (Intervall) auszuführen. Über die Eigenschaft *Intervall* kann der Intervall des Timers in Millisekunden angegeben werden. Durch das Event *Tick* können wir eine Funktion registrieren, welche in dem vom *Timer* festgelegten Intervall ausgeführt werden soll. Das *Timer*-Steuerelement verfügt über die Funktion *Start()* und *Stop(),* um den *Timer* zu starten oder anzuhalten.

Form1.cs

```
1  private void buttonStart_Click(object sender, EventArgs e)
2  {
3      // Timer starten
4      timerFortschritt.Start();
5
6      // Fortschrittsbalken zurücksetzen und -Animation vorbereiten
7      progressBarTimerBlocks.Value = 0;
8      progressBarTimerMarquee.MarqueeAnimationSpeed = 50;
9
10     // Buttons-Aktivierung ändern
11     buttonStart.Enabled = false;
12     buttonStop.Enabled = true;
13 }
14
15 private void buttonStop_Click(object sender, EventArgs e)
16 {
17     // Timer stoppen
18     timerFortschritt.Stop();
19
20     // Fortschrittsbalken-Animation stoppen
21     progressBarTimerMarquee.MarqueeAnimationSpeed = 0;
22
23     // Buttons-Aktivierung ändern
24     buttonStart.Enabled = true;
25     buttonStop.Enabled = false;
26 }
27
28 private void timerFortschritt_Tick(object sender, EventArgs e)
29 {
30     // Fortschrittsbalken bewegen
31     progressBarTimerBlocks.PerformStep();
32
33     // Falls Maximum erreicht wurde, Timer stoppen (Event des Stop-Buttons aufrufen)
34     if (progressBarTimerBlocks.Value == progressBarTimerBlocks.Maximum)
35         buttonStop_Click(null, EventArgs.Empty);
36 }
```

Das große Computer ABC

C# lernen

Windows Forms: Menüs

In C# gibt es zwei verschiedene Arten von Menüs: das *MenuStrip* und das *ContextMenuStrip*. Ein Formular besitzt meistens ein *MenuStrip*, welches am **oberen Fensterrand** platziert ist. Bei kleineren Anwendungen ist eine solche Menüleiste oft auch nicht vorhanden. Einem Formular kann das (Haupt-)Menü über die Eigenschaft *MainMenuStrip* zugewiesen werden. Das *ContextMenuStrip* ist, wie der Name schon vermuten lässt, ein Kontextmenü, welches beim Rechtsklick auf ein Steuerelement angezeigt wird. Dafür muss bei dem Steuerelement, welchem das **Kontextmenü** zugewiesen werden soll, die Eigenschaft *ContextMenuStrip* gesetzt werden.
Beiden Menü-Arten können Steuerelemente der Klassen *ToolStripMenuItem* (vergleichbar mit *Label*), *ToolStripTextBox* (vergleichbar mit *TextBox*), *ToolStripComboBox* (vergleichbar mit *ComboBox*) und *ToolStripSeparator* (horizontale Linie zur Trennung) untergeordnet werden. Bei der Zusammenstellung eines Menüs ist uns der Designer eine große Hilfe. Ein Steuerelement der Klasse *ToolStripMenuItem* kann nach Belieben verschachtelt werden (siehe Beispiel).
Für das Event-Handling gibt es zwei verschiedene Ansätze: das *ItemClicked*-Event der *MenuStrip*- oder *ContextMenuStrip*-Klasse oder das *Click*-Event der einzelnen Tool-Strips. Das Problem des *ItemClicked*-Event ist, dass dieses Ereignis auch bei einem verschachtelten Item auftritt, was meistens jedoch nicht erwünscht ist. Das Problem des *Click*-Events eines einzelnen Tool-Strips ist, dass wir es bei allen Tool-Strips „manuell“ hinzufügen müssen. Im Beispiel haben wir uns für die einzelnen *Click*-Events entschieden. Über die Eigenschaft *OwnerItem* der Klasse *ToolStripMenuItem* können wir auf das übergeordnete Item zugreifen. Wenn kein übergeordnetes Item vorhanden ist bzw. keine Verschachtelung existiert, gibt die Eigenschaft *null* zurück.

Form1.cs

```
1  private void menuStripFenster_ItemClicked(object sender, EventArgs e)
2  {
3      ToolStripMenuItem oMenuItem = (ToolStripMenuItem)sender;
4      ToolStripMenuItem oMenuItemParent = (ToolStripMenuItem)oMenuItem.OwnerItem;
5      ToolStripMenuItem oMenuItemParentParent = (ToolStripMenuItem)oMenuItemParent.OwnerItem;
6
7      // falls Beenden-ToolStrip angeklickt wurde, Fenster schließen
8      if (oMenuItem == beendenToolStripMenuItem)
9          Close();
10
11     // Ausgabe der geklickten Hierarchie (oMenuItemParentParent kann null sein, wenn
   Hierarchie nur aus 2 Elementen besteht)
12     if (oMenuItemParentParent == null)
13         MessageBox.Show("Sie haben auf \"" + oMenuItemParent.Text + "\" -> \"" +
   oMenuItem.Text + "\" geklickt!");
14     else
15         MessageBox.Show("Sie haben auf \"" + oMenuItemParentParent.Text + "\" -> \"" +
   oMenuItemParent.Text + "\" -> \"" + oMenuItem.Text + "\" geklickt!");
16 }
17
18 private void contextMenuStripButton_ItemClicked(object sender, EventArgs e)
19 {
20     if (sender == minimierenToolStripMenuItem)
21         this.WindowState = FormWindowState.Minimized;
22     else
23         Close();
24 }
```

Das große Computer ABC

C# lernen

Windows Forms: Tabs

Um in einer Windows Forms Anwendung einen Bereich mit **mehreren Registerkarten** zu erstellen, benötigen wir das *TabControl*-Steuerelement. Dieses Steuerelement verwaltet und enthält die sogenannten Registerkarten (oder auch Tab-Seiten genannt). Hierfür benötigen wir das *TabPage*-Steuerelement. Die Tab-Seiten können sowohl über den Designer, als auch während der Laufzeit erstellt, gelöscht oder geändert werden. Das *TabPage*-Steuerelement ist ebenfalls ein Steuerelement, mit welchem andere Steuerelement gruppiert zusammengefasst werden. Durch die Eigenschaft *SelectedIndex* des *TabControl*-Steuerelements können wir den Index der aktuell ausgewählten Tab-Seiten festlegen oder abrufen.
Im Beispiel verwenden wir die Eigenschaft *Dock* des *TabControl*-Steuerelements. Über die Enumeration *DockStyle* weisen wir der Eigenschaft den Wert *Fill* zu. Dadurch füllt das Steuerelement den kompletten Bereich des übergeordneten Steuerelements aus. Über die gleiche Eigenschaft könnten wir auch eine Anordnung für oben (*Top*), links (*Left*), unten (*Bottom*) und rechts (*Right*) festlegen.

Form1.cs

```
1  private void buttonGeheZuTab2_Click(object sender, EventArgs e)
2  {
3      // Wechsle zu Tab 2 (Index 0)
4      tabControl1.SelectedIndex = 1;
5  }
6
7  private void buttonÄndereHintergrund_Click(object sender, EventArgs e)
8  {
9      // Hintergrund-Farbe des 2. Tabs zwischen Blau und Weiß wechseln
10     if (tabPage2.BackColor == Color.Blue)
11         tabPage2.BackColor = Color.White;
12     else
13         tabPage2.BackColor = Color.Blue;
14 }
15
16 private void buttonBeenden_Click(object sender, EventArgs e)
17 {
18     this.Close();
19 }
```

Das große Computer ABC

C# lernen

Windows Forms: Panels

Panels (Klasse *Panel*) dienen zur **Gruppierung von Steuerelementen**. Viel mehr gibt es zu diesem Steuerelement nicht zu sagen, da es einfach nur zur Gruppierung genutzt wird. Natürlich besitzt das Steuerelement auch allgemeingültige Eigenschaften und Events. Des Weiteren gilt zu sagen, dass auch eine Anordnung der Steuerelemente an Hand der *Dock*-Eigenschaft innerhalb eines Panels möglich ist.

Das große Computer ABC

C# lernen

Windows Forms: Dialoge

Im .NET-Framework gibt es einige **vorgefertigte Dialoge**, um Dateien zu öffnen oder zu speichern, einen Ordner auszuwählen und viele mehr. Alle Dialoge können über die Funktion *ShowDialog()* aufgerufen und angezeigt werden. Dadurch ist es nicht möglich, zum vorherigen Fenster zurückzukehren, ohne den Dialog zu schließen. Die *ShowDialog()*-Funktion gibt einen Wert der Enumeration *DialogResult* zurück, mit welcher geprüft werden kann, ob der Dialog mit einem Klick auf „OK“ beendet wurde.
Der *OpenFileDialog* ist ein Dialog, mit welchem eine **Datei zum Öffnen** ausgewählt werden kann. Der Dialog zeigt ein standardisiertes Formular zum Auswählen einer Datei: links eine Navigation auf dem Computer, rechts die Dateiauswahl und unten ein Eingabefeld und Buttons. Über die Eigenschaft *Multiselect* wird angegeben, ob wir mehrere Dateien auswählen können oder nicht. Den resultierenden Dateinamen bzw. die resultierende Dateinamen können wir über die Eigenschaft *FileName* und *FileNames* abrufen. Die Eigenschaft *Filter* legt einen **Auswahlfilter** für Dateinamen fest. Dabei werden der Anzeige-Name und der Filter (z. B. *.txt für alle Dateinamen) von einem Senkrechtstrich getrennt. Natürlich können auch mehrere Filter genutzt werden. Die Trennung erfolgt auch hier mit einem Senkrechtstrich. Die Eigenschaft *FilterIndex* legt den gewählten Index fest oder gibt diesen zurück. Um einen Titel in der Fensterleiste anzuzeigen, können wir einen *string* in der Eigenschaft *Title* festlegen. Mit der Eigenschaft *DefaultExt* legen wir fest, welche Dateinamenserweiterung angefügt werden soll, wenn keine eingegeben wurde. Der *SaveFileDialog* ist das Gegenstück des *OpenFileDialog*. Hiermit wird ein Dialog zum **Speichern einer Datei** angezeigt. Die Eigenschaften sind so gut wie alle mit dem des *OpenFileDialog* vergleichbar. Zusätzlich verfügt der Dialog noch über die Eigenschaft *OverwritePrompt*, mit welcher festgelegt werden kann, ob geprüft werden soll, ob die **Datei bereits existiert** und dementsprechend eine Meldung angezeigt werden soll.
Der *FolderBrowserDialog* ist ein Dialog zum **Auswahl eines Ordners**. Die Eigenschaft *ShowNewFolderButton* legt fest, ob ein Button angezeigt werden soll, mit welchem ein **neuer Ordner erstellt** werden kann. Über die Eigenschaft *RootFolder* können wir das **Stammverzeichnis** auswählen. Hierfür benötigen wir die Enumeration *SpecialFolder*, welche viele Elemente enthält. Die gängigsten sind *Desktop* (physisches Verzeichnis, enthält alle Ordner des Computers), *DesktopDirectory* (logisches Verzeichnis, enthält alle auf dem Desktop befindende Ordner), *Favorites* (Favoriten-Verzeichnis), *MyComputer* (Arbeitsplatz) und *MyDocuments* (Eigene Dateien). Über die Eigenschaft *SelectedPath* können wir den vom Benutzer ausgewählten Ordner (mit Pfadangabe) abrufen.
Mit dem *ColorDialog* ist es möglich, den Benutzer aufzufordern, eine **Farbe auszuwählen**. Mit der Eigenschaft *AllowFullOpen* können wir unterbinden, dass der Dialog erweitert geöffnet werden kann, wodurch Farben mit Hilfe des RGB- und HSL-Farbmodells gemischt werden können. Über die Eigenschaft *Color* können wir die ausgewählte Farbe auswählen oder setzen. Die *Color*-Eigenschaft enthält ein Objekt der Klasse *Color* (*System.Drawing* Namensraum). Die Klasse (bzw. richtigerweise die Struktur) enthält zudem auch einige statische Objekte für die Auswahl einer Farbe (*Red*, *Blue*, *Lime*, …).
Der *FontDialog* ist ein Dialog zur **Auswahl der Schriftfamilie, Schriftgröße und des Schriftstils** (Fettdruck, Kursiv, Unterstrichen, Durchgestrichen). Die Schrifteinstellungen können über die Eigenschaft *Font* abgerufen oder gesetzt werden, welches ein *Font*-Objekt (*System.Drawing* Namensraum) enthält. Die einzelnen Bestandteile eines *Font*-Objekts können abgerufen, jedoch nicht zugewiesen werden. Zum Ändern einer einzelnen *Font*-Eigenschaft muss das komplette Objekt kopiert und dann über den Konstruktor der *Font*-Klasse geändert werden. Durch die Eigenschaft *ShowColor* können wir den Dialog um eine einfache **Farbauswahl** erweitern. Die dabei resultierende Farbe kann über die Eigenschaft *Color* abgerufen oder festgelegt werden.

Form1.cs

```
1  private void buttonÖffnen_Click(object sender, EventArgs e)
2  {
3      if (openFileDialog1.ShowDialog() == System.Windows.Forms.DialogResult.OK)
4          MessageBox.Show("Sie haben die folgende Datei gewählt:\n" + openFileDialog1.FileName);
5  }
6
7  private void buttonSpeichern_Click(object sender, EventArgs e)
8  {
9      if (saveFileDialog1.ShowDialog() == System.Windows.Forms.DialogResult.OK)
10         MessageBox.Show("Sie haben die folgende Datei gewählt:\n" + saveFileDialog1.FileName);
11 }
12
13 private void buttonOrdnerWählen_Click(object sender, EventArgs e)
14 {
15     if (folderBrowserDialog1.ShowDialog() == System.Windows.Forms.DialogResult.OK)
16         MessageBox.Show("Sie haben den folgenden Ordner gewählt:\n" +
   folderBrowserDialog1.SelectedPath);
17 }
18
19 private void buttonFarbeWählen_Click(object sender, EventArgs e)
20 {
21     // Farbe des Labels in den Dialog übernehmen
22     colorDialog1.Color = labelTest.ForeColor;
23
```

```
24          // Dialog anzeigen und gewählte Farbe ins Label übernehmen
25          if (colorDialog1.ShowDialog() == System.Windows.Forms.DialogResult.OK)
26              labelTest.ForeColor = colorDialog1.Color;
27      }
28
29      private void buttonFormatierungÄndern_Click(object sender, EventArgs e)
30      {
31          // Schriftformatierung des Labels in den Dialog übernehmen
32          fontDialog1.Font = labelTest.Font;
33
34          // Dialog anzeigen und gewählte Schriftformatierung ins Label übernehmen
35          if (fontDialog1.ShowDialog() == System.Windows.Forms.DialogResult.OK)
36              labelTest.Font = fontDialog1.Font;
37      }
```

Das große Computer ABC

C# lernen

Windows Forms: Kommunikation zwischen Formularen

Wenn wir eine Anwendung mit mehreren Formularen haben, gibt es das Problem, dass wir Daten oder Informationen **von einem zum anderen Formular** übertragen müssen. Wir erklären dies an Hand eines Beispiels: Formular 1 und Formular 2 hat jeweils ein Textfeld. Im Formular 1 geben wir einen Text ein und klicken auf einen Button, wodurch Formular 2 geöffnet wird und der Text, welcher in Formular 1 eingegeben wurde, im Textfeld erscheint. Doch wie funktioniert das? Wir könnten dem Textfeld-Steuerelement den Zugriffsmodifizierer *public* zuweisen. Dies wollen wir aber nicht machen, da es ein schlechter Programmierstil ist, Variablen *public* zu machen, wie wir ja bereits im Objektorientierungs-Kapitel erklärt haben. Also erstellen wir eine Eigenschaft, welche auf die *TextBox* zugreift. Somit können wir den *string* schon Mal von Formular 1 zu Formular 2 übertragen. Im Formular 2 gibt es ebenfalls einen Button, mit welchem wir das aktuelle Fenster schließen und den Text wieder zurück in das Formular 1 übertragen wollen. Hierfür registrieren wir das *FormClosing*-Event von Formular 2 über das Formular 1. Nun können wir über die Eigenschaft, welche wir im Formular 2 erstellt haben, den Text des Textfeldes abrufen. Doch es gibt ein Problem: Wir übernehmen den Text auch wenn wir das Fenster über den Schließen-Button des Fensters selbst (Programmleiste) schließen. Dies ist jedoch nicht gewünscht. Deshalb haben wir im Formular 2 noch eine weitere Eigenschaft (Datentyp *bool*) eingeführt, mit welcher wir prüfen können, ob das Fenster über den Button geschlossen wurde (siehe Beispiel).
Zum Öffnen eines Formulars können wir die Funktion *Show()* oder *ShowDialog()* aufrufen. Wird *ShowDialog()* verwendet, so können wir nicht zu dem öffnenden Fenster zurückkehren, das Fenster ist **modal**. Um ein Formular wieder zu schließen, können wir die Funktion *Hide()* oder *Close()* verwenden. Wird die *Close()*-Funktion aufgerufen, so wird das Formular zerstört und darf somit anschließend nicht mehr mit *Show()* oder *ShowDialog()* aufgerufen werden. Wird *Hide()* verwendet, so können wir nach dem Schließen des Fensters, das Formular **erneut öffnen**. Dabei bleiben Eingaben in Textfeldern etc. enthalten, da das Formular nicht geschlossen sondern nur „versteckt“ wird.

Form1.cs

```
1  private void buttonFormular2Öffnen_Click(object sender, EventArgs e)
2  {
3      Form2 oForm = new Form2();
4      oForm.sTextEingabe = textBoxEingabe.Text;
5      oForm.FormClosing += new FormClosingEventHandler(Form2_FormClosing);
6      oForm.Show();       // falls Dialog angezeigt werden sollte, kann auch ShowDialog()
   aufgerufen werden
7  }
8
9  private void Form2_FormClosing(object sender, FormClosingEventArgs e)
10 {
11     Form2 oForm = (Form2)sender;
12
13     // Prüfen ob Fenster durch den Klick auf den Button geschlossen wurde
14     if (oForm.bEingabeButton)
15         textBoxEingabe.Text = oForm.sTextEingabe;
16 }
```

Form2.cs

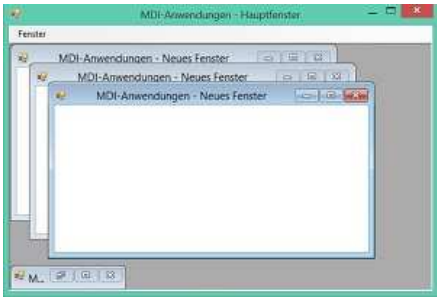
```
1  public bool bEingabeButton { get; set; }    // falls true, dann muss sTextEingabe in Form 1
   übernommen werden
2  public string sTextEingabe
3  {
4      get
5      {
6          return textBoxEingabe.Text;
7      }
8      set
9      {
10         textBoxEingabe.Text = value;
11     }
12 }
13
14 public Form2()
15 {
16     InitializeComponent();
17     bEingabeButton = false;
18 }
19
20 private void buttonZurückZuFormular1_Click(object sender, EventArgs e)
21 {
```

```
22      // Flag setzen und Fenster schließen (Form 1 hat das Event FormClosing registriert)
23      bEingabeButton = true;
24      Close();
25  }
```

Das große Computer ABC

C# lernen

Windows Forms: MDI-Anwendungen



Bei MDI-Anwendungen (Multiple Document Interface) haben wir immer ein Haupt-Formular und ein oder **mehrere Sub-Formulare**. Dadurch ist es möglich, mehrere Sub-Formulare (auch Frames genannt) in einem Fenster darzustellen. Alternative Konzepte, um mehrere Frames innerhalb eines Fensters darzustellen, sind **SDI** (Single Document Interface) und **TDI** (Tabbed Document Interface). Das heute gängigste Konzept ist TDI, welches z. B. in Editoren, Entwicklungsumgebungen und Browsern verwendet wird. Dies könnten wir zum jetzigen Zeitpunkt auch schon selbst realisieren (über TabControl und TabPage). Das SDI-Konzept wird z. B. im Bildbearbeitungsprogramm GIMP benutzt. Auch hier hätten wir aus dem jetzigen Wissensstand keine Probleme. MDI-Anwendungen wurden früher bei Office-Anwendungen (u. U. Microsoft Office) und anderen eingesetzt. Doch wie wird ein solches Konzept per Programmierung realisiert?
Um die MDI-Funktionalität nutzen zu können, müssen wir im Haupt-Formular die Eigenschaft *IsMdiContainer* auf *true* setzen. Das Frame wird über den Programmcode dynamisch erstellt und mit der Funktion *Show()* angezeigt. Bevor wir das Fenster anzeigen, müssen wir jedoch die Eigenschaft *MdiParent* des Frames auf das Objekt des Haupt-Formulars setzen. Hierfür kann zumeist das Schlüsselwort *this* verwendet werden. Die Eigenschaft *ActiveMdiChild* enthält das **aktiv ausgewählte Sub-Formular** oder *null*, falls kein Frame aktiv bzw. geöffnet ist. Die Eigenschaft *MdiChildren* enthält ein **Array von Formularen**, welche dem MDI-Container zugewiesen sind. Die Eigenschaft kann nur gelesen und nicht verändert werden. Die Zuordnung eines Frames zu der Eigenschaft des Haupt-Formulars erfolgt automatisch durch die Zuweisung der Eigenschaft *MdiParent* des Frames.

MainForm.cs

```
1  private void menuStripHauptfenster_ItemClicked(object sender, EventArgs e)
2  {
3      ToolStripMenuItem oMenuItem = (ToolStripMenuItem)sender;
4
5      if (oMenuItem == neuesÖffnenToolStripMenuItem)
6      {
7          SubForm oSubForm = new SubForm();
8          oSubForm.MdiParent = this;
9          oSubForm.Show();
10         aktuellesSchließenToolStripMenuItem.Enabled = true;
11         alleSchließenToolStripMenuItem.Enabled = true;
12     }
13     else if (oMenuItem == aktuellesSchließenToolStripMenuItem)
14     {
15         this.ActiveMdiChild.Close();
16         if (this.ActiveMdiChild == null)
17         {
18             aktuellesSchließenToolStripMenuItem.Enabled = false;
19             alleSchließenToolStripMenuItem.Enabled = false;
20         }
21     }
22     else if (oMenuItem == alleSchließenToolStripMenuItem)
23     {
24         foreach (Form oSubForm in this.MdiChildren)
25             oSubForm.Close();
26         aktuellesSchließenToolStripMenuItem.Enabled = false;
27         alleSchließenToolStripMenuItem.Enabled = false;
28     }
29     else if (oMenuItem == beendenToolStripMenuItem)
30     {
31         Close();
32     }
33 }
```

Das große Computer ABC

C# lernen

Windows Forms: Hintergrundarbeiten

Das Problem bei einer Anwendung mit grafischer Oberfläche ist, dass das Formular und die dahinter ausgeführten Funktionen innerhalb eines Threads arbeiten. Bei langwierigen Aufgaben haben wir also das Problem, dass die **GUI nicht mehr bedient werden kann**, solange eine Funktion abgearbeitet wird. Dies ist meistens jedoch nicht erwünscht und zudem schlechter Programmierstil. Deshalb sollte ein **eigener Thread** erstellt werden. Über das (unsichtbare) Steuerelement *BackgroundWorker* (Namensraum *System.ComponentModel*) können wir eine solche Funktionalität nutzen, welche in einer eigenen Klasse gekapselt ist und einige Vorteile bietet.
Mit der Eigenschaft *IsBusy* kann abgefragt werden, ob der *BackgroundWorker* aktuell arbeitet. Das Event *DoWork* wird ausgelöst, wenn die Funktion *RunWorkerAsync()* aufgerufen und somit der *BackgroundWorker* gestartet wird. Die im Event *RunWorkerCompleted* hinterlegte Ereignis-Funktion wird aufgerufen, wenn der *BackgroundWorker* **abgebrochen wurde oder dieser abgeschlossen wurde**. Um den *BackgroundWorker* abzubrechen, müssen wir die Funktion *CancelAsync()* aufrufen. Wichtig ist, dass hierdurch der *BackgroundWorker* nicht wirklich abgebrochen wird. Hierfür muss in der Event-Funktion des *DoWork*-Events die Eigenschaft *CancellationPending* abgerufen werden (siehe Beispiel). Das Abbrechen eines *BackgroundWorker*-Steuerelementes ist nur dann erlaubt, wenn die Eigenschaft *WorkerSupportsCancellation* auf *true* gesetzt ist. Ein Vorteil beim *BackgroundWorker* ist, dass wir über das Event *ProgessChanged* und die Funktion *ReportProgess()* einen **Fortschritt melden** und somit im Formular anzeigen können (z. B. Fortschritts-Balken aktualisieren). Diese Funktionalität kann nur genutzt werden, wenn *WorkerReportsProgess* auf *true* eingestellt ist. Dieser Text enthält viele Namen von Eigenschaften, Funktionen und Events, weshalb es umso wichtiger ist, sich das dazugehörige Beispiel anzuschauen, um den Sinn und Zweck zu verstehen.

Form1.cs

```
1   private void buttonStarten_Click(object sender, EventArgs e)
2   {
3       // Abbruch-Button freigeben und Fortschrittsbalken zurücksetzen
4       buttonAbbrechen.Enabled = true;
5       buttonStarten.Enabled = false;
6       progressBarFortschritt.Value = 0;
7
8       // BackgroundWorker starten
9       backgroundWorkerFortschritt.RunWorkerAsync();
10  }
11
12  private void buttonAbbrechen_Click(object sender, EventArgs e)
13  {
14      // Start-Button freigeben
15      buttonAbbrechen.Enabled = false;
16      buttonStarten.Enabled = true;
17
18      // BackgroundWorker abbrechen
19      backgroundWorkerFortschritt.CancelAsync();
20  }
21
22  private void backgroundWorkerFortschritt_DoWork(object sender, DoWorkEventArgs e)
23  {
24      for (int i = 1; i <= 100; i++)
25      {
26          // Prüfen ob Vorgang abgebrochen wurde (durch CancelAsync())
27          if (backgroundWorkerFortschritt.CancellationPending)
28          {
29              e.Cancel = true;
30              return;
31          }
32
33          // aktuelle Position in Fortschrittsbalken anzeigen und aktuelle Zahl speichern
34          backgroundWorkerFortschritt.ReportProgress(i);
35          e.Result = i;
36
37          // 50 ms warten
38          Thread.Sleep(50);
39      }
40  }
41
42  private void backgroundWorkerFortschritt_ProgressChanged(object sender,
    ProgressChangedEventArgs e)
43  {
44      // Wert für Fortschrittsbalken setzen
```

```
45      progressBarFortschritt.Value = e.ProgressPercentage;
46  }
47
48  private void backgroundWorkerFortschritt_RunWorkerCompleted(object sender,
    RunWorkerCompletedEventArgs e)
49  {
50      // Meldung ausgeben
51      if (e.Cancelled)
52          MessageBox.Show("Der Vorgang wurde abgebrochen!");
53      else
54          MessageBox.Show("Der erreichte Wert war " + (int)e.Result + "!");   // gibt immer 100
    aus
55
56      // Start-Button freigeben
57      buttonAbbrechen.Enabled = false;
58      buttonStarten.Enabled = true;
59  }
```

Das große Computer ABC

C# lernen

Tastatur- und Mausereignisse: Tastaturereignisse

Jedes Steuerelement welches von der Basisklasse *Control* abstammt (unter anderem auch die *Form*-Klasse) besitzt Tastaturereignisse. Hier sind *KeyDown*, *KeyUp* und *KeyPress* zu nennen. Die Events werden normalerweise immer alle nacheinander in folgender Reihenfolge ausgelöst: KeyDown, KeyPress und zum Schluss KeyUp.
Das *KeyDown*-Event tritt ein, wenn eine **Taste gedrückt wird**. *KeyUp* dagegen löst aus, sobald eine **Taste losgelassen wird**. Als Event-Argumente werden dem *KeyDown*- und *KeyUp*-Ereignis *KeyEventArgs* übergeben. Über die Eigenschaften *Alt*, *Control* und *Shift* kann abgefragt werden, ob die Tasten Alt, Steuerung / Control und Shift gedrückt sind. Mit der Eigenschaft *KeyValue* können wir den Code (ASCII-Code, UTF8-Code, …) der gedrückten Taste abfragen. Hierbei ist zu beachten, dass nicht alle Tasten darstellbar sind. In den meisten Fällen (z. B. für Buchstaben) kann der Wert der Eigenschaft in die *char* gecastet werden. Die *KeyCode*-Eigenschaft enthält einen Wert der Enumeration *Keys*.
Das *KeyPress*-Event wird ausgelöst, wenn die **Taste eines Zeichens, die Leertaste oder die Rücktaste gedrückt wurde**. Hier werden als Event-Argumente die *KeyPressEventArgs* übergeben. Durch die *KeyChar*-Eigenschaft können wir das Zeichen (Datentyp *char*) der gedrückten Taste abrufen. Im Gegensatz zum *KeyDown*- und *KeyUp*-Event wird nicht immer ein Großbuchstabe zurückgegeben. Hier wird lediglich ein Großbuchstabe zurückgegeben, falls die Feststell- oder Umschalt-Taste gedrückt ist, andernfalls wird ein Kleinbuchstabe weitergeben. Für die meisten Fälle wird deshalb das *KeyPressed*-Ereignis eingesetzt, da es für den Programmierer die einfachste Schnittstelle anbietet. Über die Eigenschaft *Handled* können wir dem Steuerelement mitteilen, dass das Zeichen, welches in den Event-Argumenten in der Eigenschaft *KeyChar* festgelegt wurde, **bereits behandelt** wurde und nicht mehr weiter bearbeitet werden soll. Hierdurch könnten wir also **abfangen**, dass bestimmte Zeichen in eine *TextBox* eingegeben werden können, indem wir *Handled* auf *true* setzen.

Form1.cs

```
1  private void Form1_KeyDown(object sender, KeyEventArgs e)
2  {
3      labelKeyDown.Text = "";
4
5      if (e.Alt)
6          labelKeyDown.Text += "ALT ";
7      if (e.Control)
8          labelKeyDown.Text += "STRG ";
9      if (e.Shift)
10         labelKeyDown.Text += "SHIFT ";
11
12     if (!char.IsControl((char)e.KeyValue))
13         labelKeyDown.Text += ((char)e.KeyValue).ToString();
14 }
15
16 private void Form1_KeyUp(object sender, KeyEventArgs e)
17 {
18     labelKeyUp.Text = "";
19
20     if (e.Alt)
21         labelKeyUp.Text += "ALT ";
22     if (e.Control)
23         labelKeyUp.Text += "STRG ";
24     if (e.Shift)
25         labelKeyUp.Text += "SHIFT ";
26
27     if (!char.IsControl((char)e.KeyValue))
28         labelKeyUp.Text += ((char)e.KeyValue).ToString();
29 }
30
31 private void Form1_KeyPress(object sender, KeyPressEventArgs e)
32 {
33     if (!Char.IsControl(e.KeyChar))
34         labelKeyPress.Text = e.KeyChar.ToString();
35     else
36         labelKeyPress.Text = "";
37 }
```

Das große Computer ABC

C# lernen

Tastatur- und Mausereignisse: Erweiterte Tastaturereignisse

Wenn Sie nun einige Tests mit den „normalen“ Tastaturereignissen gemacht haben, wird Ihnen auffallen, dass die Events in einigen **speziellen Fällen nicht ausgelöst werden**. Hier sind z. B. die Pfeiltasten oder die Tab-Taste zu nennen. Dies liegt daran, dass die Tab-Taste als Beispiel für das Springen von einem Textfeld zum nächsten genutzt wird und somit eine Sonderfunktion besitzt.
Wie können wir also solche Tasten abfangen? Grundsätzlich ist dies nur mit „höherer“ Programmiertechnik möglich, denn wir müssen hierzu eine Funktion, welche der Klasse *Control* angehört, **überschreiben**. Dazu benötigen wir das Schlüsselwort *override*, wodurch wir dem Compiler mitteilen, dass er nicht die Funktion der Basisklasse aufrufen soll, sondern die „überschriebene“ Funktion. Wollen wir innerhalb dieser Funktion doch die **Funktion der Basisklasse aufrufen**, so verwenden wir das Schlüsselwort *base,* gefolgt vom Funktionsaufruf. Die Funktion, welche zum Verarbeiten der Tasten-Anschläge verwendet wird heißt *ProcessCmdKey()*, welche eine Nachricht (Klasse *Message*, für uns jedoch unbedeutend) und die gedrückte Taste (Enumeration *Keys*) übergeben werden. Der Rückgabewert ist ein *bool*, welcher angibt, ob der Tasten-Anschlag vom Steuerelement verarbeitet wurde.
Zu beachten ist, dass wir im Beispiel die Funktion des **kompletten Formulars** überschrieben haben. In einigen Fällen ist es jedoch u. U. nur gewollt, die Funktion eines einzelnen Steuerelements zu überschreiben. Hierfür müssen wir eine **eigene Klasse erstellen**, welche als Basisklasse die Klasse des Steuerelements enthält. In dieser erstellten Klasse können wir dann die Funktion überschreiben. Jedoch muss beachtet werden, dass in einem solchen Fall im Designer nicht die sonst verwendete Klasse des Steuerelements gewählt werden darf, sondern die von uns erstellte Klasse des Steuerelements genutzt werden muss.

Form1.cs

```
1  protected override bool ProcessCmdKey(ref Message msg, Keys keyData)
2  {
3      switch (keyData)
4      {
5          // da wir eine System-Methode für Tastaturanschläge überschrieben haben, werden die
   Pfeiltasten
6          // für links und rechts in den Textboxen nicht mehr funktionieren!
7          case Keys.Left:
8              labelInfo.Text = "linke Pfeiltaste";
9              return true;
10         case Keys.Right:
11             labelInfo.Text = "rechte Pfeiltaste";
12             return true;
13
14         case Keys.Tab:
15             // die Tab-Taste wird zwar ebenfalls abgefangen, jedoch rufen wir über die
   base.ProcessCmdKey()
16             // die Funktion der Basis-Klasse auf, wodurch die Tab-Funktionalität weiterhin
   gegeben ist
17             labelInfo.Text = "Tabulator";
18             return base.ProcessCmdKey(ref msg, keyData);
19
20         default:
21             // alle anderen Tastenanschläge werden an die Basis-Klasse weitergegeben
22             return base.ProcessCmdKey(ref msg, keyData);
23     }
24 }
```

Das große Computer ABC

C# lernen

Tastatur- und Mausereignisse: Mausereignisse für Tasten

Um Tastendrücke der Maus festzustellen, gibt es die Ereignisse *MouseClick*, *MouseDoubleClick*, *MouseDown* und *MouseUp* sowie *Click* und *DoubleClick*. Bei einem einfachen Mausklick lösen die Events in folgender Reihenfolge aus: *MouseDown*, *Click*, *MouseClick* und *MouseUp*. Bei einem Doppelklick hingegen ist die Reihenfolge wie folgt: *MouseDown*, *Click*, *MouseClick*, *MouseUp*, *MouseDown*, *DoubleClick*, *MouseDoubleClick*, *MouseUp*.
Die Events *MouseClick*, *MouseDoubleClick*, *MouseDown* und *MouseUp* lösen **unabhängig von der gedrückten Taste** aus, d. h. das Ereignis löst sowohl beim Linksklick als auch beim Rechtsklick aus sowie bei einem Klick auf eine Spezialmaustaste. Wie der Name schon vermuten lässt, löst *MouseClick* bei einem **einfachen Mausklick** aus. *MouseDoubleClick* dagegen bei einem **Doppelklick**. *MouseDown* löst aus, sobald **eine Maustaste gedrückt** wurde. *MouseUp* hingegen sobald die **Maustaste wieder losgelassen** wurde. Zu beachten ist (wie bereits oben bei der Reihenfolge genannt), dass bei einem Doppelklick das *MouseDown*- und *MouseUp*-Event doppelt ausgelöst wird. Allen vier Funktionen werden Event-Argumente der Klasse *MouseEventArgs* übergeben. Über die Eigenschaft *Button* können wir einen Wert der Enumeration *MouseButtons* abrufen, welche die **geklickte Taste** enthält. Mit der Eigenschaft *Clicks* können die **Anzahl der Tastendrücke** abgerufen werden.
Die Ereignisse *Click* und *DoubleClick* sind Ereignisse, welche ebenfalls der *Control*-Klasse angehören, jedoch **nur in bestimmten Fällen auslösen**. Ob ein Ereignis ausgelöst wird, hängt von der gedrückten Taste und dem Steuerelement ab. So löst bei einem *Button*, einer *CheckBox* oder einem *RadioButton* das *Click*-Event nur aus, wenn die linke Maustaste gedrückt wurde. Das *DoubleClick*-Event löst bei diesen Steuerelementen nie aus. Bei den Steuerelementen *ListBox*, *ComboBox*, *TextBox* und *NumericUpDown* werden hingegen die Ereignisse *Click* und *DoubleClick* ausgelöst, wenn die linke Maustaste gedrückt wurde. Beim Steuerelement *ProgressBar* wird das *Click*-Event unabhängig von der Maustaste ausgelöst. Bei den Steuerelementen *Form*, *Label*, *Panel*, *GroupBox*, *PictureBox* und *TabPage* werden sowohl das *Click*- als auch das *DoubleClick*-Ereignis unabhängig von der Maustaste ausgelöst.

Form1.cs

```
1  private void label_MouseClick(object sender, MouseEventArgs e)
2  {
3      string sTaste = "";
4
5      switch (e.Button)
6      {
7          case System.Windows.Forms.MouseButtons.Left:
8              sTaste = "linke Taste";
9              break;
10         case System.Windows.Forms.MouseButtons.Right:
11             sTaste = "rechte Taste";
12             break;
13         case System.Windows.Forms.MouseButtons.Middle:
14             sTaste = "mittlere Taste";
15             break;
16         case System.Windows.Forms.MouseButtons.XButton1:
17             sTaste = "zusätzliche Taste 1";
18             break;
19         case System.Windows.Forms.MouseButtons.XButton2:
20             sTaste = "zusätzliche Taste 2";
21             break;
22     }
23
24     MessageBox.Show("Sie haben " + e.Clicks + " Mal geklickt (" + sTaste + ")!");
25 }
```

Das große Computer ABC

C# lernen

Tastatur- und Mausereignisse: Mausereignisse für Bewegung

Um auf Mausbewegungen zu reagieren, können wir die Events *MouseEnter*, *MouseLeave* und *MouseMove* nutzen.
Das *MouseEnter*-Ereignis tritt ein, sobald der **Mauszeiger das Steuerelement betritt**. **Verlässt die Maus das Steuerelement** wieder, so wird das *MouseLeave*-Ereignis ausgelöst. Diese beiden Ereignisse haben keine besonderen Event-Argumente (Basisklasse *EventArgs* wird verwendet).
Das *MouseMove*-Ereignis tritt ein, sobald sich die **Maus bewegt**. Hier werden, wie bei den Mausereignissen für Tastendrücke auch, die *MouseEventArgs* als Event-Argumente übergeben. Über die Eigenschaften *X* und *Y* können wir die **Position** innerhalb des Steuerelements abrufen.

Form1.cs

```
1  private void panelPosition_MouseEnter(object sender, EventArgs e)
2  {
3      labelInfo.Text = "Maus befindet sich im Panel";
4  }
5
6  private void panelPosition_MouseLeave(object sender, EventArgs e)
7  {
8      labelInfo.Text = "Maus befindet sich nicht im Panel";
9  }
10
11 private void panelPosition_MouseMove(object sender, MouseEventArgs e)
12 {
13     labelPositionX.Text = e.X.ToString();
14     labelPositionY.Text = e.Y.ToString();
15 }
```

Das große Computer ABC

C# lernen

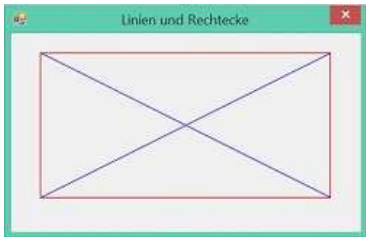
Grafische Programmierung: Einführung in die Grafik-Programmierung

Um in die grafische Programmierung einzusteigen, müssen wir als erstes einige grundlegende Informationen über die sogenannte **GDI-Schnittstelle** (Graphic Device Interface) lernen. C# bietet uns für die grafische Programmierung die **GDI+-Bibliotheken** (dabei steht das Pluszeichen für eine Weiterentwicklung der ursprünglichen GDI-Bibliotheken), welche sich im Namensraum *System.Drawing* und deren untergeordneten Namensräume befinden. Das „Zeichnen“ wird zumeist über das *Paint*-Ereignis des jeweiligen Steuerelements durchgeführt. Beim Aufruf der Funktion *Invalidate()* wird der **gezeichnete Inhalt für ungültig** erklärt und das *Paint*-Ereignis ausgelöst. Wird die Funktion ohne Parameter aufgerufen, so wird das ganze Steuerelement für ungültig erklärt. Wir können der Funktion auch ein Objekt der Struktur *Rectangle* (dazu gleich mehr) übergeben, wodurch wir den Bereich, welcher für ungültig erklärt und neu gezeichnet werden soll, angeben können. Dem Paint-Ereignis werden Event-Argumente der Klasse *PaintEventArgs* übergeben, welche die Eigenschaften *ClipRectangle* und *Graphics* enthalten. *ClipRectangle* ist ein Objekt der *Rectangle*-Struktur und enthält den **Bereich, welcher neu gezeichnet** werden soll. Dieser Wert sollte abgerufen werden, wenn *Invalidate()* mit dem *Rectangle*-Parameter aufgerufen wird. Die Eigenschaft *Graphics* enthält ein Objekt der gleichnamigen Klasse. Über diese Eigenschaft werden wir später verschiedene Funktionen zum Zeichnen aufrufen.
Für das Zeichnen sind einige Hilfsklassen von Nöten. Die Struktur *Color* haben wir ja bereits kennengelernt. Die *Color*-Struktur enthält einige statische Objekte zum Abrufen von **standardisierten Farben**. Über die Funktion *FromArgb()* können wir eine Farbe auch manuell über das **RGB-Modell** mischen. Die Klasse *Brushes* enthält ebenfalls statische Objekte für einige Standardfarben. Hier ist ein Mischen jedoch nicht möglich. Die *Brushes*-Klasse stellt sozusagen den **Pinsel zum Zeichnen** zur Verfügung. Die *Pen*-Klasse ist eine Klasse, welche den **Stift zum Zeichnen von Linien und Kurven** zur Verfügung stellt. Der Konstruktor ist mehrfach überladen und kann daher mit der Farbe (*Brushes*-Klasse oder *Color*-Struktur) und bei Bedarf auch zusätzlich mit der Breite aufgerufen werden. Die übergebenen Werte werden in den Eigenschaften *Brush* bzw. *Color* und *Width* hinterlegt. Die Point-Struktur wird verwendet, um einen **Punkt in einem Koordinatensystem** darzustellen. Dem Konstruktor werden die X- und Y-Koordinate übergeben. Diese können im Nachhinein über die gleichnamigen Eigenschaften gesetzt und abgerufen werden. Wollen wir keine Koordinate sondern eine Größe darstellen, so können wir die *Size*-Struktur verwenden, welche vom Aufbau der *Point*-Struktur ähnelt. Die X-Koordinate wird dabei durch die Breite (Eigenschaft *Width*) und die Y-Koordinate durch die Höhe (Eigenschaft *Height*) ersetzt. Mit Hilfe der *Rectangle*-Struktur können wir ein Rechteck mit **Positionierung und Größe** programmiertechnisch darstellen. Der Konstruktor wird mit vier Ganzzahlen (X-Koordinate, Y-Koordinate, Breite, Höhe) oder einem *Point*- und einem *Size*-Objekt aufgerufen. Zum Abrufen der einzelnen Werte können wir die Eigenschaften *X*, *Y*, *Width* und *Height* sowie *Left*, *Top*, *Right* und *Bottom* verwenden.

Das große Computer ABC

C# lernen

Grafische Programmierung: Linien und Rechtecke



Um eine **einzelne Linie** zu zeichnen, benötigen wir die *DrawLine()*-Funktion der *Graphics*-Klasse. Der Funktion werden drei Parameter übergeben. Dabei ist der erste Parameter ein Objekt der *Pen*-Klasse. Als zweiter und dritter Parameter werden Objekte der *Point*-Struktur übergeben, um die Start- und Endposition der Linie festzulegen. Mit der Funktion *DrawLines()* können wir eine Linie mit **mehreren Liniensegmenten** zeichnen.
Ein **einzelnes Rechteck** wird mit der *DrawRectangle()*-Funktion gezeichnet. Hier wird ebenfalls ein Objekt der *Pen*-Klasse und des Weiteren ein Objekt der *Rectangle*-Struktur übergeben. Sollen **mehrere Rechtecke** gezeichnet werden, so benötigen wir die Funktion *DrawRectangles(),* welcher ein *Pen*-Objekt und ein Array von *Rectangle*-Objekten übergeben wird.

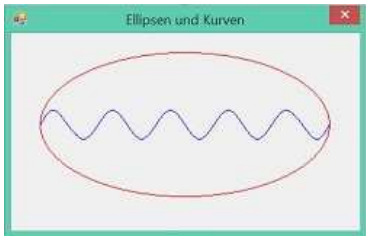
Form1.cs

```
1  private void Form1_Paint(object sender, PaintEventArgs e)
2  {
3      e.Graphics.DrawRectangle(new Pen(Brushes.Red), new Rectangle(30, 20, 300, 150));
4
5      // Linien als "Kreuz" im Rechteck
6      e.Graphics.DrawLine(new Pen(Brushes.Blue), new Point(30, 20), new Point(330, 170));
7      e.Graphics.DrawLine(new Pen(Brushes.Blue), new Point(30, 170), new Point(330, 20));
8  }
```

Das große Computer ABC

C# lernen

Grafische Programmierung: Ellipsen und Kurven



Eine **Ellipse** wird mit der Funktion *DrawEllipse()* gezeichnet. Als Übergabeparameter müssen wir den Stift (*Pen*-Klasse) und das Rechteck (*Rectangle*-Struktur) übergeben. Hierbei wird natürlich nicht das Rechteck selbst, sondern die Ellipse oder der Kreis „um das Rechteck herum“ gezeichnet. Das *Rectangle*-Objekt dient also lediglich als Rahmen.
Um eine **Kurve** zu zeichnen, können wir die Funktion *DrawCurve()* der *Graphics*-Klasse nutzen. Auch der *DrawCurve()*-Funktion wird als erster Parameter ein *Pen*-Objekt übergeben. Als zweiter Parameter wird ein Array von einzelnen Koordinatenpunkten (*Point*-Struktur) übergeben. Bei der gezeichneten Kurve handelt es sich um die sogenannte „Cardinal-Splinekurve“. Im Beispiel zeichnen wir mit Hilfe einer Berechnung eine Sinuskurve. Jedoch geht es im Beispielprogramm mehr um die grafischen Funktionen als um die Berechnung der einzelnen Punkte.

Form1.cs

```
1  private void Form1_Paint(object sender, PaintEventArgs e)
2  {
3      Point[] aKurvenPunkte = new Point[12];
4
5      e.Graphics.DrawEllipse(new Pen(Brushes.Red), new Rectangle(30, 20, 300, 150));
6
7      // erster Punkt der Kurve
8      aKurvenPunkte[0] = new Point(30, 20 + 75);
9
10     // Punkte des Arrays (Index 1 bis 10) mit Werten für Sinus-Kurve füllen
11     for (int i = 1; i < 11; i++)
12     {
13         // x = 30 (Startpunkt) - 15 (Versatz durch halbe Sinuswelle am Anfang) + aktueller
   Punkt * 30
14         // y = 20 (Startpunkt) + 75 (Mitte der Ellipse) +/- 15 (bei jedem Mal, wenn Rest 0
   ist Plus andernfalls Minus)
15         aKurvenPunkte[i] = new Point(30 - 15 + (i * 30), 20 + 75 + (i % 2 == 0 ? 15 : -15));
16     }
17
18     // letzter Punkt der Kurve
19     aKurvenPunkte[11] = new Point(30 + 300, 20 + 75);
20
21     e.Graphics.DrawCurve(new Pen(Brushes.Blue), aKurvenPunkte);
22 }
```

Das große Computer ABC

C# lernen

Grafische Programmierung: Schrift

Um einen Schriftzug dynamisch mit der GDI-Schnittstelle zu zeichnen (und nicht über ein Label), benötigen wir die Funktion *DrawString()*. Der Funktion wird die **Zeichenkette** (*string*), die Schriftformatierung (*Font*-Klasse), die Farbe (*Brushes*-Klasse) und die Koordinaten (*PointF*-Struktur) übergeben. Die *PointF*-Struktur ist im Gegensatz zur *Point*-Struktur für **Gleitkommazahlen** und nicht nur für Ganzzahlen geeignet. Der restliche Aufbau ist identisch.
Die *Font*-Klasse stellt uns eine Reihe von **Schriftformatierungen** zur Verfügung, so können wir die Schriftfamilie (Eigenschaft *FontFamily*), Schriftgröße (Eigenschaft *Size*), Maßeinheit der Schriftgröße (Eigenschaft *Unit*) und den Schriftstil (Eigenschaft *Style*) einstellen. Die gerade genannten Eigenschaften können jedoch **nicht verändert**, sondern nur abgerufen werden, d. h. **das Setzen der Werte passiert lediglich über den Konstruktor**. Deshalb ist der Konstruktor auch mehrfach überladen. Als ersten Parameter übergeben wir die **Schriftfamilie**, wobei wir diese als Zeichenkette oder als Objekt der *FontFamily*-Klasse übergeben können. Dem Konstruktor der *FontFamily*-Klasse kann ebenfalls der Name der Schriftfamilie übergeben werden. Über die statische Eigenschaft *Families* der Klasse *FontFamily* können wir ein **Array von verfügbaren Schriftfamilien** abrufen und ebenfalls verwenden. Als zweiter Parameter wird die **Größe der Schrift** übergeben. Hierbei ist die Größe jedoch abhängig von der verwendeten Maßeinheit (dazu gleich mehr). Ein Aufruf mit diesen zwei Parametern würde bereits genügen. Als dritten optionalen Parameter übergeben wir einen Wert der Enumeration *FontStyle*, welche den **Schriftstil** beschreibt: *Bold* (Fett-Druck), *Italic* (Kursiv), *Regular* (Standardeinstellung), *Strikeout* und *Underline*. Die einzelnen Werte sind Bit-Masken und können daher bit-weise verodert werden (um z. B. einen kursiven und unterstrichenen Text darzustellen). Als vierter und weiterer optionaler Parameter können wir einen Wert der *GraphicsUnit*-Enumeration übergeben, wodurch wir die **Maßeinheit der Schriftgröße** festlegen. Hierbei stehen uns folgende Werte zur Verfügung: *Document* (1/300 Zoll), *Inch* (Zoll), *Millimeter*, *Pixel* und *Point* (1/72 Zoll). Standardeinstellung ist die Einheit *Point* (Abkürzung pt).

Form1.cs

```
1  private void Form1_Paint(object sender, PaintEventArgs e)
2  {
3      e.Graphics.DrawString("Hallo Welt!", new Font(new FontFamily("Times"), 20, FontStyle.Bold
   | FontStyle.Italic), Brushes.Orange, new PointF(40, 40));
4  }
```

Das große Computer ABC

C# lernen

Grafische Programmierung: Bilder

Ein Bild zeichnen wir mit Hilfe der Funktion *DrawImage()*. Der Funktion übergeben wir ein *Image*-Objekt und vier Ganzzahlen zum Festlegen der X-Position, Y-Position, Breite und Höhe des Bildes. Alternativ kann der Funktion anstatt der Ganzzahlen auch ein *Rectangle*-Objekt übergeben werden. Um ein *Image*-Objekt aus einer Datei zu laden, können wir die statische Funktion *FromFile()* der *Image*-Klasse aufrufen. Wollen wir ein Icon (z. B. aus einer ico-Datei) zeichnen, so können wir die Funktion *DrawIcon()* verwenden. Hierbei ist ein Aufruf nur mit Hilfe des *Rectangle*-Objekts möglich.

Form1.cs

```
1  private void Form1_Paint(object sender, PaintEventArgs e)
2  {
3      // Bilder wird verzerrt (normalerweise wäre das Logo quadratisch)
4      e.Graphics.DrawImage(Image.FromFile("Logo.png"), 80, 40, 180, 140);
5  }
```

Das große Computer ABC

C# lernen

Grafische Programmierung: Drucken

Wollen wir von unserem Programm etwas **ausdrucken**, so müssen wir uns ebenfalls mit der grafischen Programmierung auseinandersetzen. Das benötigte *PrintDocument*-Objekt (welches ein unsichtbares Steuerelement ist und im Werkzeugkasten ausgewählt werden kann) befindet sich hierbei im Namensraum *System.Drawing.Printing*.
Natürlich könnten wir einfach nur unser Dokument ausdrucken, doch wir wollen unserem Benutzer auch die **Auswahl seines Druckers und deren Einstellungen** ermöglichen. Hierfür benötigen wir den Dialog *PrintDialog* (*System.Windows.Forms* Namensraum). Über die Eigenschaften *AllowCurrentPage*, *AllowSelection* und *AllowSomePages* können wir die Auswahl der Felder „Aktuelle Seite“, „Markierung“ und „Seiten: x“ aktivieren bzw. deaktivieren. Der Standardwert der Eigenschaften ist *false*, wodurch die Felder deaktiviert sind. Über die Eigenschaft *Document* legen wir das Objekt des zu druckenden Dokuments (*PrintDocument*) fest. Die Eigenschaft *UseEXDialog* sollte auf *true* gesetzt werden, da es andernfalls Probleme mit der Darstellung des Dialogs auf x64-Betriebssystemen geben kann. Dieses Problem wurde bereits von Microsoft bestätigt.
Nun wollen wir uns die Klasse *PrintDocument* etwas genauer anschauen: Die Eigenschaft *DocumentName* legt den **Namen des Dokuments** fest (dieser wird z. B. beim Drucken auf dem Display des Druckers oder in der Druckerwarteschlange des Computers angezeigt). Die Eigenschaft *OriginAtMargins* legt fest, ob sich die **Positionsangaben** beim Zeichnungsvorgang auf die linke obere Ecke des Blattes (Wert *false*) oder die linke obere Ecke der Seitenränder (Wert *true*) bezieht. Die Funktion *Print()* startet den Druckvorgang. Die *PrintDocument*-Klasse besitzt drei wichtige Ereignisse: *BeginPrint*, *EndPrint* und *PrintPage*. *BeginPrint* tritt auf bevor der eigentliche Druckvorgang beginnt und somit direkt nach dem Aufruf der *Print()*-Funktion. *EndPrint* tritt auf nachdem der Druckvorgang abgeschlossen ist. Beiden Ereignissen werden *PrintEventArgs* als Event-Argumente übergeben. Hierbei können wir über die Eigenschaft *Cancel* den **Druckvorgang abbrechen** oder abfragen, ob dieser abgebrochen wurde. Das *PrintPage*-Ereignis löst aus, sobald eine Seite gedruckt werden soll. Hier werden Event-Argumente der Klasse *PrintPageEventArgs* übergeben. Auch hier gibt es die Eigenschaft *Cancel*. Die Eigenschaft *Graphics* stellt, wie beim *Paint*-Ereignis auch, das Grafik-Objekt zur Verfügung, welches wir zum Zeichnen benötigen. Mit der Eigenschaft *HasMorePages* können wir festlegen, ob noch eine **weitere Seite** gedruckt werden soll (z. B. weil der Inhalt nicht auf eine Seite passt). Ist dieser Wert gesetzt, so wird die Ereignis-Funktion des *PrintPage*-Events erneut aufgerufen.
Doch wie erhalten wir die Information wie groß die Seite bzw. der dazugehörige Druckbereich ist? Hierfür gibt es die Eigenschaften *PageBounds* und *MarginBounds*. *PageBounds* enthält die **Größe des Papiers**. *MarginBounds* enthält hingegen die Breite und Höhe des **Druckbereichs** und die Position der Seitenränder. Dabei geben beide Eigenschaften ein *Rectangle*-Objekt zurück. Mit Hilfe dieser beiden Eigenschaften können wir z. B. errechnen, ob der gewünschte Inhalt auf eine Seite passt und ggf. *HasMorePages* auf *true* setzen. Das Drucken von Inhalten ist in einigen Umständen kompliziert. So erfordert es auch einige mathematische Kenntnisse und Berechnungen. Ein Beispiel hierfür ist, dass *DrawString()* die Zeichenkette immer fortlaufend zeichnet und somit ohne Zeilenumbruch. Wollen wir jedoch einen mehrzeiligen Text ausdrucken, so ist es erforderlich zu prüfen, wie viel von dem Text in eine Zeile passt und diesen dann aufzuteilen. Wir haben im Beispiel darauf jedoch verzichtet und versucht, das Beispiel so einfach wie möglich zu halten.

Form1.cs

```
1   private void buttonDrucken_Click(object sender, EventArgs e)
2   {
3       if (printDialog1.ShowDialog() == System.Windows.Forms.DialogResult.OK)
4       {
5           // Druckauftrag senden (Ereignis PrintPage wird aufgerufen)
6           printDocument1.Print();
7       }
8   }
9
10  private void printDocument1_BeginPrint(object sender, PrintEventArgs e)
11  {
12      if (MessageBox.Show("Wollen Sie den Druck-Auftrag wirklich starten?",
13                          "Drucken starten?",
14                          MessageBoxButtons.YesNo,
15                          MessageBoxIcon.Question) == System.Windows.Forms.DialogResult.No)
16      {
17          // falls nein gedrückt wird, Vorgang abbrechen
18          e.Cancel = true;
19      }
20      else
21          buttonDrucken.Enabled = false;
22  }
23
24  private void printDocument1_PrintPage(object sender, PrintPageEventArgs e)
25  {
26      Point[] aKurvenPunkte = new Point[12];
27
28      // siehe 11.2 Linien und Rechtecke
```

```
29      e.Graphics.DrawRectangle(new Pen(Brushes.Red), new Rectangle(30, 20, 300, 150));
30      e.Graphics.DrawLine(new Pen(Brushes.Blue), new Point(30, 20), new Point(330, 170));
31      e.Graphics.DrawLine(new Pen(Brushes.Blue), new Point(30, 170), new Point(330, 20));
32
33      // siehe 11.3 Ellipsen und Kurven
34      e.Graphics.DrawEllipse(new Pen(Brushes.Red), new Rectangle(30, 220, 300, 150));
35      aKurvenPunkte[0] = new Point(30, 220 + 75);
36      for (int i = 1; i < 11; i++)
37          aKurvenPunkte[i] = new Point(30 - 15 + (i * 30), 220 + 75 + (i % 2 == 0 ? 15 : -15));
38      aKurvenPunkte[11] = new Point(30 + 300, 220 + 75);
39      e.Graphics.DrawCurve(new Pen(Brushes.Blue), aKurvenPunkte);
40
41      // siehe 11.4 Schrift
42      e.Graphics.DrawString("Hallo Welt!", new Font(new FontFamily("Times"), 20, FontStyle.Bold
    | FontStyle.Italic), Brushes.Orange, new PointF(40, 420));
43
44      // siehe 11.5 Bilder
45      e.Graphics.DrawImage(Image.FromFile("Logo.png"), 80, 480, 180, 140);
46  }
47
48  private void printDocument1_EndPrint(object sender, PrintEventArgs e)
49  {
50      if (!e.Cancel)
51          MessageBox.Show("Die Testseite wurde erfolgreich gedruckt!");
52      buttonDrucken.Enabled = true;
53  }
```

Das große Computer ABC

C# lernen

WPF: Grundlagen

Windows Presentation Foundation (kurz WPF) ist eine **Erweiterung** des .NET-Frameworks, welches uns eine **neue grafische Oberfläche** bietet. WPF ist erst ab der Framework-Version 3.0 verfügbar. Bei der Erstellung einer Anwendung mit der WPF-Oberfläche müssen wir den Typ **„WPF-Anwendung“** in Visual Studio auswählen.
Beim Starten einer leeren WPF-Applikation fällt uns kein großer Unterschied im Vergleich zu Windows Forms auf. Auch hier gibt es einen Fensterrahmen mit Titel und den Kontroll-Boxen. Dies liegt daran, da es sich hierbei um den Standardaufbau von Windows-Fenstern handelt. Doch wenn wir uns das Programm bzw. den Designer genauer anschauen fällt uns auf, dass sich vor allem das **Designen unserer Applikation** komplett geändert hat. Hier gibt es nicht wie bei Windows Forms Objekte, die im „Designer“ in Form von Steuerelementen auf das Formular gezogen und über das Eigenschaftsfenster modifiziert werden. Vielmehr wird bei WPF ein Texteditor für den sogenannten **XAML-Code** (Extensible Application Markup Language) angezeigt. Natürlich gibt es auch bei WPF einen Designer, ein Eigenschaftsfenster und den Werkzeugkasten.
Was unterscheidet sich noch von Windows Forms zu WPF? In WPF ist es bei einigen Steuerelementen möglich, diese zu **verschachteln**. Dafür wird das einteilige XML-Element in ein zweiteiliges abgeändert und weitere Elemente verschachtelt. Meist ist eine direkte Verschachtelung nicht möglich, hier muss ein Layout-Panel (dazu später mehr) „zwischengeschachtelt“ werden. Als Beispiel könnte z. B. das integrieren eines Bildes in einen Button genannt werden. Ein großes und starkes Leistungsmerkmal von Windows Presentation Foundation ist die **grafische Darstellung**. Im Vergleich zu WinForm werden WPF-Fenster und deren Steuerelemente mit Direct3D gerendert und somit von dem Grafikprozessor (GPU) bearbeitet.
Bei Windows Forms hatte zudem jedes Steuerelement, welches im Designer angelegt wurde, zwingend einen Namen. Über diesen (Variablen-)Namen konnte auf das Steuerelement über den Programmcode zugegriffen werden. Die Variablendeklaration wurde hier in der Form1.Designer.cs-Datei vorgenommen. In WPF geschieht dies in der .g.cs-Datei (dazu gleich mehr), welche durch Visual Studio generiert wird. Hierbei muss **nicht jedes Steuerelement zwingend einen Namen** besitzen. Natürlich ist zu beachten: Besitzt ein Steuerelement keinen Namen, so können wir über den Programmcode auch nicht auf das Steuerelement mit Hilfe des Variablennamens zugreifen.
Schauen wir uns als nächstes einmal die von Visual Studio angelegten Dateien an: App.xaml, App.xaml.cs, MainWindow.xaml und MainWindow.xaml.cs. Dabei stellt die .xaml-Datei immer die XML-Datei mit dem XAML-Code zur Verfügung. Die dazugehörige .xaml.cs-Datei ist dabei die sogenannte **Code-Behind-Datei**, welche den C#-Code der Klasse enthält. Hierbei ersetzt die App.xaml- und App.xaml.cs-Datei die von WinForm bekannte Programm.cs-Datei. In der App.xaml-Datei wird das Haupt-Formular durch das Attribut *StartupUri* des Elements *Application* festgelegt. Über das Attribut *x:Class* legen wir den Namensraum und die Klasse innerhalb einer XAML-Datei fest. *x* ist hierbei eine Referenz für den Namensraum für den dazugehörigen XML-Standard (siehe Attribut *xmlns:x*). In der MainWindow.xaml.cs-Datei können wir sehen, dass das Root-Element *Window* ist. Dies ist die Klasse des standardisierten WPF-Fensters. In C# ist es mit Hilfe von WPF möglich, sogenannte User-Controls zu erstellen. Dabei handelt es sich um **eigene Steuerelemente**. Bei einem solchen eigenen Steuerelement ist das Root-Element dann nicht *Window,* sondern *UserControl* (dazu später mehr). Wenn wir ein neues Fenster oder User-Control erstellen, erhalten wir dabei immer eine .xaml- und eine .xaml.cs-Datei. Innerhalb der MainWindow.xaml.cs-Datei können wir sehen, dass sich untergeordnet vom Root-Element das Element *Grid* befindet. Dabei handelt es sich um ein sogenanntes **Layout-Panel**, welches wir auch durch ein anderes Layout-Panel (z. B. *StackPanel*) ersetzen können. Dem Root-Element werden keine Steuerelemente direkt untergeordnet, sondern immer nur einem Layout-Panel. Der Inhalt der MainWindow.xaml.cs unterscheidet sich nicht viel von der aus Windows Form bekannten Form1.cs. Wenn Sie wissen wollen was hinter der *InitializeComponent()*-Funktion steckt, dann schauen Sie mal in den Ordner obj/Debug. Dort gibt es die Dateien App.g.cs und MainWindow.g.cs. Diese stellen den Zusammenhang zwischen der .xaml- und .xaml.cs-Datei her. Unter anderem befindet sich in der .g.cs-Datei auch die Deklaration von Variablen von Steuerelementen. So wie die von Windows Forms bekannten .Designer.cs-Dateien sollten die .g.cs-Dateien ebenfalls nicht geändert werden, da diese von Visual Studio generiert werden.
Die Klassen für Windows Presentation Foundation Anwendungen befinden sich in **mehreren Namensräumen**. Bei Windows Form Anwendungen wurde lediglich der Namensraum *System.Windows.Forms* benötigt. In WPF wurde eine Aufteilung in mehrere Namensräume vorgenommen: *System.Windows* (enthält grundlegende Klasse), *System.Windows.Controls* (enthält Steuerelemente), *System.Windows.Data* (enthält Klassen zur Daten-Bindung), *System.Windows.Input* (enthält Klassen für das Eingabesystem), *System.Windows.Navigation* (enthält Klassen für die Navigierungs-Funktionen), *System.Windows.Media* (enthält Grafik- und Medien-Klassen) und *System.Windows.Shapes* (enthält grafische Steuerelemente wie Rechtecke, Linien etc.). Daneben bindet Visual Studio standardmäßig noch den Namensraum *System.Windows.Documents* ein, welcher für uns jedoch nicht von größerer Bedeutung ist.
Natürlich sind die meisten Eigenschaften, Funktionen und Ereignisse in WPF **ähnlich oder sogar identisch** mit denen aus WinForm. Doch bevor wir uns auf die verschiedenen Steuerelemente stürzen, wollen wir uns noch ein paar Unterschiede anschauen. Die Basisklasse von so gut wie allen WPF-Steuerelementen ist *Control*. Alle Ereignisse, welche keine besonderen **Event-Argumente** besitzen, werden nicht mehr Event-Argumente der Klasse *EventArgs*, sondern der Klasse *RoutedEventArgs* übergeben. Die Eigenschaften, welche als Datentyp bool besitzen, wurden in WPF umbenannt (z. B. aus *Enabled* wird *IsEnabled*, aus *Visible* wird *IsVisible*). Die Eigenschaft *Text,* welche wir aus WinForm-Anwendungen kennen, ist oftmals unter den Namen *Title* oder *Content* zu finden. Die *Content*-Eigenschaft entspricht immer dem **Inhalt des Steuerelements**. An Stelle zur Verwendung der Eigenschaft kann auch das Element in ein mehrteiliges Element ersetzt

werden, bei welchem der Text innerhalb der Tags notiert wird.
Ein weiterer sehr wichtiger Vorteil von WPF ist die Bindung von Daten und das Erstellen von Darstellungs-Templates (Vorlagen). Hiermit ist es z. B. möglich einer Auswahlliste nicht nur Text, sondern zusätzlich auch ein Bild anzuzeigen. Die Daten-Bindung mit WPF ist sehr komplex und umfangreich, weshalb wir es in ein eigenes Kapitel ausgelagert haben. Dort werden wir genauer auf dieses Thema eingehen.
Für welche grafische Oberfläche sollte ich mich als Programmierer also entscheiden? Sollte ich WPF gar nicht lernen, wenn ich von Windows Forms überzeugt bin? Sollte ich nur noch WPF einsetzen und auf Windows Forms komplett verzichten? Beide Konzepte haben ihre **Stärken** und man sollte stets beide grafischen Oberflächen kennen und anwenden können. Es muss vielmehr für jeden **spezifischen Fall** entschieden werden, welche Oberfläche verwendet werden soll. Ein großer Vorteil von WPF ist die Trennung zwischen Design und Programm. Des Weiteren sollte auch die Datenbindung (dazu mehr im nächsten Kapitel) und die grafische Aufbereitung als Vorteil von WPF angesehen werden. Ein Nachteil von WPF ist, dass es zum Teil noch nicht alle Steuerelemente von Windows Form gibt. Auch ist das Erstellen einer einfachen Applikation in Windows Forms einfacher und schneller umzusetzen als in WPF. WPF besitzt zudem keine eingebauten Dialoge (wie z. B. zum Öffnen oder Speichern einer Datei), diese müssen über den Namensraum *Microsoft.Win32* bezogen werden (Klasse *OpenFileDialog* und *SaveFileDialog*). Die Dialoge müssen komplett durch den „eigenen“ Programmcode erstellt werden. Eine Platzierung des Steuerelements über den Designer ist nicht möglich.
Über die Klasse *Window* können wir einige **Einstellungen für unser Fenster** vornehmen. Die Eigenschaft *ResizeMode* bestimmt die **Größenkonfiguration** für das Fenster. Dafür werden Werte der Enumeration *ResizeMode* verwendet: *NoResize* (das Fenster kann nicht skaliert werden, Maximierungs- und Minimierungs-Button sind ausgeblendet), *CanMinimize* (das Fenster kann nicht skaliert werden, Maximierungs- und Minimierungs-Button sind eingeblendet, Maximierungs-Button ist ausgegraut), *CanResize* (das Fenster kann skaliert werden, Maximierungs- und Minimierungs-Button sind eingeblendet) und *CanResizeWithGrip* (das Fenster kann skaliert werden, Maximierungs- und Minimierungs-Button sind eingeblendet, ein „Skalierungs-Icon“ wird in der Ecke unten rechts angezeigt). Die Eigenschaft *ShowInTaskbar* gibt an, ob das **Programm-Icon in der Taskleiste** angezeigt werden soll. *Title* legt den **Programmtitel** fest, welcher u. a. in der Programmleiste angezeigt wird. Über die *WindowState*-Eigenschaft kann der aktuelle **Status des Fensters** abgefragt werden: *Normal* (Fenster geöffnet), *Maximized* (Fenster maximiert geöffnet) und *Minimized* (Fenster minimiert geöffnet). Wie auch bei der *Form*-Klasse, können wir über die Funktion *Close()* das Fenster schließen, mit *Hide()* das Fenster unsichtbar machen und mit *Show()* oder *ShowDialog()* ein Formular anzeigen.
Die Klasse *Control* ist, wie bereits oben genannt, die **Basisklasse vieler Steuerelemente** bzw. letztendlich aller Steuerelemente. Die Eigenschaft *Background* und *Foreground* legt die **Hintergrund- und Vordergrund**- bzw. Schriftfarbe fest oder ruft diese ab. *BorderBrush* ist eine Eigenschaft für die **Farbe des Rahmens**. *BorderThickness* legt hingegen die **Breite des Rahmens** fest. Bei den Farbangaben wird eine Farbe der *Brush*-Klasse verwendet (die statischen vordefinierten Farben befinden sich in der *Brushes*-Klasse). *Height* und *Width* legt die Höhe und Breite des Steuerelements fest. Eine Positionierung erfolgt je nach gewähltem Layout-Panel unterschiedlich. In den ersten Beispielen werden wir immer das Layout-Panel *Grid* verwenden. Hier erfolgt eine Ausrichtung z. B. über Abstände. *Margin* legt dabei den Außenabstand (vom Rahmen zum übergeordneten Element) fest. Mit der Eigenschaft *Padding* legen wir den Innenabstand (vom Rahmen zum Inhalt des Elementes) fest. Die *HorizontalAlignment*- und *VerticalAlignment*-Eigenschaft legt die Ausrichtung fest. In den Anfängen wollen wir immer eine Ausrichtung von der linken oberen Ecke ausführen, weshalb wir die Enumerations-Werte *Left* und *Top* benötigen. Wollen wir nicht das Steuerelement selbst, sondern den Inhalt des Steuerelementes positionieren, so gibt es die Eigenschaften *HorizontalContentAlignment* und *VerticalContentAlignment*.

MainWindow.xaml

```
<Window x:Class="CSV20.WPF_Grundlagen.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

MainWindow.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace CSV20.WPF_Grundlagen
{
    /// <summary>
    /// Interaktionslogik für MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
```

```
        {
            InitializeComponent();
        }
    }
}
```

App.xaml

```
<Application x:Class="CSV20.WPF_Grundlagen.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

App.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Windows;

namespace CSV20.WPF_Grundlagen
{
    /// <summary>
    /// Interaktionslogik für "App.xaml"
    /// </summary>
    public partial class App : Application
    {
    }
}
```

Das große Computer ABC

C# lernen

WPF: Text und Eingabefelder

Um in WPF einen Text darzustellen, stehen uns die Steuerelemente *Label*, *TextBlock* und *TextBox* zur Verfügung. Das *Label*-Steuerelement kann nicht wie in Windows Forms für mehrzeiligen Text verwendet werden. Wollen wir einen **Text über mehrere Zeilen** anzeigen, so benötigen wir das Steuerelement *TextBlock*. Sowohl *Label* als auch *TextBlock* werden für **statischen Text** verwendet. Der Inhalt kann über die Eigenschaft *Content* oder innerhalb des Start- und End-Tags notiert werden. Der Text bei einem *TextBlock*-Steuerelement wird jedoch nicht automatisch umgebrochen. Wir müssen hierfür den Wert *Wrap* von der Enumeration *TextWrapping* der Eigenschaft *TextWrapping* zuweisen. Der Enumerations-Wert *WrapWithOverflow* löst ebenfalls einen Zeilenumbruch aus. Passt ein Wort nicht in eine ganze Zeile, so wird es nicht unterbrochen und läuft über die Grenzen hinaus (wird jedoch nicht angezeigt).
Die *TextBox* wird verwendet, um dem Benutzer eine Eingabe anzubieten. Hierbei kann der Inhalt über die Eigenschaft *Text* abgerufen werden. Über die Eigenschaft *MaxLength* (maximale Zeichenlänge) und *MaxLines* (maximale Zeilenanzahl) können wir den Inhalt des Eingabefelds begrenzen. Standardmäßig ist die *TextBox* einzeilig, weshalb wir über die Eigenschaft *TextWrapping* und einen passenden Wert der gleichnamigen Enumeration die **Mehrzeiligkeit aktivieren** können. Hierdurch ist es jedoch noch nicht erlaubt, die Enter-Taste zu drücken, um einen **manuellen Zeilenumbruch** auszulösen. Hierfür muss der Wert *AcceptsReturn* auf *true* gesetzt werden. Über das Event *TextChanged* können wir Änderungen am Text durch ein Ereignis feststellen und darauf reagieren. Ein Eingabefeld für Passwörter ist das Steuerelement *PasswordBox*. Hier können wir über die Eigenschaft *PasswordChar* das Zeichen festlegen, welches als **Dummy-Zeichen** für das Passwort angezeigt werden soll.

MainWindow.xaml

```
<TextBlock TextWrapping="Wrap" Height="60" Margin="10,10,10,0"
           VerticalAlignment="Top" HorizontalAlignment="Left">
    Dies ist ein mehrzeiliger Text, weshalb wir den TextBlock als Element verwenden. In
    das untere Textfeld können Sie einen Text eingeben, welcher über ein Ereignis in einem
    Label angezeigt wird. Probieren Sie es aus ...
</TextBlock>
<TextBox Name="textBoxEingabe" Height="25" Width="150" Margin="20,80,0,0"
         VerticalAlignment="Top" HorizontalAlignment="Left"
         MaxLength="50" TextChanged="textBoxEingabe_TextChanged" />
<Label Name="labelEingabe" Height="25" Width="400" Margin="40,120,0,0"
       VerticalAlignment="Top" HorizontalAlignment="Left"
       Content="Sie haben '' eingegeben!" />
```

MainWindow.xaml.cs

```
private void textBoxEingabe_TextChanged(object sender, TextChangedEventArgs e)
{
    labelEingabe.Content = "Sie haben '" + textBoxEingabe.Text + "' eingegeben!";
}
```

Das große Computer ABC

C# lernen

WPF: Formularfelder

Zu den Steuerelementen für Formulare gehören *CheckBox*, *RadioButton*, *GroupBox*, *ComboBox*, *TextBox*, *PasswordBox* und *Button*. Die *TextBox* und *PasswordBox* haben wir ja bereits kennengelernt. Die anderen wollen wir Ihnen nun vorstellen.
Die *CheckBox* stellt ein **Kontrollkästchen** zur Verfügung. Die Eigenschaft *Content* legt den **Beschriftungstext** fest, welcher rechts vom Kontrollkästchen angezeigt wird. Über die Eigenschaft *IsChecked* kann abgefragt oder festgelegt werden, ob die *CheckBox* ausgewählt ist oder nicht. Als Datentyp wird ein **nullable** *bool* (Datentyp-Darstellung als *Nullable* oder als *bool?*) verwendet, wodurch die Zuweisung des Werts *null* erlaubt ist. Alle „einfachen" Datentypen die nullable sind, können nicht direkt umgewandelt werden. Hierfür können wir die Eigenschaft *Value* des Datentyp-Objekts nutzen. Es gilt jedoch zu beachten, dass eine Exception ausgelöst wird, wenn der Wert des Objekts *null* ist. Wollen wir dies abfangen, müssen wir die Eigenschaft *HasValue* abfragen bzw. eine Abfrage, ob der Wert *null* ist, durchführen. Doch warum wird überhaupt eine nullable *bool* verwendet? Der Grund ist, dass die *CheckBox* einen dritten Status unterstützt (nur falls *IsThreeState* auf *true* gesetzt ist). In diesem Fall wird kein Häkchen in der *CheckBox*, sondern eine gefüllte *CheckBox* angezeigt. Ist dies der Fall, so entspricht der Wert von *IsChecked* *null*. Im Gegensatz zu Windows Forms gibt es als Ereignis nicht das Event *CheckedChanged*, sondern die Ereignisse *Checked* und *Unchecked*. Diese Aufteilung der Ereignisse ist für einige Fälle sehr praktisch, für andere hingegen weniger. Zumeist ist es jedoch ein Vorteil, andernfalls kann natürlich für beide Ereignisse die gleiche Event-Funktion registriert werden (siehe Beispiel).
Der *RadioButton* ist ein **Auswahlkästchen**, welches zusammen mit anderen *RadioButton*-Steuerelementen verwendet wird, um eine einzelne Auswahl von mehreren Möglichkeiten zu treffen. Auch bei dem *RadioButton* gibt es die Eigenschaft *IsChecked* und *IsThreeState* und die Ereignisse *Checked* und *Unchecked*. Wollen wir mehrere Gruppen von *RadioButton*-Steuerelementen benutzen, so benötigen wir die Möglichkeit, diese Steuerelemente zu gruppieren. Hierfür können wir, wie auch bei Windows Forms, die *GroupBox* benutzen oder aber die Gruppierung mit Hilfe der Eigenschaft *GroupName* durchführen. Bei letzterem müssen alle *RadioButton*-Steuerelemente, welche gruppiert werden sollen, den gleichen Namen in der Eigenschaft *GroupName* aufweisen.
Die *ComboBox* stellt uns eine Auswahlliste in Form einer **Drop-Down-Liste** dar. Hierfür werden dem *ComboBox*-Element, *ComboBoxItem*-Elemente verschachtelt, welche als Auswahl angezeigt werden sollen. Dem *ComboBoxItem* wird über die Eigenschaft *Content* der Name der Auswahlmöglichkeit zugewiesen. Die Eigenschaft *SelectedIndex* legt den Index der ausgewählten Auswahl fest oder ruft diesen ab. Über die Eigenschaft *Items* können wir auch programmiertechnisch auf die Liste der Auswahlmöglichkeiten zugreifen.
Der *Button* dient für das **gewollte Auslösen eines Ereignisses** durch den Benutzer. Dafür wird das *Click*-Event verwendet. Über die Eigenschaft *Content* können wir den Titel des Buttons festlegen.

MainWindow.xaml

```
<CheckBox Margin="20,20,0,0" VerticalAlignment="Top" HorizontalAlignment="Left"
Name="checkBoxGroupA"
          Content="Gruppe A aktivieren" Checked="checkBoxGroupA_CheckedChanged"
Unchecked="checkBoxGroupA_CheckedChanged" />
<GroupBox VerticalAlignment="Top" HorizontalAlignment="Left" Margin="30,50,0,0" Width="125"
Header="Gruppe A" Name="groupBoxA">
    <Grid>
        <RadioButton VerticalAlignment="Top" HorizontalAlignment="Left" Margin="10,10,0,0"
IsEnabled="False" Name="radioButtonA1" Content="Auswahl A.1" IsChecked="True" />
        <RadioButton VerticalAlignment="Top" HorizontalAlignment="Left" Margin="10,30,0,0"
IsEnabled="False" Name="radioButtonA2" Content="Auswahl A.2" />
        <RadioButton VerticalAlignment="Top" HorizontalAlignment="Left" Margin="10,50,0,0"
IsEnabled="False" Name="radioButtonA3" Content="Auswahl A.3" />
    </Grid>
</GroupBox>
<GroupBox VerticalAlignment="Top" HorizontalAlignment="Left" Margin="180,50,0,0" Width="200"
Header="Gruppe B">
    <Grid>
        <TextBox VerticalAlignment="Top" HorizontalAlignment="Left" Margin="30,30,0,0"
Name="textBoxEingabe" Width="125" TextChanged="textBoxEingabe_TextChanged" />
        <PasswordBox VerticalAlignment="Top" HorizontalAlignment="Left" Margin="30,80,0,0"
Name="passwordBoxEingabe" Width="125" PasswordChar="*" IsEnabled="False" />
        <!-- Anordnung verändert, da das Checked-Event das Event auslösen würde, bevor die
Textfelder bekannt sind -->
        <RadioButton VerticalAlignment="Top" HorizontalAlignment="Left" Margin="10,10,0,0"
Name="radioButtonText" Content="Text-Eingabe" Checked="radioButtonGroupB_Checked"
IsChecked="True" />
        <RadioButton VerticalAlignment="Top" HorizontalAlignment="Left" Margin="10,60,0,0"
Name="radioButtonPassword" Content="Passwort-Eingabe" Checked="radioButtonGroupB_Checked" />
    </Grid>
</GroupBox>
<ComboBox VerticalAlignment="Top" HorizontalAlignment="Left" Margin="20,150,0,0" Width="120"
```

```
SelectedIndex="1" Name="comboBoxAuswahl">
    <ComboBoxItem Content="Auswahl 1" />
    <ComboBoxItem Content="Auswahl 2" />
    <ComboBoxItem Content="Auswahl 3" />
</ComboBox>
<Button VerticalAlignment="Top" HorizontalAlignment="Left" Margin="290,180,0,0"
        FontWeight="Bold" Content="Info anzeigen" Click="Button_Click" />
```

MainWindow.xaml.cs

```
private void checkBoxGroupA_CheckedChanged(object sender, RoutedEventArgs e)
{
    radioButtonA1.IsEnabled = radioButtonA2.IsEnabled = radioButtonA3.IsEnabled =
checkBoxGroupA.IsChecked.Value;
    if (checkBoxGroupA.IsChecked.Value)
        checkBoxGroupA.Content = "Gruppe A deaktivieren";
    else
        checkBoxGroupA.Content = "Gruppe A aktivieren";
}

private void radioButtonGroupB_Checked(object sender, RoutedEventArgs e)
{
    if (radioButtonText.IsChecked.Value)
        textBoxEingabe.IsEnabled = true;
    else
        textBoxEingabe.IsEnabled = false;
    passwordBoxEingabe.IsEnabled = !textBoxEingabe.IsEnabled;
}

private void textBoxEingabe_TextChanged(object sender, TextChangedEventArgs e)
{
    passwordBoxEingabe.Password = textBoxEingabe.Text;
}

private void Button_Click(object sender, RoutedEventArgs e)
{
    string sInfoAusgabe = "Informationen zum Formular:\r\n";

    if (checkBoxGroupA.IsChecked.Value)
    {
        sInfoAusgabe += "- CheckBox für Gruppe A ausgewählt\r\n";
        if (radioButtonA1.IsChecked.Value)
            sInfoAusgabe += "  > Radio-Button 1\r\n";
        else if (radioButtonA2.IsChecked.Value)
            sInfoAusgabe += "  > Radio-Button 2\r\n";
        else if (radioButtonA3.IsChecked.Value)
            sInfoAusgabe += "  > Radio-Button 3\r\n";
    }

    if (radioButtonText.IsChecked.Value)
        sInfoAusgabe += "- Gruppe B: Text-Eingabe\r\n";
    else
        sInfoAusgabe += "- Gruppe B: Passwort-Eingabe\r\n";

    sInfoAusgabe += "- DropDown-Liste: Index " + comboBoxAuswahl.SelectedIndex;

    MessageBox.Show(sInfoAusgabe, "Formular-Info", MessageBoxButton.OK,
MessageBoxImage.Information);
}
```

Das große Computer ABC

C# lernen

WPF: Grafiken

Eine Grafik in WPF können wir mit Hilfe des Steuerelements *Image* anzeigen. Die Eigenschaft erwartet hierbei ein Objekt der *ImageSource*-Klasse. Ist die Grafik im Projekt eingebunden (und wird somit in die resultierende exe-Datei mit „einkompiliert"), so kann lediglich der Dateiname (mit evtl. dem Pfad) angegeben werden (siehe Beispiel). Über die Eigenschaft *Stretch* und die gleichnamige Enumeration können wir angeben wie bzw. ob das **Bild skaliert** werden soll: *None* (das Bild wird nicht skaliert), *Fill* (das Bild wird skaliert, das Seitenverhältnis wird nicht beibehalten), *Uniform* (das Bild wird skaliert, das Seitenverhältnis wird beibehalten, dies ist die Standardeinstellung) oder *UniformToFill* (das Bild wird skaliert, das Seitenverhältnis wird beibehalten, jedoch werden „überstehende" Inhalte abgeschnitten). Die *StretchDirection*-Eigenschaft gibt an wie bzw. in welche **Richtung das Bild skaliert** werden soll. Als Wert für diese Eigenschaft benötigen wir einen Wert der Enumeration *StretchDirecton*: *Both* (Skalierung je nach Stretch-Eigenschaft, dies ist die Standardeinstellung), *DownOnly* (Skalierung nach oben, wenn das Bild kleiner als das übergeordnete Element ist) oder *UpOnly* (Skalierung nach unten, wenn das Bild kleiner als das übergeordnete Element ist).

MainWindow.xaml

```
1 <Image Source="CSharp-Icon.png" />
```

Das große Computer ABC

C# lernen

WPF: Menüs

Um in einem WPF-Fenster ein **Hauptmenü** anzuzeigen, benötigen wir das *Menu*-Element. Das *Menu*-Objekt benötigt standardmäßig immer die komplette Breite des Fensters (was auch so üblich ist). Das Menü erhält **untergeordnete Elemente** in Form von *MenuItem*-Elementen. Natürlich ist es auch möglich, die *MenuItem*-Elemente zu **verschachteln**, um so eine Baumstruktur aufzubauen. Bei Bedarf ist auch eine mehrfache Verschachtelung möglich. Dem *MenuItem* wird über die Eigenschaft *Header* der **anzuzeigende Name** übergeben. Über das Attribut *Click* können wir die Ereignis-Funktion für das Klicken auf eines der Menü-Einträge registrieren. Über die Eigenschaft *Icon* können wir dem *MenuItem*-Steuerelement ein **Icon zuordnen**, welches links neben dem Titel angezeigt wird. Das *Separator*-Steuerelement kann ebenfalls dem *Menu*-Steuerelement untergeordnet werden und dient als Trennstrich.
Auch in WPF gibt es ein **Kontextmenü**, welches angezeigt wird, wenn mit der rechten Maustaste auf ein Steuerelement geklickt wird. Wurde ein solches Menü nicht deklariert, so wird es auch nicht angezeigt. Die Zuweisung des Kontextmenüs geschieht mit Hilfe der Eigenschaft *ContextMenu*. Vom Aufbau des Menüs ist das Kontextmenü gleich bzw. sehr ähnlich zum Hauptmenü. Auch dem Kontextmenü können *MenuItem*- und *Separator*-Elemente untergeordnet werden. Doch nochmals zurück zur Zuweisung: Wie können wir das *ContextMenu*- und die *MenuItem*-Elemente einem Steuerelement wie dem *Button* unterordnen? Als erstes machen wir aus dem Steuerelement ein zweiteiliges XML-Element und ordnen diesem das XML-Element *Button.ContextMenu* unter. Dadurch können wir auf die Eigenschaft per XAML zugreifen. Diesem Element wird nun das *ContextMenu*-Element untergeordnet.

MainWindow.xaml

```
<Menu VerticalAlignment="Top">
    <MenuItem Header="Datei">
        <MenuItem Header="Neu">
            <MenuItem Header="Textdokument" Click="menu_ItemClicked" />
            <MenuItem Header="Tabelle" Click="menu_ItemClicked" />
        </MenuItem>
        <MenuItem Header="Öffnen">
            <MenuItem Header="Textdokument" Click="menu_ItemClicked" />
            <MenuItem Header="Tabelle" Click="menu_ItemClicked" />
        </MenuItem>
        <MenuItem Header="Speichern" Click="menu_ItemClicked" />
        <Separator />
        <MenuItem Header="Beenden" Click="menu_ItemClicked" />
    </MenuItem>
    <MenuItem Header="Hilfe">
        <MenuItem Header="Hilfe öffnen" Click="menu_ItemClicked" />
        <Separator />
        <MenuItem Header="Info" Click="menu_ItemClicked" />
    </MenuItem>
</Menu>
<Button VerticalAlignment="Top" HorizontalAlignment="Left" Padding="5" Margin="40,40,0,0"
Content="Bitte rechts klicken">
    <Button.ContextMenu>
        <ContextMenu>
            <MenuItem Name="menuItemMinimieren" Header="Minimieren"
Click="contextMenu_ItemClicked" />
            <MenuItem Name="menuItemSchließen" Header="Schließen"
Click="contextMenu_ItemClicked" />
        </ContextMenu>
    </Button.ContextMenu>
</Button>
```

MainWindow.xaml.cs

```
private void menu_ItemClicked(object sender, RoutedEventArgs e)
{
    MenuItem oMenuItem = (MenuItem)sender;
    MenuItem oMenuItemParent = (MenuItem)oMenuItem.Parent;

    if (oMenuItem.Header.ToString() == "Beenden")
        this.Close();
    else
    {
        // Prüfen ob übergeordnetes Element vom oMenuItemParent ein MenuItem ist (andernfalls
exisitiert nur eine 2 teilige Hierarchie)
        if (oMenuItemParent.Parent is MenuItem)
            MessageBox.Show("Sie haben auf \"" + ((MenuItem)oMenuItemParent.Parent).Header +
```

```
13  "\" -> \"" + oMenuItemParent.Header + "\" -> \"" + oMenuItem.Header + "\" geklickt!");
14          else
                MessageBox.Show("Sie haben auf \"" + oMenuItemParent.Header + "\" -> \"" +
15  oMenuItem.Header + "\" geklickt!");
16      }
17  }
18
19  private void contextMenu_ItemClicked(object sender, RoutedEventArgs e)
20  {
21      if (sender == menuItemMinimieren)
22          this.WindowState = System.Windows.WindowState.Minimized;
23      else
24          this.Close();
    }
```

Das große Computer ABC

C# lernen

WPF: Tabs

In WPF ist es möglich, **Registerkarten** (auch Tabs genannt) zu erstellen. Diese Möglichkeit kennen wir bereits von WinForm. Der Aufbau der Steuerelemente sowie der Eigenschaften ist ähnlich und teilweise identisch mit denen von Windows Forms. Deshalb haben wir unser Beispiel ähnlich wie im WinForm-Beispiel aufgebaut. So können Sie einen direkten Vergleich vornehmen.
Als Haupt-Element für die Tab-Funktionalität benötigen wir das Steuerelement *TabControl*. Diesem Steuerelement werden *TabItem*-Steuerelemente in der XML-Struktur untergeordnet. Dabei stellen die *TabItem*-Elemente die einzelnen Registerkarten dar, welchen nun ein Layout-Panel untergeordnet wird. Diesem werden dann üblicherweise weitere Steuerelemente untergeordnet. Im *TabItem*-Objekt selbst setzen wir noch die Eigenschaft *Header*, um den **Titel des Tabs** festzulegen. Wollen wir den **Index der angezeigten Registerkarte** abfragen oder festlegen, so kann uns die Eigenschaft *SelectedIndex* des *TabControl*-Steuerelements helfen.

MainWindow.xaml

```
<TabControl Name="tabControlFenster">
    <TabItem Header="Tab 1">
        <Grid Background="White">
            <Button VerticalAlignment="Center" HorizontalAlignment="Center" Padding="10"
FontWeight="Bold" Content="Zu Tab 2 wechseln" Click="tab1Button_Click" />
        </Grid>
    </TabItem>
    <TabItem Header="Tab 2">
        <Grid Background="White">
            <Button VerticalAlignment="Center" HorizontalAlignment="Center" Padding="10"
FontWeight="Bold" Content="Ändere Hintergrund" Click="tab2Button_Click" />
        </Grid>
    </TabItem>
    <TabItem Header="Tab 3">
        <Grid Background="White">
            <Button VerticalAlignment="Center" HorizontalAlignment="Center" Padding="10"
FontWeight="Bold" Content="Programm beenden" Click="tab3Button_Click" />
        </Grid>
    </TabItem>
</TabControl>
```

MainWindow.xaml.cs

```
private void tab1Button_Click(object sender, RoutedEventArgs e)
{
    tabControlFenster.SelectedIndex = 1;
}

private void tab2Button_Click(object sender, RoutedEventArgs e)
{
    Grid oGrid = (Grid)((Button)sender).Parent;

    if (oGrid.Background == Brushes.Blue)
        oGrid.Background = Brushes.White;
    else
        oGrid.Background = Brushes.Blue;
}

private void tab3Button_Click(object sender, RoutedEventArgs e)
{
    this.Close();
}
```

Das große Computer ABC

C# lernen

WPF: Formen

Anders als in Windows Forms ist es in WPF einfach, Rechtecke, Ellipsen (oder auch Kreise) und Linien zu zeichnen. Hierfür müssen wir kein *Paint*-Ereignis registrieren, sondern können es **direkt im Designer** anlegen. Auch das dynamische Erstellen eines solchen grafischen Objekts erfolgt gleich wie das dynamische Erstellen eines Buttons o. Ä.. Doch wie funktioniert das dynamische Hinzufügen eines Steuerelements grundsätzlich? Jedem Steuerelement, welchem Steuerelemente untergeordnet werden können (zumeist einem Layout-Panel), können wir über die Funktion *Add()* der Eigenschaft *Children* ein **Steuerelement hinzufügen**. Im Programmcode erzeugen wir hierfür „programmiertechnisch“ ein Objekt des Steuerelements, welches wir neu erstellen wollen und übergeben es der *Add()*-Funktion der *Children*-Eigenschaft des übergeordneten Steuerelements.
Nun aber zurück zu den Steuerelementen selbst: Die Formen-Steuerelemente befinden sich alle im Namensraum *System.Windows.Shapes*. Die *Rectangle*-Klasse stellt ein **Rechteck** dar, wohingegen die *Ellipse*-Klasse eine **Ellipse** oder auch einen **Kreis** darstellt. Die Klasse *Line* bietet uns die Möglichkeit, eine Linie von einem zum anderen Punkt zu zeichnen. Alle **Shapes** (englisch für Formen) sind von der Klasse *Shape* abgeleitet. Dort gibt es die wichtigen Eigenschaften *Fill*, *Stroke* und *StrokeThickness*. *Fill* und *Stroke* erwarten ein Objekt der *Brush*-Klasse. *Stroke* und *StrokeThicknes* sind für den Rahmen zuständig, wohingegen die *Fill*-Eigenschaft die **Füllfarbe** angibt. Die *Stroke*-Eigenschaft legt die **Farbe des Rahmens** fest. *StrokeThickness* wird als Gleitkommazahl (Datentyp *dobule*) angegeben und gibt die **Breite des Rahmens** an. Natürlich sind auch die Eigenschaften *Margin*, *Height* und *Width* von großer Bedeutung. Diese gehören jedoch der Basisklasse *FrameworkElement* an. Übrigens: Sowohl *Control* als auch *Shape* leiten sich von der *FrameworkElement*-Klasse ab. Bei der Klasse *Line* erfolgt die Positionierung und Größensteuerung nicht mit Hilfe von *Margin*, *Height* und *Width*, sondern über die **Koordinaten-Eigenschaften** *X1*, *X2*, *Y1* und *Y2*. Hierbei legt die X1- und Y1-Eigenschaft die erste Koordinate fest und X2 und Y2 die zweite.

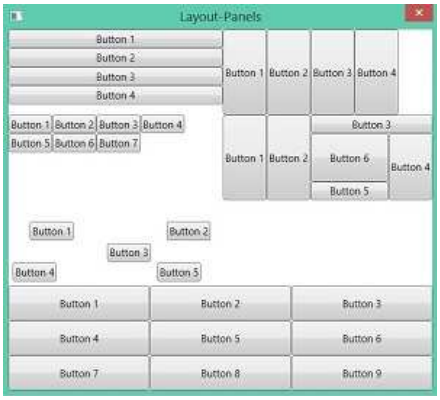
MainWindow.xaml

```
1  <Rectangle Stroke="Blue" StrokeThickness="5" Width="350" Height="180" Fill="AliceBlue" />
2  <Ellipse Fill="Violet" Width="150" Height="150" />
3  <Line Stroke="Red" StrokeThickness="1" X1="57" Y1="48" X2="269" Y2="170" />
```

Das große Computer ABC

C# lernen

WPF: Layout-Panels



Bisher haben wir immer das Layout-Panel *Grid* verwendet. Nun wollen wir auch die anderen Layout-Panels kennenlernen. Ein Layout-Panel (kurz Panel) dient zur **Anordnung von Steuerelementen**. Einem Layout-Panel können direkt ein oder mehrere **Steuerelement(e) untergeordnet** werden. Eine **Liste mit Steuerelementen** können wir über die Eigenschaft *Children* abrufen. Hier können wir auch über die Funktion *Add()* ein neues Steuerelement hinzufügen und über *Remove()* entfernen. Alle Layout-Panels gehören der Basisklasse *Panel* an. Jedes Layout-Panel von WPF hat verschiedene „Eigenschaften“ und erzeugt somit eine andere Ausgabe. Natürlich können durch verschiedene Attribute wie z. B. *Margin*, *Height* und *Width* mit unterschiedlichen Layout-Panels gleiche Anzeigen erzeugt werden.
Beim *Grid*-Panel können wir mit Hilfe des untergeordneten *Grid.RowDefinitions*- und *Grid.ColumnDefinitions*-Elements **Definitionen für Zeilen und Spalten** erstellen. Somit ist die Hauptaufgabe des *Grid*-Panels also die **tabellarische Darstellung**. Den Elementen werden Elemente des Typs *RowDefinition* und *ColumnDefinition* untergeordnet. Hier können wir über die Eigenschaften *Height* und *Width* die Größe einer Zeile oder Spalte steuern. Wird die Angabe weggelassen, so wird der Wert automatisch durch die restliche Höhe bzw. Breite ermittelt. Steuerelemente die sich innerhalb des *Grid* Layout-Panels befinden, können über die Eigenschaften *Grid.Row* und *Grid.Column* positioniert werden. Dabei handelt es sich um einen nullbasierenden Index. Soll sich ein Steuerelement über mehrere Zeilen oder Spalten erstrecken, so kann uns die Eigenschaft *Grid.RowSpan* und *Grid.ColumnSpan* helfen.
Das *UniformGrid* wird ebenfalls zur **tabellarischen Darstellung** verwendet. Jedoch besitzt jede Zeile und jede Spalte eine **einheitliche Größe**. Die Anzahl an Zeilen und Spalten wird über die Eigenschaft *Rows* und *Columns* angegeben. Die untergeordneten Steuerelemente werden über die Reihenfolge, in welcher diese notiert werden, positioniert.
StackPanel ist ein Layout-Panel, bei welchem die Anordnung **horizontal untereinander** oder **vertikal nebeneinander** erfolgt. Diese Richtung kann mit Hilfe der Eigenschaft *Orientation* festgelegt werden: *Horizontal* (Ausrichtung untereinander von oben nach unten), *Vertical* (Ausrichtung nebeneinander von links nach rechts).
Beim *WrapPanel* wird eine Anordnung ähnlich zum *StackPanel* vorgenommen. Auch hier können wir die Anordnungs-Richtung mit der Eigenschaft *Orientation* festlegen. Beim *WrapPanel* werden die Elemente jedoch nicht nur untereinander oder nebeneinander platziert. Vielmehr erfolgt sowohl eine Anordnung **untereinander und nebeneinander**. Erläutern wir dies etwas genauer für den *Horizontal*-Wert der Eigenschaft *Orientation* (Enumeration *Orientation*): Zuerst werden alle Steuerelemente nebeneinander platziert. Ist der gegebene Platz voll, so wird ein Zeilenumbruch (englisch wordwrap) erzeugt. Dieses Vorgehen kann sich ein bis mehrere Mal(e) wiederholen.
Das Layout-Panel *DockPanel* erlaubt eine Anordnung an eine der **vier Seiten eines Rechtecks** (links, oben, rechts oder unten). Hierfür werden die Steuerelemente, welche untergeordnet sind, über die Eigenschaft *DockPanel.Dock* angeordnet. Mögliche Werte sind *Left*, *Top*, *Right* und *Bottom*. Natürlich können die Anordnungen an eine bestimmte Seite auch **gestapelt** werden (siehe Beispiel). Die Eigenschaft *LastChildFill* bestimmt, ob das letzte untergeordnete Element den **restlichen Platz des Layout-Panels auffüllen** soll. Der Standardwert der Eigenschaft ist *true*. Steuerelemente bei denen das *DockPanel.Dock*-Attribut nicht gesetzt ist, werden automatisch links angeordnet.
Das letzte Layout-Panel, welches wir uns anschauen möchten, ist *Canvas*. Das *Canvas*-Panel eignet sich sehr gut für die **absolute Anordnung** von Steuerelementen. Hierbei können die untergeordneten Steuerelemente über *Canvas.Left*, *Canvas.Top*, *Canvas.Right* und *Canvas.Bottom* angeordnet werden. Natürlich können wir auch Steuerelemente gegeneinander überlagern. Hierbei kann es notwendig sein, die **Reihenfolge auf der Z-Achse** zu ändern. Die Eigenschaft *Canvas.ZIndex* legt den Index für die Anordnung auf der Z-Achse fest. Dabei gilt, umso größer der Wert, umso weiter „vorne“ befindet sich das Steuerelement.

MainWindow.xaml

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="100" />
        <RowDefinition Height="100" />
        <RowDefinition Height="100" />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="250" />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <StackPanel Grid.Row="0" Grid.Column="0" Orientation="Vertical">
        <Button Content="Button 1" />
        <Button Content="Button 2" />
        <Button Content="Button 3" />
```

```
        <Button Content="Button 4" />
    </StackPanel>
    <StackPanel Grid.Row="0" Grid.Column="1" Orientation="Horizontal">
        <Button Content="Button 1" />
        <Button Content="Button 2" />
        <Button Content="Button 3" />
        <Button Content="Button 4" />
    </StackPanel>
    <WrapPanel Grid.Row="1" Grid.Column="0">
        <Button Content="Button 1" />
        <Button Content="Button 2" />
        <Button Content="Button 3" />
        <Button Content="Button 4" />
        <Button Content="Button 5" />
        <Button Content="Button 6" />
        <Button Content="Button 7" />
    </WrapPanel>
    <DockPanel Grid.Row="1" Grid.Column="1">
        <Button Content="Button 1" DockPanel.Dock="Left" />
        <Button Content="Button 2" DockPanel.Dock="Left" />
        <Button Content="Button 3" DockPanel.Dock="Top" />
        <Button Content="Button 4" DockPanel.Dock="Right" />
        <Button Content="Button 5" DockPanel.Dock="Bottom" />
        <Button Content="Button 6" />
    </DockPanel>
    <Canvas Grid.Row="2" Grid.Column="0">
        <Button Canvas.Left="25" Canvas.Top="25" Content="Button 1" />
        <Button Canvas.Left="185" Canvas.Top="25" Content="Button 2" />
        <Button Canvas.Left="115" Canvas.Top="50" Content="Button 3" />
        <Button Canvas.Left="5" Canvas.Bottom="5" Content="Button 4" />
        <Button Canvas.Right="25" Canvas.Bottom="5" Content="Button 5" />
    </Canvas>
    <UniformGrid Grid.Row="3" Grid.Column="0" Grid.ColumnSpan="2" Rows="3" Columns="3">
        <Button Content="Button 1" />
        <Button Content="Button 2" />
        <Button Content="Button 3" />
        <Button Content="Button 4" />
        <Button Content="Button 5" />
        <Button Content="Button 6" />
        <Button Content="Button 7" />
        <Button Content="Button 8" />
        <Button Content="Button 9" />
    </UniformGrid>
</Grid>
```

Das große Computer ABC

C# lernen

WPF: Benutzersteuerelemente

Um in WPF **eigene Steuerelemente**, sogenannte Benutzersteuerelemente, zu definieren, hilft uns Visual Studio beim Erstellen eines Grundgerüsts (Projektkontextmenü > Hinzufügen > Neues Element > Benutzersteuerelement). Das Steuerelement ist zu diesem Zeitpunkt noch leer. Wurzelelement des Benutzersteuerelements ist *UserControl*. Nun können wir dort alle mögliche andere Steuerelemente verschachteln (stets untergeordnet in einem Layout-Panel). Auf Grund des Konzepts der Objektorientierung können wir von einer fremden Klasse (z. B. dem Fenster, in welchem das Steuerelement angezeigt wird) nicht auf die Steuerelemente des Benutzersteuerelements zugreifen. Hierfür müssen wir (falls erforderlich) in der Code-Behind-Datei **Eigenschaften und Funktionen implementieren**. Wer versucht, das Benutzersteuerelement im Designer im XAML-Code hinzuzufügen, wird vorerst daran scheitern. Dies kommt daher, dass der Namensraum des eigenen Programms im XAML-Code noch nicht bekannt ist. Hierfür definieren wir einen eigenen **XML-Namensraum** mit Hilfe von *xmlns:name* (*name* ist frei wählbar). Als Wert des Attributs muss *clr-namespace:Programm_Namensraum* angegeben werden. Das Benutzersteuerelement kann nun über den Elementname *name:Mein_Benutzersteuerelement* eingebunden werden.

TextBoxClear.xaml

```
<UserControl x:Class="CSV20.WPF_Steuerelemente.TextBoxClear"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             mc:Ignorable="d" d:DesignHeight="80" d:DesignWidth="300">
    <Grid>
        <Label HorizontalAlignment="Left" VerticalAlignment="Top" Margin="10,10,0,0"
Name="labelBeschriftung" />
        <TextBox HorizontalAlignment="Left" VerticalAlignment="Top" Margin="150,15,0,0"
Width="125" Name="textBoxInhalt" />
        <Button HorizontalAlignment="Left" VerticalAlignment="Top" Margin="150,45,0,0"
Padding="3" Name="buttonLöschen" Content="Inhalt zurücksetzen" Click="buttonLöschen_Click" />
    </Grid>
</UserControl>
```

TextBoxClear.xaml.cs

```
using System.Windows;
using System.Windows.Controls;

namespace CSV20.WPF_Steuerelemente
{
    /// <summary>
    /// Interaktionslogik für TextBoxClear.xaml
    /// </summary>
    public partial class TextBoxClear : UserControl
    {
        public string Beschriftung
        {
            get
            {
                return labelBeschriftung.Content.ToString();
            }
            set
            {
                labelBeschriftung.Content = value;
            }
        }

        public string Text
        {
            get
            {
                return textBoxInhalt.Text;
            }
            set
            {
                textBoxInhalt.Text = value;
            }
        }
```

```
34
35          public TextBoxClear()
36          {
37              InitializeComponent();
38          }
39
40          private void buttonLöschen_Click(object sender, RoutedEventArgs e)
41          {
42              textBoxInhalt.Text = "";
43          }
44      }
45  }
```

MainWindow.xaml

```
1  <Window x:Class="CSV20.WPF_Steuerelemente.MainWindow"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          xmlns:o="clr-namespace:CSV20.WPF_Steuerelemente"
5          Title="Benutzersteuerelemente" Height="200" Width="300" ResizeMode="NoResize">
6      <Grid>
7          <o:TextBoxClear HorizontalAlignment="Left" VerticalAlignment="Top" Margin="5,5,0,0"
   Beschriftung="Eingabe 1:" Text="Hallo Welt!" />
8          <o:TextBoxClear HorizontalAlignment="Left" VerticalAlignment="Top" Margin="5,85,0,0"
   Beschriftung="Eingabe 2:" />
9      </Grid>
10 </Window>
```

Das große Computer ABC

C# lernen

WPF: Fenster mit Navigierung

WPF bietet die Funktionalität der **Navigierung**. Darunter versteht man die Verwendung eines Haupt-Fensters, welches die **Anzeige verschiedener Seiten** ermöglicht. Über die Pfeilicons oben links kann eine Navigierung zwischen den vorherigen und folgenden Seiten durchgeführt werden. Das Root-Element des Navigierungs-Fensters ist *NavigationWindow*. Dem Root-Element werden keine weiteren Steuerelemente untergeordnet. Die Seiten (Klasse *Page*) werden separat als eigene Klassen erstellt und mit Visual Studio designt. Dem Hauptfenster können wir mit Hilfe der Eigenschaft *Source* den Pfad der XAML-Datei, der anzuzeigenden Seite zuweisen. Alle Klassen für die Navigierung befinden sich im Namensraum *System.Windows.Navigation*. Ein Beispiel für ein solches Navigierungs-Fenster ist die Windows Systemsteuerung (ab Windows Vista).
Für die Navigierung zur Laufzeit können uns einige Funktionen der Klasse *NavigationService* helfen. *GoBack()* navigiert zur **vorherigen Seite** (falls möglich) und *GoForward()* navigiert zur **folgenden Seite** (falls möglich). Beide Funktionen sollten nur aufgerufen werden, wenn die Eigenschaft *CanGoBack* und *CanGoForward* den Wert *true* zurückgibt. Im Beispiel haben wir darauf verzichtet, da das Zurückgehen zur letzten Seite immer möglich ist. Die Funktion *Navigate()* ermöglicht das **Navigieren zu einer bestimmten Seite**. Hierfür können wir der Funktion ein Objekt der zu öffnenden Seite übergeben. Alternativ können wir auch einen Pfad (Klasse *Uri*) der zu öffnenden Seite (Pfad zur XAML-Datei) als Parameter übergeben.

Page1.xaml

```
<Page x:Class="CSV20.WPF_Navigierung.Page1"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="300" Title="Eingabe-Seite"
Background="White">
    <Grid>
        <Label HorizontalAlignment="Left" VerticalAlignment="Top" Margin="20,15,0,0"
Content="Ihre Eingabe:" />
        <TextBox HorizontalAlignment="Left" VerticalAlignment="Top" Margin="120,20,0,0"
Width="100" Name="textBoxEingabe" />
        <Button HorizontalAlignment="Left" VerticalAlignment="Top" Margin="120,45,0,0"
Padding="3" Content="Zur Seite 2" Click="Button_Click" />
    </Grid>
</Page>
```

Page1.xaml.cs

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Navigation;

namespace CSV20.WPF_Navigierung
{
    /// <summary>
    /// Interaktionslogik für Page1.xaml
    /// </summary>
    public partial class Page1 : Page
    {
        public Page1()
        {
            InitializeComponent();
        }

        private void Button_Click(object sender, RoutedEventArgs e)
        {
            NavigationService.Navigate(new Page2(textBoxEingabe.Text));
        }
    }
}
```

Page2.xaml

```xaml
<Page x:Class="CSV20.WPF_Navigierung.Page2"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="300" Title="Anzeige-Seite" Background="LightGray">
    <Grid>
        <Label HorizontalAlignment="Left" VerticalAlignment="Top" Margin="20,15,0,0" Content="Ihre Eingabe:" />
        <Label HorizontalAlignment="Left" VerticalAlignment="Top" Margin="120,17,0,0" Width="100" Name="labelEingabe" />
        <Button HorizontalAlignment="Left" VerticalAlignment="Top" Margin="120,45,0,0" Padding="3" Content="zurück zur Seite 1" Click="Button_Click" />
    </Grid>
</Page>
```

Page2.xaml.cs

```csharp
using System.Windows.Controls;

namespace CSV20.WPF_Navigierung
{
    /// <summary>
    /// Interaktionslogik für Page2.xaml
    /// </summary>
    public partial class Page2 : Page
    {
        public Page2(string sEingabe)
        {
            InitializeComponent();

            labelEingabe.Content = sEingabe;
        }

        private void Button_Click(object sender, System.Windows.RoutedEventArgs e)
        {
            NavigationService.GoBack();
        }
    }
}
```

MainWindow.xaml

```xaml
<NavigationWindow x:Class="CSV20.WPF_Navigierung.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Fenster mit Navigierung" Height="250" Width="400" ResizeMode="NoResize"
        Source="Page1.xaml">
</NavigationWindow>
```

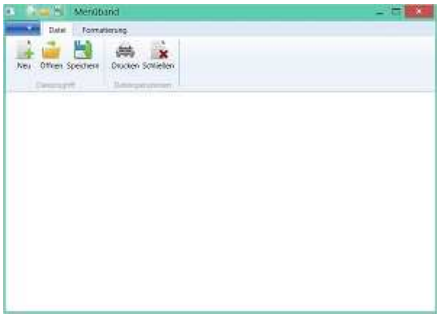
MainWindow.xaml.cs

```csharp
using System.Windows.Navigation;

namespace CSV20.WPF_Navigierung
{
    /// <summary>
    /// Interaktionslogik für MainWindow.xaml
    /// </summary>
    public partial class MainWindow : NavigationWindow
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

Das große Computer ABC

C# lernen

WPF: Menüband



Das Menüband ist eine sehr schöne, aber auch etwas komplexere Art und Weise, ein **„besseres“ Menü** aufzubauen. Anwendungen, die ein solches Menüband nutzen, sind z. B. die Microsoft Office Programme (ab Microsoft Office 2007), Windows Paint (ab Windows 7) und der Windows Explorer (ab Windows 8). Dieses Menü wird auch als **Ribbon** (englisch für Band) oder Ribbon Menu bezeichnet. Dieses Feature ist erst ab dem .NET Framework 4.5 vollständig verfügbar. Grundsätzlich kann das Menü in verschiedenen Arten und Designs aufgebaut werden. Wir wollen hier das Office 2010 ähnliche **Ribbon** vorstellen, welches im .NET Framework implementiert ist. Ein vergleichbares Ribbon können wir z. B. über die Library (DLL) „Fluent Ribbon“ einbinden (Download und Infos unter: https://github.com/fluentribbon/Fluent.Ribbon).
Bevor wir mit der Entwicklung eines Programms mit Menüband beginnen, müssen wir einen **Verweis** auf *System.Windows.Controls.Ribbon* (dabei handelt es sich um den Namensraum, welche die Steuerelemente für das Menüband enthalten) erstellen. Dazu klicken wir mit der rechten Maustaste auf den Eintrag „Verweise“ des Projekts. Im Kontextmenü wählen wir „Verweis hinzufügen“. Ein neues Fenster erscheint, in welchem wir in der Navigation auf der linken Seite den Eintrag „Assemblys“ > „Framework“ wählen. Nun muss auf der rechten Seite nur noch der passende Namensraum bzw. die passende Library (DLL) gesucht und gewählt werden. Mit einem Klick auf „OK“ wird der Verweis hinzugefügt und somit im Projekt eingebunden.
Normalerweise ist es bei Programmen mit Ribbon Menu der Fall, dass die „normale“ Windows-Titelleiste durch eine zum Menüband passende **Titelleiste** ersetzt wird. Um dies zu erreichen, müssen wir das Root-Element *Window* durch *RibbonWindow* ersetzen. Innerhalb unseres Layout-Panels notieren wir nun das Steuerelement *Ribbon*. Dieses stellt uns nun **verschiedene Komponenten** des Menübands zur Verfügung: Anwendungsmenü (Button oben links unterhalb der Titelleiste), Titelleiste mit Schnellzugriff (Icons oben links) und Registerkarten mit Menü-Einträgen.
Um auf das **Anwendungsmenü** zuzugreifen, benötigen wir die Eigenschaft *ApplicationMenu*, welche ein Objekt der *RibbonApplicationMenu*-Klasse erwartet. Dem Menü können nun *RibbonApplicationMenuItem*-Elemente und *RibbonSeparator*-Elemente untergeordnet werden. Dem *RibbonApplicationMenuItem*-Element können wir über die Eigenschaft *Header,* den **anzuzeigenden Text** zuweisen. Die Eigenschaft *ImageSource* legt das [b][i]anzuzeigende Bild[/b][/i] (Icon) fest, welches links neben dem Text angezeigt wird. Des Weiteren ist es auch möglich, *RibbonApplicationMenuItem*-Elemente zu verschachteln.
Die **Schnellzugriffleiste** können wir mit Hilfe der Eigenschaft *QuickAccessToolBar* erstellen. Dort werden *RibbonButton*-Elemente untergeordnet. Diese werden wir später auch noch bei den Registerkarten als Button verwenden. Die Eigenschaft *ToolTip* legt den **anzuzeigenden Info-Text** fest, welcher angezeigt werden soll, sobald man mit der Maus darüber fährt. Die Eigenschaft *SmallImageSource* legt das Icon fest. Auch über die Eigenschaft *LargeImageSource* kann das Icon festgelegt werden, jedoch wird hierbei ein „größeres Bild“ festgelegt. Für die Schnellzugriffleiste wird lediglich die *SmallImageSource*-Eigenschaft benötigt, da diese hier von Bedeutung ist. *LargeImageSource* wird dafür jedoch bei den „großen“ Buttons in den Registerkarten benötigt.
Dem *Ribbon*-Element können wir über die Eigenschaft *Title* den Titel festlegen, welcher innerhalb der Titelleiste (rechts neben der Schnellzugriffsleiste) angezeigt wird. Um die Registerkarten für das Menü zu erstellen, müssen wir dem *Ribbon*-Element *RibbonTab*-Elemente unterordnen.
Ein *RibbonTab*-Element stellt eine einzelne **Registerkarte** dar, welcher nun einzelne Gruppen (*RibbonGroup*) untergeordnet werden. Eine **Gruppe** dient zur Gruppierung von Buttons (*RibbonButton*, *RibbonSplitButton* und *RibbonToggleButton*), Auswahllisten (*RibbonComboBox*), Kontrollkästchen (*RibbonCheckBox*), Auswahlkästchen (*RibbonRadioButton*) und Eingabefeldern (*RibbonTextBox*). Der *RibbonGroup* wird über die Eigenschaft *Header,* die **Beschriftung** zugewiesen, welche am unteren Rand der Gruppe angezeigt wird.
Dem *RibbonButton* können wir über die Eigenschaft *Label* eine Beschriftung hinzufügen, welche unterhalb des Icons angezeigt wird und dabei einen **einfachen Button** darstellt. Als Eigenschaft für die Zuweisung des Icon verwenden wir innerhalb eines *RibbonGroup*-Steuerelements *LargeImageSource* oder *SmallImageSource* je nachdem, ob wir einen großen oder kleinen Button wollen.
Der *RibbonSplitButton* stellt einen **Button mit Unterpunkten** dar. Über die XML-Struktur werden Elemente der *RibbonMenuItem*-Klasse untergeordnet. Bei den *RibbonMenuItem*-Elementen benötigen wir die Eigenschaft *Header,* um die Beschriftung festzulegen. Auch hier ist das Festlegen eines Icons möglich (Eigenschaft *ImageSource*).
Der *RibbonToggleButton* ist ein **Button, welcher zwei Zustände kennt** (gewählt und nicht gewählt). Er ist also mit der *RibbonCheckBox* zu vergleichen. Der Zustand kann über die Eigenschaft *IsChecked* abgefragt oder gesetzt werden.
Die *RibbonComboBox* dient zur Darstellung einer **Drop-Down-Liste**. Über die Eigenschaft *Label* legen wir die Beschriftung fest, welche links neben der Liste angezeigt wird. Dem Element wird das *RibbonGallery*-Element untergeordnet. Diesem Element ordnen wir das Element *RibbonGalleryCategory* unter, welchem wir wiederrum ein oder mehrere *RibbonGalleryItem*-Elemente unterordnen. Die Eigenschaft *Content* legt den **anzuzeigenden Text** für das einzelne Item fest. Die Eigenschaft *SelectedValue* des *RibbonGallery*-Elements ruft den Wert des aktuell ausgewählten Items ab oder legt diesen fest. Zusätzlich sollten wir noch die Eigenschaft *SelectedValuePath* und *MaxColumnCount* festlegen (siehe Beispiel). Durch das Setzen des Werts *IsEditable* auf *true*, erlauben wir dem Benutzer eine Texteingabe. Jedoch sind nur unsere festgelegten Werte erlaubt. Doch warum können wir der *RibbonComboBox* nicht einfach nur *ComboBoxItem*-Elemente unterordnen? Dies geht schon,

jedoch haben wir hierbei weniger „Komfort“. So können wir z. B. bei den einzelnen Items mit Hilfe der *RibbonGallery* unterschiedliche Schriftarten verwenden, um die Auswahl verschiedener Schriftarten zu verdeutlichen.
Zur *RibbonCheckBox* und dem *RibbonRadioButton* gibt es nicht viel Weiteres zu sagen. Die Beschriftung erfolgt über die Eigenschaft *Label*. Auch die Zuweisung eines Icons ist hier möglich. Um eine Gruppe von *RibbonRadioButton*-Steuerelementen zusammenzufassen und somit diese zu gruppieren, benötigen wir die *RibbonGroup* (siehe Beispiel).
Viele weitere Eigenschaften und Ereignisse von den Ribbon-Steuerelementen (wie RibbonButton, RibboCheckBox etc.) sind identisch mit den dazugehörigen „normalen“ Steuerelementen. Dies kommt daher, dass z. B. der *RibbonButton* von der Klasse *Button* abgeleitet ist. So gibt es also auch bei dem *RibbonButton* das *Click*-Event. Übrigens: Auch das *RibbonApplicationMenuItem*-Steuerelement verfügt über ein *Click*-Ereignis.
Neben den teilweisen **großen Schaltflächen** und dem **übersichtlichen Aufbau** des Menüs, verfügt das Ribbon Menu über einen weiteren Vorteil. Beim Drücken der Alt-Taste können wir über verschiedene Buchstaben oder Zahlen durch die Menüs navigieren. Dafür gibt es sowohl bei den Steuerelementen *RibbonApplicationMenu* und *RibbonApplicationMenuItem* als auch *RibbonTab* und *RibbonButton* (und natürlich auch anderen Ribbon-Steuerelementen) die Eigenschaft *KeyTip*. Als Wert des Attributs legen wir einen Buchstaben oder auch eine Zahl fest (siehe Beispiel).

MainWindow.xaml

```
<RibbonWindow x:Class="CSV20.WPF_Menüband.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Menüband" Height="500" Width="700">
    <DockPanel LastChildFill="True">
        <Ribbon DockPanel.Dock="Top" Title="Menüband">
            <Ribbon.ApplicationMenu>
                <RibbonApplicationMenu KeyTip="M">
                    <RibbonApplicationMenuItem ImageSource="Images/Neu.png" Header="Neu"
KeyTip="N" />
                    <RibbonApplicationMenuItem ImageSource="Images/Öffnen.png"
Header="Öffnen" KeyTip="O" />
                    <RibbonApplicationMenuItem ImageSource="Images/Speichern.png"
Header="Speichern" KeyTip="S">
                        <RibbonApplicationMenuItem Header="Speichern" KeyTip="S" />
                        <RibbonApplicationMenuItem Header="Speichern unter" KeyTip="U" />
                        <RibbonSeparator />
                        <RibbonApplicationMenuItem Header="Kopie speichern" KeyTip="K" />
                    </RibbonApplicationMenuItem>
                    <RibbonSeparator />
                    <RibbonApplicationMenuItem ImageSource="Images/Drucken.png"
Header="Drucken" KeyTip="P" />
                    <RibbonApplicationMenuItem ImageSource="Images/Schließen.png"
Header="Schließen" KeyTip="C" />
                    <RibbonSeparator />
                    <RibbonApplicationMenuItem Header="Vorlagen" KeyTip="V">
                        <RibbonApplicationMenuItem Header="leere Datei" KeyTip="L" />
                        <RibbonApplicationMenuItem Header="Einladung" KeyTip="E" />
                        <RibbonApplicationMenuItem Header="Brief" KeyTip="B" />
                    </RibbonApplicationMenuItem>
                    <RibbonSeparator />
                    <RibbonApplicationMenuItem Header="Beenden" KeyTip="E"
Click="MenuExit_Click" />
                </RibbonApplicationMenu>
            </Ribbon.ApplicationMenu>
            <Ribbon.QuickAccessToolBar>
                <RibbonQuickAccessToolBar>
                    <RibbonButton ToolTip="Neu" SmallImageSource="Images/Neu.png" KeyTip="N"
/>
                    <RibbonButton ToolTip="Öffnen" SmallImageSource="Images/Öffnen.png"
KeyTip="O" />
                    <RibbonButton ToolTip="Speichern"
SmallImageSource="Images/Speichern.png" KeyTip="S" />
                </RibbonQuickAccessToolBar>
            </Ribbon.QuickAccessToolBar>
            <RibbonTab Header="Datei" KeyTip="D">
                <RibbonGroup Header="Dateizugriff">
                    <RibbonButton LargeImageSource="Images/Neu.png" Label="Neu" KeyTip="N" />
                    <RibbonButton LargeImageSource="Images/Öffnen.png" Label="Öffnen"
KeyTip="O" />
                    <RibbonSplitButton LargeImageSource="Images/Speichern.png"
Label="Speichern" KeyTip="S">
                        <RibbonMenuItem Header="Speichern" />
                        <RibbonMenuItem Header="Speichern unter" />
                        <RibbonSeparator />
                        <RibbonMenuItem Header="Kopie speichern" />
                    </RibbonSplitButton>
                </RibbonGroup>
                <RibbonGroup Header="Dateioperationen">
                    <RibbonButton LargeImageSource="Images/Drucken.png" Label="Drucken"
KeyTip="P" />
                    <RibbonButton LargeImageSource="Images/Schließen.png" Label="Schließen"
```

```
KeyTip="C" />
                </RibbonGroup>
                <RibbonGroup Header="Programmoptionen">
                    <RibbonCheckBox Label="Automatisch Speichern" KeyTip="A"
IsChecked="True" />
                    <RibbonCheckBox Label="Backups anlegen" KeyTip="B" />
                </RibbonGroup>
            </RibbonTab>
            <RibbonTab Header="Formatierung" KeyTip="F">
                <RibbonGroup Header="Schrifteinstellungen">
                    <RibbonComboBox Label="Schriftart:" IsEditable="True">
                        <RibbonGallery SelectedValue="Times" SelectedValuePath="Content"
MaxColumnCount="1">
                            <RibbonGalleryCategory>
                                <RibbonGalleryItem Content="Arial" FontFamily="Arial" />
                                <RibbonGalleryItem Content="Arial Black" FontFamily="Arial
Black" />
                                <RibbonGalleryItem Content="Serif" FontFamily="Serif" />
                                <RibbonGalleryItem Content="Sans-Serif" FontFamily="Sans-
Serif" />
                                <RibbonGalleryItem Content="Times" FontFamily="Times" />
                                <RibbonGalleryItem Content="Times New Roman"
FontFamily="Times New Roman" />
                            </RibbonGalleryCategory>
                        </RibbonGallery>
                    </RibbonComboBox>
                    <RibbonComboBox Label="Schriftgröße" IsEditable="True">
                        <RibbonGallery SelectedValue="10" SelectedValuePath="Content"
MaxColumnCount="1">
                            <RibbonGalleryCategory>
                                <RibbonGalleryItem Content="8" FontSize="8" />
                                <RibbonGalleryItem Content="10" FontSize="10" />
                                <RibbonGalleryItem Content="12" FontSize="12" />
                                <RibbonGalleryItem Content="14" FontSize="14" />
                                <RibbonGalleryItem Content="16" FontSize="16" />
                                <RibbonGalleryItem Content="18" FontSize="18" />
                            </RibbonGalleryCategory>
                        </RibbonGallery>
                    </RibbonComboBox>
                </RibbonGroup>
                <RibbonGroup Header="Schriftoptionen">
                    <RibbonToggleButton Label="Fett" SmallImageSource="Images/Fett.png"
KeyTip="F" />
                    <RibbonToggleButton Label="Kursiv"  SmallImageSource="Images/Kursiv.png"
KeyTip="K" />
                </RibbonGroup>
                <RibbonGroup Header="Farbe">
                    <RibbonButton LargeImageSource="Images/Farbe.png" Label="Vordergrund"
KeyTip="V" />
                    <RibbonButton LargeImageSource="Images/Farbe.png" Label="Hintergrund"
KeyTip="H" />
                </RibbonGroup>
                <RibbonGroup Header="Seitengröße">
                    <RibbonRadioButton SmallImageSource="Images/Seite.png" Label="DIN A3"
KeyTip="3" />
                    <RibbonRadioButton SmallImageSource="Images/Seite.png" Label="DIN A4"
KeyTip="4" IsChecked="True" />
                    <RibbonRadioButton SmallImageSource="Images/Seite.png" Label="DIN A5"
KeyTip="5" />
                </RibbonGroup>
                <RibbonGroup Header="Seitenformat">
                    <RibbonRadioButton SmallImageSource="Images/Seite.png"
Label="Hochformat" IsChecked="True" />
                    <RibbonRadioButton SmallImageSource="Images/Seite.png"
Label="Querformat" />
                </RibbonGroup>
            </RibbonTab>
        </Ribbon>
        <Grid>
            <!-- Hier kommt der eigentliche Fensterinhalt! -->
        </Grid>
    </DockPanel>
</RibbonWindow>
```

MainWindow.xaml.cs

```
using System.Windows;
using System.Windows.Controls.Ribbon;

```

```
4   namespace CSV20.WPF_Menüband
5   {
6       /// <summary>
7       /// Interaktionslogik für MainWindow.xaml
8       /// </summary>
9       public partial class MainWindow : RibbonWindow
10      {
11          public MainWindow()
12          {
13              InitializeComponent();
14          }
15
16          private void MenuExit_Click(object sender, RoutedEventArgs e)
17          {
18              Close();
19          }
20      }
21  }
```

Das große Computer ABC

C# lernen

Datenbindung für WPF: Grundlagen

Datenbindungen ermöglichen das Binden von Daten an Steuerelemente. Dadurch ist es möglich, eine Verbindung zwischen einer **Datenquelle** (z. B. einer Objektliste) und der **Benutzeroberfläche** herzustellen. Dies ist ein großer Vorteil und bietet uns eine viel bessere Möglichkeit als die „einfache“ Datenbindung von Windows Forms (z. B. Laden eines Bildes aus der internen Ressourcen-Tabelle).
Doch welche Vorteile haben wir mit Hilfe der Datenbindung? Erläutern wir einige Praxisbeispiele, um den Sinn und Zweck von Datenbindungen besser zu verstehen.
Stellen Sie sich vor, Sie haben ein Programm mit mehreren Fenstern. In allen Fenstern befindet sich am unteren Fensterrand ein *Label* für die Programm-Version. Nun verbessern oder überarbeiten Sie das Programm. Da sich hierdurch die Version geändert hat, müssten Sie alle Formulare öffnen und dort den Inhalt des Labels ändern. Durch eine **statische Ressource** müssten Sie den Wert lediglich einmal in der Ressourcen-Tabelle des Programms ändern.
Ein weiteres Beispiel: Sie haben ein Programm bei der es eine Auswahlliste gibt. Für eine bessere Übersicht wollen Sie in einem *Label* die aktuelle Auswahl des Benutzers nochmals anzeigen. Sie könnten dies über das *SelectionChanged*-Event programmiertechnisch lösen. Doch auch hier lässt sich dieses „Problem“ über eine **dynamische Bindung** einfach und elegant lösen. Zu Wissen ist, dass auch das Binden einer Variablen aus dem Daten-Kontext-Objekt der Klasse mit dieser Technik möglich ist (dazu später mehr).
Vielleicht haben Sie auch schon mal eine Auswahlliste gesehen, bei der nicht nur ein einfacher Titel, sondern verschiedene Texte und / oder Bilder angezeigt werden. Doch wie realisiert man so etwas in C# mit WPF? Hierfür bietet WPF die Definition sogenannter **Daten-Templates**. Dabei kann das Aussehen eines einzelnen Items in einem Steuerelement (z. B. einer Auswahlliste) verändert werden. Um nun das Aussehen eines einzelnen Items an unsere Datenquelle zu binden, benötigen wir ebenfalls eine Datenbindung.
Auf die einzelnen Möglichkeiten der Datenbindung gehen wir im Laufe des Kapitels genauer ein. Bei allen Datenbindungen notieren wir in XAML innerhalb der Anführungszeichen eckige Klammern. Innerhalb dieser Klammern können wir nun verschiedene **Bindungs-Anweisungen** vornehmen. Dazu mehr in den nächsten Themen.

MainWindow.xaml

```
<Window x:Class="CSV20.Statische_Ressourcen.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:p="clr-namespace:CSV20.Statische_Ressourcen.Properties"
        Title="Statische Ressourcen" Height="125" Width="300" ResizeMode="NoResize">
    <Grid>
        <Label VerticalAlignment="Top" HorizontalAlignment="Center" Margin="0,10,0,0"
               Content="{x:Static p:Resources.Titel}" FontSize="14" FontWeight="Bold" />
        <Label VerticalAlignment="Top" HorizontalAlignment="Center" Margin="0,40,0,0"
               Content="{x:Static p:Resources.Beschreibung}" FontSize="14" FontWeight="Bold"
/>
    </Grid>
</Window>
```

Das große Computer ABC

C# lernen

Datenbindung für WPF: Statische Ressourcen

Statische Ressourcen werden in der Ressourcen-Tabelle hinzugefügt und hinterlegt. Hierfür müssen wir die Eigenschaften des Projekts öffnen. In der Registerkarte „Ressourcen“ können wir nun unsere angelegten **Ressourcen editieren oder neue hinzufügen**. Statische Ressourcen werden zur Kompilierungszeit fest im Programm hinterlegt und können somit **zur Laufzeit nicht verändert** werden. Sie erlauben eine zentrale Verwaltung von Text und Bildern.
Um auf eine statische Ressource im XAML-Dokument zuzugreifen, benötigen wir das Schlüsselwort *Static* vom XAML-Namensraum. Davor müssen wir noch einen XML-Namensraum für unsere **Ressourcen-Namensraum** des Projekts festlegen (z. B. *MeinProjekt.Properties*). Nun können wir nach dem Schlüsselwort *Static* über die Klasse *Resources* des Ressourcen-Namensraums und den Namen unserer Ressource auf eine unserer definierten statischen Ressourcen zugreifen. Das Beispiel sollte diesen etwas theoretischen Text etwas verdeutlichen.
Falls Sie sich fragen, woher der Namensraum **.Properties* und die Klasse **.Resources* herkommen, dann schauen Sie sich mal die Dateien Resources.resx und Resources.Designer.cs im Ordner Properties an. Beide Dateien werden automatisch durch Visual Studio verwaltet.

DatenQuelle.cs

```
1  namespace CSV20.Ressourcen_Steuerelemente
2  {
3      public class DatenQuelle
4      {
5          public string sInhalt { get; set; }
6
7          public DatenQuelle(string sInhalt)
8          {
9              this.sInhalt = sInhalt;
10         }
11     }
12 }
```

MainWindow.xaml

```
1  <ComboBox VerticalAlignment="Top" HorizontalAlignment="Left" Width="100" Margin="10,10,0,0"
   Name="comboBoxAuswahl" SelectedIndex="0">
2      <ComboBoxItem Content="Auswahl A" />
3      <ComboBoxItem Content="Auswahl B" />
4      <ComboBoxItem Content="Auswahl C" />
5      <ComboBoxItem Content="Auswahl D" />
6      <ComboBoxItem Content="Auswahl E" />
7  </ComboBox>
8  <Label VerticalAlignment="Top" HorizontalAlignment="Left" Margin="10,50,0,0"
   FontWeight="Bold" Content="Ihre Auswahl:" />
9  <Label VerticalAlignment="Top" HorizontalAlignment="Left" Margin="120,50,0,0" Content="
   {Binding ElementName=comboBoxAuswahl, Path=SelectedItem.Content}" />
10 <Label VerticalAlignment="Top" HorizontalAlignment="Left" Margin="10,80,0,0"
   FontWeight="Bold" Content="Ihre Texteingabe: " />
11 <TextBox VerticalAlignment="Top" HorizontalAlignment="Left" Margin="120,85,0,0" Width="100"
   Text="{Binding Path=sInhalt, Mode=TwoWay}" />
```

MainWindow.xaml.cs

```
1  public MainWindow()
2  {
3      InitializeComponent();
4      this.DataContext = new DatenQuelle("Hallo Welt");
5  }
```

Das große Computer ABC

C# lernen

Datenbindung für WPF: Dynamische Bindungen

Wollen wir eine Datenbindung von einem sich **ändernden Wert** vornehmen, so benötigen wir das Schlüsselwort *Binding*. Diese Art von Bindung kann sowohl dafür genutzt werden, einen bestimmten **Wert von einem, zu einem anderen Steuerelement** zu binden, als auch einen **Steuerelement-Wert an eine Variable** (aus dem Daten-Kontext-Objekt der Klasse) zu binden.
Hinter dem Schlüsselwort *Bindung* notieren wir die Attribute *ElementName* und *Path*. Hierbei wird der Wert durch ein Gleichheitszeichen dem Attribut zugewiesen (hier dürfen keine doppelten Anführungszeichen notiert werden). *ElementName* legt den Namen des Steuerelements oder allgemein bezeichnet, den **Variablennamen des Objekts,** welches den zu bindenden Wert enthält, fest. Das *Path*-Attribut wird dazu genutzt, den Weg (oder auch Pfad genannt) zu dem zu bindenden Wert festzulegen (zumeist die Eigenschaft der Klasse).
Das Binden einer Klassen-Variable ist nicht direkt möglich. Hierfür dient das Objekt, welches in der Eigenschaft *DataContext* hinterlegt ist und somit den **Daten-Kontext der Klasse** zur Verfügung stellt. Das hier hinterlegte Objekt sollte selbst definiert werden (im Beispiel ist dies die Klasse *DatenQuelle,* wovon die Zuweisung im Konstruktor erfolgt). Die in dieser Klasse vorhandenen Variablen (nur wenn Zugriffsmodifizierer *public* ist) können nun über XAML an die Eigenschaft eines Steuerelements gebunden werden. Das Attribut *ElementName* kann hier weggelassen werden, da standardmäßig auf den Daten-Kontext der Klasse zugegriffen wird. Wenn wir die Bindung in beide „Richtungen“ nutzen wollen, sodass sich auch der Wert der Ursprungsvariable ändert, falls der gebundene Wert geändert wurde, so müssen wir das Attribut *Mode* auf den Wert *TwoWay* setzen. Dabei handelt es sich um eine Bindung mit **Lese- und Schreibzugriff**. Übrigens: Benötigen wir lediglich das *Path*-Attribut, so genügt es den Pfadnamen alleine (ohne Attributname) zu notieren.

Das große Computer ABC

C# lernen

Datenbindung für WPF: Daten-Templates

Daten-Templates dienen grundsätzlich gesehen dazu, das **Aussehen von (wiederholenden) Elementen zu verändern**. Was können wir dadurch erreichen? Ein Beispiel hierfür ist die Erstellung einer Auswahlliste (*ComboBox*), bei welcher wir ein einzelnes Element so verändern wollen, dass ein Rechteck mit einer Füllfarbe und der Titel der Farbe als ein Auswahlelement angezeigt wird.
In solchen Fällen erfolgt die Zuweisung der Elemente (bzw. deren Daten-Quelle) meist über die Code-Behind-Datei und nicht über den XAML-Code. Um auf die **Daten-Quelle der Auswahlliste** zuzugreifen, benötigen wir die Eigenschaft *ItemsSource*. Dieser Eigenschaft wird ein **Array von anzuzeigenden Elementen** zugewiesen. Bisher hatte jedes Item immer nur eine Zeichenkette, die als Beschriftung genutzt wurde. Über ein Daten-Template (Klasse *DataTemplate*) können wir nun das Design dieser Auswahlliste verändern und die einzelnen Eigenschaften unserer eigenen erstellten Klasse (welche als Daten-Kontext verwendet wird) **an einzelne Steuerelemente binden, welche im Daten-Template vorhanden sind**. In unserem Fall wollen wir das Design eines einzelnen Items ändern, wodurch wir der Eigenschaft *ItemTemplate* der *ComboBox* unser Daten-Template zuweisen müssen (siehe Beispiel).
Haben Sie sich vielleicht schon einmal gefragt, warum die Eigenschaften *SelectedItem* oder ein einzelner Eintrag des Arrays der Eigenschaft *Items* als Rückgabewert *object* haben? Zurückzuführen ist dies auf die **„Neutralität“ des Steuerelements**. Dem *ComboBox*-Steuerelement ist es egal, ob er lediglich Zeichenketten oder Objekte mit Datenbindung zugewiesen bekommt. Daraus folgt also, dass wir bei der Datenbindung an eine Auswahlliste (oder auch ein anderes Steuerelement) über beide oben genannten Eigenschaften auf das dahinterliegende Datenobjekt jederzeit zugreifen können.

ComboBoxItem.cs

```cs
using System.Windows.Media;

namespace CSV20.Ressourcen_Daten_Templates
{
    public class ComboBoxItem
    {
        public Brush Color { get; set; }
        public string Name { get; set; }
    }
}
```

MainWindow.xaml

```xaml
<ComboBox Name="comboBoxListe" Height="40">
    <ComboBox.ItemTemplate>
        <DataTemplate>
            <Canvas Height="40" Background="White">
                <Rectangle Canvas.Left="5" Canvas.Top="5" Width="30" Height="30" Fill="{Binding Color}" />
                <Label Canvas.Left="45" Canvas.Top="5" Height="30" FontSize="14" FontWeight="Bold" Content="{Binding Name}" />
            </Canvas>
        </DataTemplate>
    </ComboBox.ItemTemplate>
</ComboBox>
```

MainWindow.xaml.cs

```cs
using System.Windows;
using System.Windows.Media;

namespace CSV20.Ressourcen_Daten_Templates
{
    /// <summary>
    /// Interaktionslogik für MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        private ComboBoxItem[] aListe = new ComboBoxItem[] {
            new ComboBoxItem() {    Color = Brushes.Red,    Name = "Rot"        },
            new ComboBoxItem() {    Color = Brushes.Lime,   Name = "Grün"       },
            new ComboBoxItem() {    Color = Brushes.Blue,   Name = "Blau"       },
            new ComboBoxItem() {    Color = Brushes.White,  Name = "Weiß"       },
            new ComboBoxItem() {    Color = Brushes.Orange, Name = "Orange"     },
            new ComboBoxItem() {    Color = Brushes.Yellow, Name = "Gelb"       },
```

```
            new ComboBoxItem() {    Color = Brushes.Black,  Name = "Schwarz"    },
        };

        public MainWindow()
        {
            InitializeComponent();
        }

        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            // Objekt-Array der ComboBox-Item-Liste zuweisen
            comboBoxListe.ItemsSource = aListe;
            // erstes Element vorselektieren
            comboBoxListe.SelectedIndex = 0;
        }
    }
}
```

Das große Computer ABC

C# lernen

Erweiterte Programmierthemen: Netzwerkzugriffe

Um Netzwerkzugriffe zu programmieren, benötigen wir als erstes die Namensräume *System.Net* und *System.Net.Sockets*. Als Basis (für verschiedene Netzwerkzugriffe) wollen wir uns mit den Klassen *IPAddress*, *IPEndPoint* und *Dns* vertraut machen. *IPAddress* speichert eine **IP-Adresse** in einem Objekt. Dem Konstruktor kann sowohl eine 64bit-Zahl oder auch ein *byte*-Array übergeben werden. Die Eigenschaft *AddressFamily* ruft die **Adressen-Familie** (Enumeration *AddressFamily*: *InterNetwork* für IPv4 und *InterNetworkV6* für IPv6) ab. Zumeist nutzen wir jedoch die Funktion *TryParse(),* um eine Zeichenkette in ein IP-Adressen-Objekt umzuwandeln. Die *IPEndPoint*-Klasse stellt im Gegensatz zur *IPAddress*-Klasse einen **Netzwerk-Endpunkt** zur Verfügung, welcher **eine IP-Adresse und ein Port** enthalten. Dem Konstruktor der *IPEndPoint*-Klasse werden ein IPAddress-Objekt und ein Port (als Int-Wert) übergeben. Die *Dns*-Klasse ist statisch und stellt uns **Funktionen zur DNS-Auflösung** zur Verfügung. Die Funktion *GetHostAddresses()* erwartet als Parameter eine Zeichenkette (der aufzulösende DNS-Name) und gibt eine Array von IP-Adressen zurück. Diese Funktion arbeitet synchron, d. h. das Programm kann erst weitere Statements ausführen, sobald die Funktion vollständig abgearbeitet wurde (also nach erfolgreicher Namensauflösung oder einem Timeout). Bitte bedenken Sie, dass ggf. keine Namensauflösung erfolgt, wenn die DNS-Datenbank des Computers bereits einen passenden Eintrag enthält. Für eine asynchrone Ausführung stehen uns die Funktionen *BeginGetHostAddresses()* und *EndGetHostAddresses()* zur Verfügung.
Für **UDP-Verbindungen** steht uns die Klasse *UdpClient* zur Verfügung. Bei UDP gibt es programmiertechnisch gesehen keine Unterscheidung zwischen Client und Server, da UDP verbindungslos ist, weshalb auch für Server-Zwecke die Klasse *UdpClient* verwendet wird. Dem Konstruktor wird zumeist kein Parameter übergeben, da sich die Parameter immer auf die lokale Schnittstelle beziehen, wir jedoch (in diesem Fall) wollen, dass Windows sich die lokale Netzwerk-Schnittstelle automatisch heraussucht. Die Eigenschaft *Available* gibt die Datenmenge in Bytes an, welche am Socket (also der „Anschlussdose“) eingegangen sind. Die Funktion *Connect()* stellt eine Verbindung zu einem Netzwerk-Endpunkt her. Dieser Funktionsaufruf baut nicht wirklich eine Verbindung auf (da UDP verbindungslos ist), sondern übernimmt vielmehr den Endpunkt in die Klasse. Die Funktion *Send()* **sendet ein Daten-Paket an den Ziel-Endpunkt** (welcher mit *Connect()* übergeben wurde). Hierzu werden der Funktion ein *byte*-Array und eine Größe des Arrays übergeben. Über die Funktion *Receive()* können wir **ein Daten-Paket empfangen**. Als Parameter müssen wir einen Verweis auf eine Endpunkt-Variable übergeben. Darüber können wir prüfen, von wem das gesendete Paket stammt. Da es sich bei einem Socket um einen Stream handelt, müssen wir diesen am Ende wieder schließen (Funktion *Close()*). Das Beispiel zeigt einen (S)NTP-Client, mit welchem wir die Uhrzeit und das Datum von einem Zeitserver abrufen können. Auf das Protokoll selbst wollen wir jedoch hier nicht genauer eingehen. Nähere Informationen zum Protokoll im RFC 958.
Für **TCP-Verbindungen** stehen uns die Klassen *TcpClient* und *TcpListener* (für Server-Zwecke) zur Verfügung. Ein paar Eigenschaften und Funktionen sowie der Grundaufbau sind ähnlich zur *UdpClient*-Klasse. Bei TCP-Client-Verbindungen werden Daten über einen Stream versendet und empfangen (*GetStream()*-Funktion). Bei einem *TcpListener* können wir über die Funktion *AcceptTcpClient()* ein *TcpClient*-Objekt abrufen, um eine einkommende Nachricht zu bearbeiten. Mit Hilfe dieses Objekts können wir dann die **Nachricht des Clients abrufen**, als auch eine **Nachricht an den Client versenden**. Zu beachten ist, dass *AcceptTcpClient()* eine blockierende Funktion ist und somit auf die Anfrage eines Clients wartet. Dieses „Problem“ lässt sich über die asynchronen Funktionen *BeginAcceptTcpClient()* und *EndAcceptTcpClient()* lösen.
Um einen **Ping** (ICMP-Paket) zu senden, benötigen wir ein Objekt der Klasse *Ping*. Über die Funktion *Send()* kann ein ICMP-Paket synchron gesendet werden. Hierfür müssen wir der Funktion eine Zeichenkette, welche eine IP-Adresse oder einen Hostnamen enthält, übergeben. Die Funktion ist mehrfach überladen, wodurch wir zusätzlich den Timeout, die zu versendenden Daten und Ping-Optionen festlegen können. Die Funktion liefert ein Objekt der Klasse *PingReply* zurück. Über die Eigenschaft *Status* können wir einen Wert der Enumeration *IPStatus* abrufen (*Success* = Erfolg, *TimedOut* = Timeout, …). Über die Funktion *SendAsync()* kann ein asynchroner ICMP-Paket-Versand gestartet werden. Anders als bei anderen asynchronen Funktionen müssen wir hier das Ereignis *PingCompleted* des *Ping*-Objekts registrieren. Die Ping-Funktionalität befindet sich im Namensraum *System.Net.NetworkInformation*.

Program.cs

```
1  using System;
2  using System.Net;
3  using System.Net.Sockets;
4
5  namespace CSV20.Netzwerkzugriffe
6  {
7      class Program
8      {
9          private static string sTitle = "SNTP-Client";
10         private static string sSntpServer = "ptbtime1.ptb.de";
11
12         static void Main(string[] args)
13         {
14             IPAddress oIpAddress;
```

```
            IPEndPoint oIpEndPoint;
            UdpClient oSocket = null;
            DateTime oDateTime;
            byte[] aPacket = new byte[48];
            uint uiTimestamp;

            Console.Title = sTitle;
            Console.WriteLine(sTitle);
            Console.WriteLine();

            try
            {
                // Umwandlung von sSntpServer in eine IP-Adressen ausprobieren
                if (!IPAddress.TryParse(sSntpServer, out oIpAddress))
                {
                    // sSntpServer ist ein DNS-Name, starte Auflösung
                    oIpAddress = Dns.GetHostAddresses(sSntpServer)[0];
                    Console.WriteLine("DNS-Name {0} in {1} aufgelöst!", sSntpServer,
oIpAddress.ToString());
                }

                // IP-Endpunkt erstellen
                oIpEndPoint = new IPEndPoint(oIpAddress, 123);

                // Socket öffnen
                oSocket = new UdpClient();
                oSocket.Connect(oIpEndPoint);

                // leeres Packet mit Kennezeichnung an Server senden
                aPacket[0] = 0x1B;              // NTP-Version 3, Client-Modus (siehe RFC
5905)
                oSocket.Send(aPacket, 48);

                // Daten vom Server abholen
                aPacket = oSocket.Receive(ref oIpEndPoint);

                // UTC-Sekunden aus NTP-Packet laden
                uiTimestamp = BitConverter.ToUInt32(aPacket, 40);
                uiTimestamp = (((uiTimestamp & 0x000000ff) << 24) +
                               ((uiTimestamp & 0x0000ff00) <<  8) +
                               ((uiTimestamp & 0x00ff0000) >>  8) +
                               ((uiTimestamp & 0xff000000) >> 24));

                // in Datum- / Uhrzeit-Objekt umwandeln und ausgeben
                oDateTime = new DateTime(1900, 1, 1).AddSeconds(uiTimestamp);
                Console.WriteLine("UTC-Zeit: " + oDateTime.ToString());
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.ToString());
            }
            finally
            {
                // Socket schließen
                if (oSocket != null)
                    oSocket.Close();
            }

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

Das große Computer ABC

C# lernen

Erweiterte Programmierthemen: Excel-Dokumente lesen und bearbeiten

C# bietet unter anderem eine **Schnittstelle zu den Office-Programmen** von Microsoft. Ein praktisches Beispiel hierfür ist das Auslesen und die Editierung einer Excel-Tabelle. Die hierfür benötigten Klassen befinden sich im Namensraum *Microsoft.Office.Interop.Excel*. Zusätzlich ist noch ein Verweis auf „Microsoft Excel xx-Objectlibrary“ nötig, welcher sich in der Gruppe „COM“ befindet.
Die Klasse *Application* (nicht zu verwechseln mit der Klasse *Application* des Namensraums *System.Windows.Forms*) stellt unser Excel-Programm dar und kommuniziert also mit der Excel-Applikation (Excel.exe). Über die Eigenschaft *Workbooks* können wir auf **Arbeitsmappen** zugreifen. Mit der Funktion *Open()* öffnen wir eine Arbeitsmappe. Hierfür übergeben wir der Funktion den Dateinamen (evtl. mit Pfad). Als Rückgabe erhalten wir ein *Workbook*-Objekt. Dieses stellt eine einzelne Arbeitsmappe bzw. die Datei dar.
Nun müssen wir nur noch auf die einzelnen **Tabellen** der Arbeitsmappe zugreifen. Die Eigenschaft *Worksheets* enthält eine Auflistung von allen Tabellen der Arbeitsmappe. Über die Eigenschaft *ActiveSheet* können wir die aktuelle Tabelle (zumeist die zuletzt bearbeitete Tabelle) abrufen.
Die Klasse *Range* dient zur Selektion eines **Zellbereichs** innerhalb unserer Tabelle. Mit Hilfe der Eigenschaft *UsedRange* des *Worksheet*-Objekts erhalten wir ein *Range*-Objekt, welches den vom Benutzer verwendeten Bereich angibt. Die Eigenschaft *Rows* stellt eine Auflistung (Array) von allen Zeilen zur Verfügung, *Columns* hingegen die Auflistung aller Spalten. Über die Eigenschaft *Cells* können wir auf eine einzelne Zelle zugreifen. Hierzu übergeben wir zwei Indexe (da es sich bei der Eigenschaft um ein zwei-dimensionales Array handelt), welche beide 1-basierend sind (nicht 0-basierend). Die Eigenschaft *Value* gibt den **Zellwert** zurück. Diese Eigenschaft kann nicht nur gelesen, sondern auch gesetzt werden. Um unser **Dokument zu speichern**, benötigen wir die Funktion *Save()* des *Workbook*-Objekts. Bitte denken Sie am Ende auch daran, die Arbeitsmappe mit Hilfe der Funktion *Close()* zu schließen.
Im Beispiel verwenden wir eine Windows Forms Applikation mit einem *DataGridView*-Steuerelement. Mit diesem ist es möglich, Daten tabellarisch darzustellen (also ähnlich wie bei Excel). Das Beispiel-Programm liest die aktuelle Tabelle der gewählten Arbeitsmappe aus. Bei Bedarf können die Zellwerte verändert und die Inhalte über einen Button gespeichert werden.

Form1.cs

```
1  using System;
2  using System.Windows.Forms;
3  using Microsoft.Office.Interop.Excel;   // zusätzlich ist der Projektverweis auf COM ->
   Microsoft Excel xx-Objectlibrary
4
5  namespace CSV20.Excel
6  {
7      public partial class Form1 : Form
8      {
9          // oExcelApp benötigt den kompletten Namensraum, da andernfalls der Compiler den
   Unterschied zwischen
10         //  "Microsoft.Office.Interop.Excel.Application" und
   "System.Windows.Forms.Application" nicht erkennen kann
11         private Microsoft.Office.Interop.Excel.Application oExcelApp = new
   Microsoft.Office.Interop.Excel.Application();
12         private Workbook oWorkbook = null;
13         private Worksheet oWorksheet = null;
14
15         public Form1()
16         {
17             InitializeComponent();
18         }
19
20         private void Form1_Load(object sender, EventArgs e)
21         {
22             Range oRange;
23             string sSpaltenName;
24             object oValue;
25
26             if (openFileDialogExcel.ShowDialog() == System.Windows.Forms.DialogResult.OK)
27             {
28                 // Datei öffnen und aktives Blatt und benutzten Bereich auswählen
29                 oWorkbook = oExcelApp.Workbooks.Open(openFileDialogExcel.FileName);
30                 oWorksheet = (Worksheet)oWorkbook.ActiveSheet;
31                 oRange = oWorksheet.UsedRange;
32
33                 // Zellen vom Dokument in die Ansicht laden
34                 for (int i = 0; i < oRange.Columns.Count; i++)
```

```
                {
                    if (i >= dataGridViewExcel.Columns.Count)
                    {
                        // bilden der Spalten-Namen: A, B, C, D, E, ..., A1, B1, C1, ..., A2, B2, ...
                        sSpaltenName = ((char)('A' + i % 26)).ToString();
                        if (Math.Floor(i / 26.0) != 0)
                            sSpaltenName += Math.Floor(i / 26.0).ToString();
                        dataGridViewExcel.Columns.Add(sSpaltenName, sSpaltenName);
                    }

                    for (int j = 0; j < oRange.Rows.Count; j++)
                    {
                        // Zeilen werden nur beim 1. Durchlauf hinzugefügt
                        if (i == 0)
                            dataGridViewExcel.Rows.Add();
                        oValue = (oRange.Cells[j + 1, i + 1] as Range).Value;
                        // muss unbedingt abgefangen werden
                        if (oValue != null)
                            dataGridViewExcel.Rows[j].Cells[i].Value = oValue.ToString();
                    }
                }

                // Fenster minimieren und danach wieder in normale Fenstergröße zurückkehren,
                // dies ist ein Trick, sodass wir die Anwendung wieder in den Vordergrund bekommen,
                // da Excel unsichtbar geöffnet wird
                this.WindowState = FormWindowState.Minimized;
                this.WindowState = FormWindowState.Normal;
            }
            else // Bei Abbruch, Fenster schließen
                Close();
        }

        private void Form1_FormClosing(object sender, FormClosingEventArgs e)
        {
            // Beim Schließen des Fensters, Dokument schließen
            if (oWorkbook != null)
                oWorkbook.Close();
        }

        private void buttonSpeichern_Click(object sender, EventArgs e)
        {
            if (MessageBox.Show("Sind Sie sicher dass Sie die Änderungen übernehmen möchten?\nDie Quelldatei wird überschrieben!", "Speichern?", MessageBoxButtons.YesNo, MessageBoxIcon.Question) == System.Windows.Forms.DialogResult.Yes)
            {
                // Daten von der Ansicht in das Dokument schreiben
                for (int i = 0; i < dataGridViewExcel.Rows.Count - 1; i++)
                    for (int j = 0; j < dataGridViewExcel.Columns.Count; j++)
                        if (dataGridViewExcel.Rows[i].Cells[j].Value != null)
                            (oWorksheet.Cells[i + 1, j + 1] as Range).Value = dataGridViewExcel.Rows[i].Cells[j].Value.ToString();
                        else
                            (oWorksheet.Cells[i + 1, j + 1] as Range).Value = "";

                // Speichervorgang auslösen
                oWorkbook.Save();
            }
        }
    }
}
```

Das große Computer ABC

C# lernen

Erweiterte Programmierthemen: Diagramme

Eine Funktionalität zum **Zeichnen von Diagrammen** mit Windows Forms befindet sich im Namensraum *System.Windows.Forms.DataVisualization.Charting* und benötigt das .NET-Framework 4.0 oder höher. Das Hauptelement des Diagramms ist *Chart*.
Die Klasse *Series* stellt eine Wertserie zur Verfügung. Ein Diagramm kann mehrere Serien (oder auch **Datenreihen** genannt) haben. Hierfür müssen wir der Eigenschaft *Series* des *Chart*-Objekts, Datenreihen mit Hilfe der Funktion *Add()* zuweisen. Die Klasse *Series* selbst besitzt die Eigenschaft *Points*. Hier kann über die Eigenschaft *AddXY()* ein Wert für die X- und Y-Achse und somit ein einzelner **Datensatz**, hinzugefügt werden. Die Art der Serie (also Linie, Balken, etc.) kann über die Eigenschaft *ChartType* und einen Wert der Enumeration *SeriesChartType* festgelegt werden. In C# ist es dadurch möglich, dass ein Diagramm mehrere Serien besitzt, welche unterschiedliche Diagrammarten haben.

Form1.cs

```
using System;
using System.Windows.Forms;
using System.Windows.Forms.DataVisualization.Charting;      // wichtig: benötigt das
.NET-Framework 4.0 oder höher

namespace CSV20.Diagramme
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            Random oZufall = new Random();

            Series oSerieA1 = new Series("Serie A1");
            for (int i = 1; i < 25; i++)
                oSerieA1.Points.AddXY(i, oZufall.Next(0, 1001));
            chart1.Series.Add(oSerieA1);

            Series oSerieA2 = new Series("Serie A2");
            for (int i = 1; i < 25; i++)
                oSerieA2.Points.AddXY(i, oZufall.Next(0, 1001));
            chart1.Series.Add(oSerieA2);

            Series oSerieA3 = new Series("Serie A3");
            for (int i = 1; i < 25; i++)
                oSerieA3.Points.AddXY(i, oZufall.Next(0, 1001));
            chart1.Series.Add(oSerieA3);

            Series oSerieB1 = new Series("Serie B1");
            oSerieB1.ChartType = SeriesChartType.Line;
            oSerieB1.BorderWidth = 3;
            for (int i = 1; i < 25; i++)
                oSerieB1.Points.AddXY(i, oZufall.Next(0, 1001));
            chart1.Series.Add(oSerieB1);

            Series oSerieB2 = new Series("Serie B2");
            oSerieB2.ChartType = SeriesChartType.Line;
            oSerieB2.BorderWidth = 3;
            for (int i = 1; i < 25; i++)
                oSerieB2.Points.AddXY(i, oZufall.Next(0, 1001));
            chart1.Series.Add(oSerieB2);
        }
    }
}
```

Das große Computer ABC

C# lernen

Erweiterte Programmierthemen: Ordnerüberwachung

Die *FileSystemWatcher*-Klasse bietet uns die Möglichkeit, über Änderungen an Ordnern und Dateien benachrichtigt zu werden. Dabei „überwacht“ die Klasse einen bestimmten Ordner (und deren Dateien und evtl. Unterordner), welcher über den Konstruktor oder die Eigenschaft *Path* gesetzt werden kann. Mit Hilfe der Eigenschaft *Filter* können wir einen **Filter für Datei- und Ordnernamen** angeben. Mehrere Filter werden mit dem Senkrechtstrich (|) getrennt. Eine Angabe eines Teilnamens (z. B. *.txt oder CSharp_*.txt) ist mit Hilfe des Platzhalters (*) möglich. Wollen wir **Unterordner** bei der Überwachung mit einbeziehen, so muss die Eigenschaft *IncludeSubdirectories* auf *true* gesetzt werden. Der Standardwert der Eigenschaft ist *false*.
Um auf die verschiedenen Änderungen an den Dateien oder Ordnern zu reagieren, können wir die **Ereignisse** *Changed* (Datei geändert), *Created* (Datei oder Ordner erstellt), *Deleted* (Datei oder Ordner gelöscht) und *Renamed* (Datei oder Ordner umbenannt) registrieren. Den Events werden ein *FileSystemEventArgs*-Objekt als Event-Argument übergeben. Über die Eigenschaft *ChangedType* können wir die Art der Änderungen (also ändern, erstellen, löschen oder umbenennen) als Wert der Enumeration *WatcherChangeTypes* abrufen. Die Eigenschaft *FullPath* gibt den Pfad mit Datei- oder Ordnernamen der geänderten Datei oder des Ordners zurück. Durch die Eigenschaft *Name* können wir hingegen nur den Datei- oder Ordnernamen abrufen. Beim *Renamed*-Event empfiehlt es sich, das Event mit dem *RenamedEventHandler* zu registrieren, sodass wir als Event-Argument ein *RenamedEventArgs*-Objekt übergeben bekommen. Hierdurch können wir zusätzlich über die Eigenschaft *OldFullPath* und *OldName* den Namen oder Pfad vor der Umbenennung abrufen.
Standardmäßig ist die *FileSystemWatcher*-Klasse deaktiviert. Wollen wir diese aktivieren, sodass Ereignisse ausgelöst werden falls Dateiänderungen auftreten, so muss die Eigenschaft *EnableRaisingEvents* auf *true* gesetzt werden. Das Beispielprogramm überwacht alle Dateien und Ordner im ausgewählten Ordner. Wählen Sie doch einmal das komplette Laufwerk, auf welchem das Betriebssystem installiert ist. Sie werden erstaunt sein, wie viele Dateioperationen bei verschiedenen Programmaufrufen (z. B. bei Mozilla Firefox) oder auch im Leerlauf ausgeführt werden.
Im Beispiel verwenden wir die Funktion *Select()* der *TextBox,* in welcher wir Logausgaben zu den Dateioperationen ausführen, um den Textzeiger zu setzen (im Beispiel ans Ende). Die Funktion *ScrollToCaret()* scrollt bis zur Position, an welcher sich der Cursor befindet. Wie Ihnen vermutlich auffallen wird, befindet sich der komplette Code in einem Block mit dem Funktionsaufruf *Invoke()*. Dies ist zwingend notwendig, da die grafische Oberfläche auf einem anderen Task arbeitet wie die *FileSystemWatcher*-Klasse. *Invoke()* erlaubt den Zugriff auf die Elemente der grafischen Oberfläche von einem anderen Task heraus. Hierfür muss ein *delegate*-Block deklariert werden, welcher für die *Invoke()*-Funktion in *MethodInvoker* gecastet werden muss.

Form1.cs

```
1  using System;
2  using System.IO;
3  using System.Windows.Forms;
4
5  namespace CSV20.Ordnerüberwachung
6  {
7      public partial class Form1 : Form
8      {
9          private FileSystemWatcher oFileWatch = new FileSystemWatcher();
10
11         public Form1()
12         {
13             InitializeComponent();
14         }
15
16         private void Form1_Load(object sender, EventArgs e)
17         {
18             if (folderBrowserDialogOrdner.ShowDialog() ==
   System.Windows.Forms.DialogResult.OK)
19             {
20                 // Ordner-Pfad und Filter setzen
21                 oFileWatch.Path = folderBrowserDialogOrdner.SelectedPath;
22                 oFileWatch.Filter = "*.*";
23                 // Events registrieren (wir könnten auch für jedes Event eine eigene Event-
   Funktion verwenden)
24                 oFileWatch.Created += new FileSystemEventHandler(oFileWatch_Action);
25                 oFileWatch.Changed += new FileSystemEventHandler(oFileWatch_Action);
26                 oFileWatch.Deleted += new FileSystemEventHandler(oFileWatch_Action);
27                 // Event für Umbennung registrieren (wir benötigen die RenamedEventArgs um
   den
28                 // alten und neuen Namen der Datei / des Ordners zu erhalten)
29                 oFileWatch.Renamed += new RenamedEventHandler(oFileWatch_ActionRename);
30                 // Unterverzeichnisse mit einbeziehen
31                 oFileWatch.IncludeSubdirectories = true;
```

```
                // Events aktivieren
                oFileWatch.EnableRaisingEvents = true;
                // Fenster minimieren und danach wieder in normale Fenstergröße zurückkehren,
                // dies ist ein Trick, sodass wir die Anwendung wieder in den Vordergrund
bekommen
                this.WindowState = FormWindowState.Minimized;
                this.WindowState = FormWindowState.Normal;
            }
            else // Bei Abbruch, Fenster schließen
                Close();
        }

        void oFileWatch_Action(object sender, FileSystemEventArgs e)
        {
            this.Invoke((MethodInvoker)delegate
            {
                textBoxLog.Text += DateTime.Now.ToString();

                switch (e.ChangeType)
                {
                    case WatcherChangeTypes.Created:
                        textBoxLog.Text += " Erstellt ";
                        break;

                    case WatcherChangeTypes.Changed:
                        textBoxLog.Text += " Geändert ";
                        break;

                    case WatcherChangeTypes.Deleted:
                        textBoxLog.Text += " Gelöscht ";
                        break;
                }

                textBoxLog.Text += e.FullPath + "\r\n";

                textBoxLog.Select(textBoxLog.Text.Length, 0);
                textBoxLog.ScrollToCaret();
            });
        }

        void oFileWatch_ActionRename(object sender, RenamedEventArgs e)
        {
            this.Invoke((MethodInvoker)delegate
            {
                textBoxLog.Text += DateTime.Now.ToString();

                textBoxLog.Text += " Umbenannt ";

                textBoxLog.Text += e.OldFullPath + " zu ";

                textBoxLog.Text += e.FullPath + "\r\n";

                textBoxLog.Select(textBoxLog.Text.Length, 0);
                textBoxLog.ScrollToCaret();
            });
        }
    }
}
```

Das große Computer ABC

C# lernen

Erweiterte Programmierthemen: Klassenbibliotheken

Eine Klassenbibliothek (auch DLL genannt, für Dynamic Link Library) erlaubt die **Auslagerung von Klassen und Namensräumen** in einer separaten Datei. Hierbei erhalten wir nicht wie bei einer Konsolen-, Windows Forms- oder WPF-Anwendung eine .exe-Datei, sondern eine .dll-Datei. Beim Erstellen eines Projekts mit Visual Studio kann hierbei der Typ „Klassenbibliothek“ verwendet werden. Eine DLL kann **nicht direkt ausgeführt werden**, da diese keinen Einstiegspunkt besitzt. Somit handelt es sich bei einer DLL also um ein nicht ausführbares Programm. Jedoch können wir über Verweise oder einen Import **DLLs in unser Programm einbinden**.
Der **Import von DLLs** wird nur für Klassenbibliotheken benötigt, welche nicht in .NET-Sprachen (also C# oder Visual Basic) programmiert wurden (sondern z. B. in C oder C++). Hierfür müssen wir [DllImport("MeineDll.dll")] notieren. Unterhalb diesem sogenannten Attribut (also in der nächsten Zeile) erfolgt nun eine Funktionsdeklaration, jedoch mit dem Schlüsselwort *extern*, welches zwischen Zugriffsmodifizierer und Rückgabetyp angegeben werden muss. Zusätzlich muss der Namensraum *System.Runtime.InteropServices* per *using*-Schlüsselwort eingebunden werden.
Um eine **.NET-Klassenbibliothek einzubinden,** müssen wir über den Verweis-Explorer die DLL einbinden (mit Hilfe des „Dursuchen“-Reiters). Dabei wird die DLL beim Kompilieren automatisch in den bin/Debug oder bin/Release Ordner kopiert. Befindet sich unser Klassenbibliotheks-Projekt in der gleichen Projektmappe wie unsere Anwendung, in welcher wir diese verwenden wollen, so kann direkt ein Verweis auf ein Projekt der Projektmappe erstellt werden.
DLLs sind jedoch nicht nur für die Auslagerung und Strukturierung von Programmcode gedacht, sondern dienen vielmehr auch zur **Wiederverwendung**. So können Sie die gleiche DLL in verschiedenen Anwendungen einbinden und verwenden.
Das Beispiel zeigt einen einfachen Taschenrechner für zwei Zahlen. In der DLL gibt es Funktionen zur Berechnung (Plus, Minus, Mal und Geteilt) sowie Funktionen zur Prüfung der eingegebenen Zahl. Das Hauptprogramm enthält hingegen die Textfelder und Auswahlfelder. Über den Button-Klick wird die *Checked*-Eigenschaft der *RadioButton*-Steuerelemente überprüft und dementsprechend Funktionen der Klassenbibliothek aufgerufen.

Calculator.cs

```
1   namespace CSV20.Klassenbibliothek_Library
2   {
3       public class Calculator
4       {
5           public static int Addition(int iZahl1, int iZahl2)
6           {
7               return iZahl1 + iZahl2;
8           }
9
10          public static double Addition(double dZahl1, double dZahl2)
11          {
12              return dZahl1 + dZahl2;
13          }
14
15          public static int Subtraktion(int iZahl1, int iZahl2)
16          {
17              return iZahl1 - iZahl2;
18          }
19
20          public static double Subtraktion(double dZahl1, double dZahl2)
21          {
22              return dZahl1 - dZahl2;
23          }
24
25          public static int Multiplikation(int iZahl1, int iZahl2)
26          {
27              return iZahl1 * iZahl2;
28          }
29
30          public static double Multiplikation(double dZahl1, double dZahl2)
31          {
32              return dZahl1 * dZahl2;
33          }
34
35          public static int Division(int iZahl1, int iZahl2)
36          {
37              return iZahl1 / iZahl2;
38          }
39
40          public static double Division(double dZahl1, double dZahl2)
41          {
```

```
            return dZahl1 / dZahl2;
        }
    }
}
```

Number.cs

```
namespace CSV20.Klassenbibliothek_Library
{
    public class Number
    {
        public static bool IsNumber(string sEingabe, out int iZahl)
        {
            return int.TryParse(sEingabe, out iZahl);
        }

        public static bool IsNumber(string sEingabe, out double dZahl)
        {
            return double.TryParse(sEingabe, out dZahl);
        }

        public static bool DivisionErlaubt(int iZahl)
        {
            return iZahl != 0;
        }

        public static bool DivisionErlaubt(double dZahl)
        {
            return dZahl != 0;
        }
    }
}
```

Form1.cs

```
using System;
using System.Windows.Forms;
using CSV20.Klassenbibliothek_Library;

namespace CSV20.Klassenbibliothek_Form
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void buttonBerechnen_Click(object sender, EventArgs e)
        {
            int iZahl1, iZahl2, iZahlErgebnis;
            double dZahl1, dZahl2, dZahlErgebnis;

            if (radioButtonGanz.Checked)
            {
                if (Number.IsNumber(textBoxZahl1.Text, out iZahl1))
                {
                    if (Number.IsNumber(textBoxZahl2.Text, out iZahl2))
                    {
                        if (radioButtonAdd.Checked)
                            iZahlErgebnis = Calculator.Addition(iZahl1, iZahl2);
                        else if (radioButtonSub.Checked)
                            iZahlErgebnis = Calculator.Subtraktion(iZahl1, iZahl2);
                        else if (radioButtonMul.Checked)
                            iZahlErgebnis = Calculator.Multiplikation(iZahl1, iZahl2);
                        else // if (radioButtonDiv.Checked)
                        {
                            if (Number.DivisionErlaubt(iZahl2))
                            {
                                iZahlErgebnis = Calculator.Division(iZahl1, iZahl2);
                            }
                            else
                            {
                                MessageBox.Show("Die Division ist mit der eingegebenen Zahl
nicht möglich!", "Rechenoperation ungültig", MessageBoxButtons.OK, MessageBoxIcon.Error);
                                return; // muss gemacht werden, da andernfalls das Ergebnis
ausgegeben wird!
                            }
                        }
                        MessageBox.Show("Ergebnis: " + iZahlErgebnis, "Ergebnis",
```

```
44  MessageBoxButtons.OK, MessageBoxIcon.Information);
45                      }
46                      else
                            MessageBox.Show("Die Zahl 2 ist keine gültige Zahl!", "Zahl
47  ungültig", MessageBoxButtons.OK, MessageBoxIcon.Error);
48                  }
49                  else
                        MessageBox.Show("Die Zahl 1 ist keine gültige Zahl!", "Zahl ungültig",
50  MessageBoxButtons.OK, MessageBoxIcon.Error);
51              }
52              else // if (radioButtonKomma.Checked)
53              {
54                  if (Number.IsNumber(textBoxZahl1.Text, out dZahl1))
55                  {
56                      if (Number.IsNumber(textBoxZahl2.Text, out dZahl2))
57                      {
58                          if (radioButtonAdd.Checked)
59                              dZahlErgebnis = Calculator.Addition(dZahl1, dZahl2);
60                          else if (radioButtonSub.Checked)
61                              dZahlErgebnis = Calculator.Subtraktion(dZahl1, dZahl2);
62                          else if (radioButtonMul.Checked)
63                              dZahlErgebnis = Calculator.Multiplikation(dZahl1, dZahl2);
64                          else // if (radioButtonDiv.Checked)
65                          {
66                              if (Number.DivisionErlaubt(dZahl2))
67                              {
68                                  dZahlErgebnis = Calculator.Division(dZahl1, dZahl2);
69                              }
70                              else
71                              {
                                    MessageBox.Show("Die Division ist mit der eingegebenen Zahl
72  nicht möglich!", "Rechenoperation ungültig", MessageBoxButtons.OK, MessageBoxIcon.Error);
                                    return; // muss gemacht werden, da andernfalls das Ergebnis
73  ausgegeben wird!
74                              }
75                          }
                            MessageBox.Show("Ergebnis: " + dZahlErgebnis, "Ergebnis",
76  MessageBoxButtons.OK, MessageBoxIcon.Information);
77                      }
78                      else
                            MessageBox.Show("Die Zahl 2 ist keine gültige Zahl!", "Zahl
79  ungültig", MessageBoxButtons.OK, MessageBoxIcon.Error);
80                  }
81                  else
                        MessageBox.Show("Die Zahl 1 ist keine gültige Zahl!", "Zahl ungültig",
82  MessageBoxButtons.OK, MessageBoxIcon.Error);
83              }
84          }
85      }
    }
```

Das große Computer ABC

C# lernen

Erweiterte Programmierthemen: Dienste

Dienste ermöglichen es, die Anwendung, welche zumeist unsichtbar ist, als **Hintergrundprozess** arbeiten zu lassen. Sie sind praktisch nicht da und übernehmen trotzdem eine wichtige, teilweise sogar notwendige Funktionalität. Auch mit C# kann ein solcher Dienst (in Englisch auch Service genannt) programmiert werden. Für was könnte so etwas nützlich sein? Grundsätzlich werden für die meisten Anwendungs-Zwecke keine Dienste benötigt. Einige Hobby-Programmierer werden einen Dienst möglicherweise nie programmieren. Trotzdem lohnt es sich zu wissen, wie man einen Dienst programmiert. Nützlich ist ein Dienst immer dann, wenn etwas **überwacht** werden soll (z. B. Protokollierung von freiem Festplattenspeicher, Erreichbarkeit von Netzwerkgeräten usw.). Bei der Erstellung des Projekts müssen wir „Windows-Dienst“ (in der Gruppe „Windows-Desktop“) wählen. Hierdurch erhalten wir bereits ein vorgefertigtes Gerüst als Grundaufbau.
Die Datei Program.cs enthält (wie bereits bekannt) die *Main()*-Funktion, jedoch ohne Übergabeparameter. In der Funktion werden mehrere Dienste, welche von der Klasse *ServiceBase* abstammen, in einem Array deklariert und der statischen Funktion *Run()* der *ServiceBase*-Klasse übergeben. Hierdurch wäre es möglich, die Dienst-Anwendung in mehrere Dienste zu unterteilen. Zumeist findet sich hier jedoch nur ein Eintrag.
Jeder Dienst verfügt, ähnlich wie bei Windows Forms, über die Datei Service1.cs und Service1.Designer.cs. Natürlich kann auch hier der Name geändert werden (rechte Maustaste und Umbenennen). Die Service1.Desinger.cs-Datei enthält so gut wie keinen Programmcode und ist für uns eher uninteressant. In der Service1.Designer.cs-Datei gibt es nun die überschriebenen Methoden (Schlüsselwort *override*) *OnStart()* und *OnStop()*. *OnStart()* wird beim **Starten des Dienstes** aufgerufen und bekommt als Übergabe ein Array mit Parametern übergeben. *OnStop()* wird beim **Stoppen des Dienstes** ausgeführt. Übliche Praxis ist es, eine eigene Klasse zu erstellen, welche ebenfalls über eine Start- und Stopp-Funktion verfügen und diese über die *OnStart()* und *OnStop()* aufzurufen. Die im Dienst ausgeführten Statements sollten immer in einem **eigenen Thread** ausgeführt werden und dürfen niemals als synchroner (auch blockierend genannt) Code in der *OnStart()*-Funktion notiert werden.
Im Beispiel haben wir einen Dienst programmiert, welcher minütlich die CPU- und RAM-Last protokolliert. Das ganze wird über eine *while*-Schleife und eine Abfrage gelöst. Die *while*-Schleife prüft ein globales Bit (*bool*-Variable), welches über die Stopp-Funktion auf *false* gesetzt wird, um somit den Dienst anzuhalten. Das Protokoll wird in dem Verzeichnis der .exe-Datei (dies kann über *AppDomain.CurrentDomain.BaseDirectory* abgerufen werden) in der Datei log.txt gespeichert. In der *while*-Schleife notieren wir einen Aufruf der *Sleep()*-Funktion der *Thread*-Klasse. Dies ist zu empfehlen, da andernfalls die CPU-Last des Computers stark zunehmen würde, weil der Computer immer damit beschäftigt ist, die Zeitprüfung durchzuführen.
Die **Installation eines Dienstes** erfolgt mit dem Tool **InstallUtil.exe,** welches im Ordner des .NET-Frameworks enthalten ist (zumeist C:WindowsMicrosoft.NETFrameworkv4.0.30319). Bei 64bit-Systemen gibt es zudem den Ordner Framework64. Die Versionsnummer kann sich ebenfalls unterscheiden. Als Parameter müssen wir den Pfad zur .exe-Datei des Dienstes übergeben. Soll der Dienst deinstalliert werden, muss der Parameter /u vor dem Pfad angegeben werden. Der Aufruf erfolgt über die Windows-Eingabeaufforderung oder auch über die **Developer-Eingabeaufforderung für Visual Studio**.
Falls Sie sich fragen, woher der Dienst nun seine Einstellungen bekommt (z. B. Name oder Beschreibung), dann sollten Sie sich die Datei DienstInstallation.cs anschauen. Diese Datei wurde von uns hinzugefügt und empfiehlt sich, in jedem Dienst einzubauen, denn damit ist es möglich, dass sich der **komplette Dienst mit allen Einstellungen über das InstallUtil-Tool installieren lässt**. Die Klasse stammt von der *Installer*-Klasse (Namensraum *System.Configuration.Install*) ab. Im Konstruktor erfolgt nun die eigentliche Installation, bei welcher der Starttyp (Enumeration *ServiceStartMode*), der Name (ohne Sonderzeichen), der Anzeige-Name und die Beschreibung einem Objekt der *ServiceInstaller*-Klasse übergeben werden. Zusätzlich ist noch die Klasse *ServiceProcessInstaller* notwendig, um den Anmelde-Typ festzulegen (Enumeration *ServiceAccount*). Beide Objekte müssen der statischen Funktion *Add()* der *Installers*-Klasse übergeben werden.
Bitte bedenken Sie, dass die Vorlage für eine Dienst-Anwendung bei den Express-Editionen von Visual Studio 2013 und vorherigen Versionen fehlen. Ein Dienst kann jedoch auch mit Hilfe der Vorlage „leeres Projekt“ und dem Beispielcode dieses Projekts erstellt werden.

Program.cs

```
1  using System.ServiceProcess;
2
3  namespace CSV20.Dienste
4  {
5      static class Program
6      {
7          /// <summary>
8          /// Der Haupteinstiegspunkt für die Anwendung.
9          /// </summary>
10         static void Main()
11         {
12             ServiceBase[] ServicesToRun;
13             ServicesToRun = new ServiceBase[]
14             {
15                 new Service1()
16             };
```

```
            ServiceBase.Run(ServicesToRun);
        }
    }
}
```

Service1.cs

```
using System.ServiceProcess;
using System.Threading;

namespace CSV20.Dienste
{
    public partial class Service1 : ServiceBase
    {
        private PerformanceLogger oLogger = new PerformanceLogger();

        public Service1()
        {
            InitializeComponent();
        }

        protected override void OnStart(string[] args)
        {
            Thread oThread = new Thread(new ThreadStart(oLogger.StartLogging));
            oThread.Start();
        }

        protected override void OnStop()
        {
            oLogger.StoppeLogging();
        }
    }
}
```

Service1.Designer.cs

```
namespace CSV20.Dienste
{
    partial class Service1
    {
        /// <summary>
        /// Erforderliche Designervariable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Verwendete Ressourcen bereinigen.
        /// </summary>
        /// <param name="disposing">True, wenn verwaltete Ressourcen gelöscht werden sollen;
andernfalls False.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Vom Komponenten-Designer generierter Code

        /// <summary>
        /// Erforderliche Methode für die Designerunterstützung.
        /// Der Inhalt der Methode darf nicht mit dem Code-Editor geändert werden.
        /// </summary>
        private void InitializeComponent()
        {
            components = new System.ComponentModel.Container();
            this.ServiceName = "Service1";
        }

        #endregion
    }
}
```

PerformanceLogger.cs

```
using System;
using System.Diagnostics;
using System.IO;
```

```
using System.ServiceProcess;
using System.Threading;

namespace CSV20.Dienste
{
    public class PerformanceLogger
    {
        private bool bLoggingRunning;

        public void StartLogging()
        {
            DateTime oLetzterZeitpunkt;
            PerformanceCounter oPerformanceCpu;
            PerformanceCounter oPerformanceRam;
            DateTime oAktuellerZeitpunkt;
            string sAusgabe;

            // globale Variable setzen
            bLoggingRunning = true;

            // interne Variablen setzen
            oLetzterZeitpunkt = DateTime.Now - TimeSpan.FromMinutes(1); // dadurch wird die 1. Protokollierung sofot ausgefürht
            oPerformanceCpu = new PerformanceCounter("Processor",    "% Processor Time",         "_Total");
            oPerformanceRam = new PerformanceCounter("Memory",       "% Committed Bytes In Use");

            // Task ausführen (solange bis StoppeLogging() aufgerufen wird)
            while (bLoggingRunning)
            {
                // aktuelle Zeit ermitteln
                oAktuellerZeitpunkt = DateTime.Now;
                // prüfen ob 1 Minuten seit letztem Aufruf vergangen sind
                if (oAktuellerZeitpunkt - oLetzterZeitpunkt > TimeSpan.FromMinutes(1))
                {
                    try
                    {
                        sAusgabe = oAktuellerZeitpunkt.ToString() + ": ";
                        sAusgabe += "CPU " + oPerformanceCpu.NextValue().ToString("F2") + "%";
                        sAusgabe += " RAM " + oPerformanceRam.NextValue().ToString("F2") + "%";
                        sAusgabe += Environment.NewLine;    // enthält standardmäßig \r\n
                        File.AppendAllText(AppDomain.CurrentDomain.BaseDirectory + "\\log.txt", sAusgabe);
                    }
                    catch (Exception ex)
                    {
                        // Fehlerbehandlung
                    }
                    // neuen "letzten" Zeitpunkt merken
                    oLetzterZeitpunkt = oAktuellerZeitpunkt;
                }
                // 1 Sekunde warten
                Thread.Sleep(1000);
            }
        }

        public void StoppeLogging()
        {
            bLoggingRunning = false;
        }
    }
}
```

DienstInstallation.cs

```
using System.ComponentModel;
using System.Configuration.Install;
using System.ServiceProcess;

namespace CSV20.Dienste
{
    [RunInstaller(true)]
    public class DienstInstallation : Installer
    {
        public DienstInstallation()
        {
            ServiceInstaller oInstaller = new ServiceInstaller();
```

```
            ServiceProcessInstaller oProcessInstaller = new ServiceProcessInstaller();

            oInstaller.StartType = ServiceStartMode.Manual; // für automatischen Start:
ServiceStartMode.Automatic
            oInstaller.DisplayName = "CPU- und RAM-Protokollierung";
            oInstaller.ServiceName = "PerformanceLogger";
            oInstaller.Description = "Protokolliert die CPU- und RAM-Nutzung";

            oProcessInstaller.Account = ServiceAccount.LocalSystem;

            Installers.Add(oInstaller);
            Installers.Add(oProcessInstaller);
        }
    }
}
```

Das große Computer ABC

C# lernen

Erweiterte Programmierthemen: ASP.NET WebForms

ASP.NET (Active Server Pages) dient zusammen mit einer .NET-Programmiersprache (also C# oder Visual Basic) als **serverseitige Programmiersprache für Webseiten**. ASP.NET fungiert hierbei als Konkurrenz zu PHP, welches mit über 80 % jedoch einen deutlich höheren Anteil hat. Zumeist wird der Komponente WebForms für die Entwicklung von Webseiten mit ASP.NET verwendet. Als Grundlage für die Entwicklung von Webseiten mit ASP.NET sind **Vorkenntnisse in HTML und CSS** nötigt. JavaScript-Kenntnisse sind ebenfalls von Vorteil. Falls Sie an diesen Programmiersprachen Interesse haben, schauen Sie mal auf der Webseite http://www.homepage-webhilfe.de/ vorbei. Vorteil von ASP.NET ist die große Komplexität der Klassen des .NET-Frameworks. Als Nachteil ist zu nennen, dass ASP.NET (aktuell) nur auf Windows Servern läuft und somit auf Grund der hohen Lizenzgebühren die Hosting-Angebote teuer sind.
In Visual Studio kann ein ASP.NET-Projekt erstellt werden, indem „ASP.NET-WebForms-Anwendung“ gewählt wird. Beim Ausführen des Programms über Visual Studio wird die Webseite mit Hilfe von **IIS Express** gehostet und im Standardbrowser geöffnet. Die Datei Default.aspx stellt die **Indexdatei** dar. Weitere Seiten können ebenfalls mit Visual Studio erstellt werden. Bei der Datei Default.aspx.cs handelt es sich um die sogenannte Code-Behind-Datei. Die Default.aspx.designer.cs-Datei wird automatisch durch die Änderungen an der Default.aspx-Datei generiert. Grundsätzlich sieht die .aspx-Datei nicht viel anders als ein klassisches HTML-Dokument aus. Über die Tags <% und %> können wir eine oder mehrere C#-Anweisung(en) notieren. Dies dient zumeist zur **dynamischen Ausgabe**. Hierfür müssen wir die statische Funktion *Wirte()* der *Response*-Klasse aufrufen.
Alle Elemente im HTML-Code können mit dem Attribut *runat* und dem Wert *server* versehen werden. Dadurch wird in der .aspx.designer.cs eine Variable für das Element erstellt (erfordert jedoch, dass das Attribut id gesetzt ist). Darüber können wir dann im C#-Code auf ein HTML-Element zugreifen, wie wenn es eine Variable wäre. Einige weitere Steuerelemente des ASP.NET-Frameworks können über den Namensraum *asp* bezogen werden. Einige dieser Steuerelemente sind zwar auch über die *input*-Elemente von HTML realisierbar, erlauben jedoch die **Rückmeldung mit Ereignissen** (sogenannte Postbacks). Umgesetzt wird diese Technik über AJAX, bei welcher durch ein JavaScript-Code eine Anfrage an den Server gesendet wird und dadurch die Seite neu geladen wird. Die Eigenschaft *RequestType* der *Request*-Klasse enthält eine Zeichenkette, die die Request-Art (zumeist GET oder POST) enthält. Bei einer GET- oder POST-Anfrage, bei welcher Formular-Daten übertragen werden, können wir über das Array *Form* und den Key (Zeichenkette als Name) den Wert eines Eingabefelds abfragen. Der Wert *IsPostBack* gibt an, ob die Anfrage über ein Postback (also mit einem Ereignis eines Steuerelements) ausgeführt wurde. Ein wichtiges Ereignis, welches nicht registriert werden muss, ist das *Load*-Event. Dieses wird beim Laden der Seite ausgeführt. Auf den Webserver müssen alle .aspx-Dateien sowie der Ordner „bin“ mit der .dll-Datei kopiert werden.

Default.aspx

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
Inherits="CSV20.ASP_NET_WebForms.Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1
/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
    <head runat="server">
        <title><% Response.Write(sTitle); %></title>
    </head>
    <body>
        <!-- Dieser Kommentar wird an den Browser gesendet! -->
        <% /* Dieser Kommentar wird nicht an den Browser gesendet! */  %>
        <h1><% Response.Write(sTitle); %></h1>
        <p><% Response.Write(sBeschreibung); %></p>
        <br />
        <form action="Default.aspx" method="post">
            <table>
                <tr>
                    <th colspan="2" style="text-align: center;">Formular 1 <i>(klassiches
HTML)</i>:</th>
                </tr>
                <tr>
                    <td>Vorname:</td>
                    <!-- durch das setzen des id- und runat-Attributs wird automatisch das
name-Attribut mit dem Wert des id-Attributs gefüllt -->
                    <td><input type="text" size="40" id="inputTextVorname" runat="server"
/></td>
```

```
                </tr>
                <tr>
                    <td>Nachname:</td>
                    <td><input type="text" size="40" id="inputTextNachname" runat="server"
/></td>
                </tr>
                <tr>
                    <td></td>
                    <td><input type="submit" name="inputSubmit1" value="Formular senden"
/><span id="formPost1" runat="server"></span></td>
                </tr>
            </table>
        </form>
        <br />
        <form action="Default.aspx" method="post" runat="server">
            <table>
                <tr>
                    <th colspan="2" style="text-align: center;">Formular 2 <i>(ASP.NET
Controls)</i>:</th>
                </tr>
                <tr>
                    <td>Kalender:</td>
                    <td><asp:Calendar ID="inputKalender" runat="server" Width="250px" /></td>
                </tr>
                <tr>
                    <td></td>
                    <td><input type="submit" name="inputSubmit2" value="Formular senden"
/><span id="formPost2" runat="server"></span></td>
                </tr>
            </table>
        </form>
        <hr style="margin: 5px;" />
        <i>
            <b><% Response.Write(DateTime.Now.ToLongDateString() + " - " +
DateTime.Now.ToLongTimeString()); %></b>
            <br />
            <span>Besucheranzahl: <% Response.Write(BesucherAnzahl()); %></span>
        </i>
    </body>
</html>
```

Default.aspx.cs

```
using System;
using System.Web.UI;

namespace CSV20.ASP_NET_WebForms
{
    public partial class Default : Page
    {
        public const string sTitle = "C# ASP.NET WebForms Anwendung";
        public const string sBeschreibung = "Dies ist unsere eigene Webseite mit C# ASP.NET
WebForms. Der Vorteil von " +
                                            "ASP.NET im Vergleich zu PHP ist, dass wir
relativ einfach eine dynamische " +
                                            "Webseite bauen können. Hierbei kooperiert die
HTML-Seite ohne Probleme mit " +
                                            "der Code-Behind-Datei. Auf dem Webserver muss
die .aspx-Datei und die dazugehörige " +
                                            "von Visual Studio kompilierte DLL abgelegt
werden.<br /><br />" +
                                            "<i>Weitere Informationen entnehmen Sie bitte dem
Kapitel 14 Thema 7 auf der " +
                                            "Webseite von \"Das große Computer ABC\" ...</i>";

        private static int iBesucherZaehler = 0;        // wichtig ist "static" -> hierdurch
ist die Variable von allen Objekt-Erzeugungen
                                                        // der Klasse "Default" und somit von
jedem Webseiten-Aufruf zugreifbar

        protected void Page_Load(object sender, EventArgs e)
        {
            // Formular-Info-Text zurücksetzen
            formPost1.InnerHtml = formPost2.InnerHtml = "";

            // prüfen ob Seite auf Grund eines "Post-Back" geladen wurde (z. B. beim Klicken
auf den Kalender)
            //  --> in diesem Fall Besucherzähler nicht hochzählen
            if (!Page.IsPostBack)
                iBesucherZaehler++;
```

```
            if (Request.RequestType == "POST")
            {
                if (Request.Form["inputSubmit1"] != null)
                {
                    if (Request.Form["inputTextVorname"] != null &&
Request.Form["inputTextNachname"] != null)
                    {
                        inputTextVorname.Value = Request.Form["inputTextVorname"];
                        inputTextNachname.Value = Request.Form["inputTextNachname"];
                        formPost1.InnerHtml = "<br />Formular erfolgreich abgesendet!";
                    }
                    else
                        formPost1.InnerHtml = "<br />Formular fehlerhaft versendet!";
                }
                else if (Request.Form["inputSubmit2"] != null)
                {
                    // Abfrage wie if (Request.Form["inputKalender"] != null) trifft nie zu,
da der Kalender immer durch die
                    // Post-Backs übertragen werden und nicht über das Formular
                    formPost2.InnerHtml = "<br />Formular erfolgreich abgesendet!";
                    formPost2.InnerHtml += "<br />Ausgewähltes Datum: " +
inputKalender.SelectedDate.ToShortDateString();
                }
            }
        }

        public string BesucherAnzahl()
        {
            return iBesucherZaehler.ToString();
        }
    }
}
```

Default.aspx.designer.cs

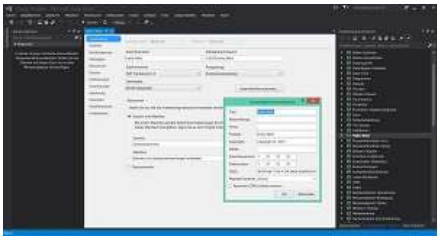
```
//------------------------------------------------------------------------------
// <automatisch generiert>
//     Der Code wurde von einem Tool generiert.
//
//     Änderungen an der Datei führen möglicherweise zu falschem Verhalten, und sie gehen
verloren, wenn
//     der Code erneut generiert wird.
// </automatisch generiert>
//------------------------------------------------------------------------------

namespace CSV20.ASP_NET_WebForms {


    public partial class Default {

        /// <summary>
        /// inputTextVorname-Steuerelement
        /// </summary>
        /// <remarks>
        /// Automatisch generiertes Feld
        /// Um dies zu ändern, verschieben Sie die Felddeklaration aus der Designerdatei in
eine Code-Behind-Datei.
        /// </remarks>
        protected global::System.Web.UI.HtmlControls.HtmlInputText inputTextVorname;

        /// <summary>
        /// inputTextNachname-Steuerelement
        /// </summary>
        /// <remarks>
        /// Automatisch generiertes Feld
        /// Um dies zu ändern, verschieben Sie die Felddeklaration aus der Designerdatei in
eine Code-Behind-Datei.
        /// </remarks>
        protected global::System.Web.UI.HtmlControls.HtmlInputText inputTextNachname;

        /// <summary>
        /// formPost1-Steuerelement
        /// </summary>
        /// <remarks>
        /// Automatisch generiertes Feld
        /// Um dies zu ändern, verschieben Sie die Felddeklaration aus der Designerdatei in
eine Code-Behind-Datei.
        /// </remarks>
        protected global::System.Web.UI.HtmlControls.HtmlGenericControl formPost1;

```

```
42          /// <summary>
43          /// inputKalender-Steuerelement
44          /// </summary>
45          /// <remarks>
46          /// Automatisch generiertes Feld
47          /// Um dies zu ändern, verschieben Sie die Felddeklaration aus der Designerdatei in
    eine Code-Behind-Datei.
48          /// </remarks>
49          protected global::System.Web.UI.WebControls.Calendar inputKalender;
50
51          /// <summary>
52          /// formPost2-Steuerelement
53          /// </summary>
54          /// <remarks>
55          /// Automatisch generiertes Feld
56          /// Um dies zu ändern, verschieben Sie die Felddeklaration aus der Designerdatei in
    eine Code-Behind-Datei.
57          /// </remarks>
58          protected global::System.Web.UI.HtmlControls.HtmlGenericControl formPost2;
59      }
60  }
```

Das große Computer ABC

C# lernen

Tipps und Tricks: Projektmanagement



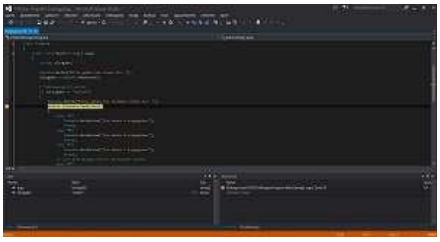
Visual Studio hilft uns natürlich auch bei der **Verwaltung mehrerer Projekte**. Hierbei gibt es die Unterscheidung zwischen Projekt und Projektmappe. Eine **Projektmappe** umfasst mehrere **Projekte**. In den Eigenschaften der Projektmappe können wir festlegen welches Projekt das Startprojekt ist. Das Startprojekt ist das Projekt, welches beim Ausführen über den Start- oder Debug-Button gestartet wird. Hierbei können wir mehrere Projekte als Startprojekt, die aktuelle Auswahl oder ein einzelnes Projekt wählen. Zum einen dient die Projektmappe zur **Gruppierung** einer großen Anwendung (mit mehreren exe- und dll-Dateien) oder zur Gruppierung mehrerer zusammengehöriger Programme (z. B. unsere Beispielprogramme).

Ein einzelnes Projekt hat ebenfalls Eigenschaften. Dort können **Assembly-Name** (Name der exe- oder dll-Datei), **Standardnamensraum** und **Zielframework** eingestellt werden. Bei den **Assemblyinformationen** handelt es sich um Informationen, welche in der resultierenden exe- oder dll-Datei hinterlegt werden. Es handelt sich also um den Titel und die Beschreibung, sowie Copyright und Version. Des Weiteren können wir in den Eigenschaften des Projektes das **Icon** für die exe-Datei festlegen.

Das große Computer ABC

C# lernen

Tipps und Tricks: Debugging



Debugging dient zur **Fehlersuche** und **Fehlerbehebung**. Dieser sogenannte Debugger ist ein wichtiger Bestandteil einer Entwicklungsumgebung.
Durch das Klicken auf die Leiste auf der linken Seite wird ein **Breakpoint** an die angeklickte Zeile gesetzt. Hierdurch unterbricht der Debugger den Programmablauf, sobald das Programm an diese Stelle im Programmcode gelangt. Bitte denken Sie daran, dass das Setzen eines Breakpoints während dem Programmablauf nicht möglich ist, sondern lediglich vor der Kompilierungszeit oder während der Pausierung des Programms. Wollen wir über die aktuelle Anweisung springen, so können wir über F10 springen. Mit F11 springen wir in die Anweisung (dies ist jedoch nur bei Funktionsaufrufen relevant).
Ist die Anwendung pausiert, so können wir **Werte von Variablen** durch das Darüberfahren über die Variablennamen anzeigen lassen. Objekte lassen sich noch mit dem Pluszeichen weiter aufklappen, um Eigenschaften des Objektes anzuzeigen. Des Weiteren ist es möglich, Variablen im Fenster **Überwachen** anzudocken. Hierbei hat man eine bessere Übersicht über die zu überwachenden Variablen.
Wie Sie vielleicht schon einmal gesehen haben, wird ein Fenster in Visual Studio angezeigt, wenn eine **Exception** ausgelöst wird. Auch hier hat der Debugger seine Finger im Spiel. Würde dieser das Laufzeitverhalten der Anwendung nicht überwachen, so würde ein .NET-Framework Fehlermeldung ausgegeben werden oder das Programm würde einfach „abstürzen“.
Doch wie hilft uns Debugging bei der Fehlerbehebung? Fehler in einem eigenen Programmcode entstehen durch Logikfehler bzw. Denkfehler oder Wissenslücken. Logikfehler können durch diese Features ermittelt werden. Wenn wir besser verstehen, was in unserem Programm abläuft (z. B. wenn eine Exception auftritt), so finden wir auch noch mögliche Ursachen für Fehler und Probleme. Des Weiteren können wir mit Hilfe der Einzelschritt-Funktion den Programmablauf durchgehen und in dieser Zeit Variablen, sowie deren Wertänderungen, ständig beobachten. Dadurch wird die Suche nach dem Fehler um ein vielfaches vereinfacht.

Das große Computer ABC

C# lernen

Tipps und Tricks: using-Block

Der *using*-Block (oder auch *using*-Anweisung genannt) stellt die korrekte Verwendung von Objekten, welche von dem Interface *IDisposable* abgeleitet sind sicher. Grundsätzlich sollte bei allen *IDisposable*-Objekten am Ende der Verwendung die Funktion *Dispose()* aufgerufen werden. Dadurch verlieren die Variablen des Objektes Ihre **Gültigkeit**, indem diese auf *null* gesetzt werden. Jeder Klasse muss jedoch seine eigene konkrete Dispose-Funktionalität implementieren. Doch warum ist diese Funktionalität zu empfehlen? Durch das Löschen der Referenz der internen Variablen werden die Objekte vom Garbage Collector gelöscht. Zumeist kommen solche Objekte bei **Streams** vor. Bisher haben wir bei Streams immer gesagt, dass man die *Close()* aufrufen soll. Haben wir also bisher etwas falsch gemacht? Nein, denn die *Close()*-Funktion macht zumeist nichts anderes als die Funktion *Dispose()* aufzurufen.
Nun zum Syntax: Als erstes notieren wir das Schlüsselwort *using*. Welches jedoch nicht mit dem *using*-Schlüsselwort zur Einbindung von Namensräumen verwechselt werden darf. Es folgt ein rundes Klammernpaar, in welchem wir ein **Objekt erzeugen** und dieses einer Variablen zuweisen. Auf diese Variable kann nun ausschließlich innerhalb des **dazugehörigen Blocks** zugegriffen werden. Am Ende des Blocks wird nun automatisch die *Dispose()*-Funktion aufgerufen. Das Beispiel zeigt einen Dateizugriff. Der auskommentierte Code stellt hierbei den Code dar, welcher dem Code des *using*-Blocks entspricht.

Program.cs

```
1  using System;
2  using System.IO;
3
4  namespace CSV20.using_Block
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             using (StreamWriter oStream = new StreamWriter("test.txt"))
11             {
12                 oStream.WriteLine("Hallo Welt!");
13             }
14
15             //StreamWriter oStream = new StreamWriter("test.txt");
16             //try
17             //{
18             //    oStream.WriteLine("Hallo Welt!");
19             //}
20             //finally
21             //{
22             //    if (oStream != null)
23             //        oStream.Dispose();
24             //}
25
26             Console.Write("Programmende. Bitte Taste drücken ... ");
27             Console.ReadKey();
28         }
29     }
30 }
```

Das große Computer ABC

C# lernen

Tipps und Tricks: Töne ausgeben

Über die statische Funktion *Beep()* der *Console*-Klasse können wir Töne ausgeben. Hierzu übergeben wir der Funktion zwei Parameter. Der erste Parameter stellt die **Frequenz in Hertz** dar. Der zweite Parameter gibt die **Länge des Tons in Millisekunden** an. Mit Hilfe dieser Funktion und einer Frequenztabelle für die verschiedenen Tonhöhen in der Musik können wir nun verschiedene Lieder abspielen. Das Beispiel zeigt wie wir das Lied „Alle meine Entchen“ mit Hilfe der *Beep()*-Funktion spielen können. Bitte bedenken Sie, dass die *Beep()*-Funktion synchron arbeitet und somit den Programmablauf für die angegebene Dauer in Millisekunden blockiert.

Program.cs

```
using System;

namespace CSV20.Ton_Ausgabe
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Al");
            Console.Beep(262,  500);    // C    Al-
            Console.Write("le ");
            Console.Beep(294,  500);    // D    le
            Console.Write("mei");
            Console.Beep(330,  500);    // E    mei-
            Console.Write("ne ");
            Console.Beep(349,  500);    // F    ne
            Console.Write("Ent");
            Console.Beep(392, 1000);    // G    Ent-
            Console.Write("chen ");
            Console.Beep(392, 1000);    // G    chen
            Console.Write("schwim");
            Console.Beep(440,  500);    // A    schwim-
            Console.Write("men ");
            Console.Beep(440,  500);    // A    men
            Console.Write("auf ");
            Console.Beep(440,  500);    // A    auf
            Console.Write("dem ");
            Console.Beep(440,  500);    // A    dem
            Console.Write("See, ");
            Console.Beep(392, 1500);    // G    See,
            Console.Write("schwim");
            Console.Beep(440,  500);    // A    schwim-
            Console.Write("men ");
            Console.Beep(440,  500);    // A    men
            Console.Write("auf ");
            Console.Beep(440,  500);    // A    auf
            Console.Write("dem ");
            Console.Beep(440,  500);    // A    dem
            Console.WriteLine("See,");
            Console.Beep(392, 1500);    // G    See,

            Console.Write("Köpf");
            Console.Beep(349,  500);    // F    Köpf-
            Console.Write("chen ");
            Console.Beep(349,  500);    // F    chen
            Console.Write("in ");
            Console.Beep(349,  500);    // F    in
            Console.Write("das ");
            Console.Beep(349,  500);    // F    das
            Console.Write("Was");
            Console.Beep(330, 1000);    // E    Was-
            Console.Write("ser ");
            Console.Beep(330, 1000);    // E    ser,
            Console.Write("Schwänz");
            Console.Beep(392,  500);    // G    Schwänz-
            Console.Write("chen ");
            Console.Beep(392,  500);    // G    chen
            Console.Write("in ");
```

```
59              Console.Beep(392,  500);    // G    in
60              Console.Write("die ");
61              Console.Beep(392,  500);    // G    der
62              Console.WriteLine("Höh'");
63              Console.Beep(262, 1500);    // C    Höh'
64
65              Console.ReadKey();
66          }
67      }
68  }
```

Das große Computer ABC

C# lernen

Tipps und Tricks: Namensregeln und Kommentierung

Namensregeln bzw. korrekterweise als **Namenskonventionen** bezeichnet sind Empfehlungen, welche einen im Allgemeinen **besser lesbaren und strukturierten Programmcode** mit sich bringen sollen. Diese „Regeln“ müssen keinesfalls eingehalten werden und können sich von Programmiersprache zu Programmiersprache unterscheiden. Einige Programmierer halten sich daran, andere wiederum nicht.
Wie wir bereits am Anfang dieses Kurses angesprochen haben, sollten alle **Variablennamen mit einem Kürzel** versehen werden, welche den Datentyp widerspiegeln (z. B. eine 32bit-Ganzzahl namens „Test“ sollte *iTest* genannt werden). Auch die **Einrückung** eines Programmcodes sollte beachtet werden (zumeist ein Tab bzw. 4 Leerzeichen für eine Unterordnung). Klassennamen, Enumerationstypen und -werte, Funktionen, Eigenschaften und Namensräume sollten **mit einem Großbuchstaben beginnen**. Werden **mehrere Wörter** verwendet, so werden die **Anfangsbuchstaben** jeden Wortes ebenfalls wieder groß geschrieben (z. B. *AddiereGanzzahlen()*). Ob Sie ihren Programmcode in **Englisch oder Deutsch** programmieren, ist jedem selbst überlassen. Unsere Beispiele haben wir bewusst auf Deutsch programmiert. Selbstverständlich sollten Sie auch **in jeder Zeile immer nur eine Anweisung oder Deklaration** notieren. Zwischen Schlüsselwörtern und Klammern oder Kommas sollte des Weiteren unbedingt jeweils ein **Leerzeichen** notiert werden. Zudem empfiehlt es sich *for*-Schleifen bei Arrays durch *foreach*-Schleifen zu ersetzen, falls dies möglich ist.
Wie wir in C# Kommentare notieren haben wir ja bereits im Grundlagen-Kapitel gelernt. Eine gute Kommentierung ist sehr wichtig, sodass wir auch noch nach ein paar Jahren unseren Programmcode nachvollziehen können. Doch zusätzlich zu diesen „normalen“ Kommentaren ermöglicht es Visual Studio eine **Dokumentation** für Klassen, Funktionen und Eigenschaften anzulegen. Hierfür notieren wir drei Schrägstriche. Dadurch erstellt Visual Studio automatisch ein Dokumentations-Gerüst, welches unten beispielhaft aufgeführt ist. Die Dokumentation wird in der IntelliSense-Autovervollständigung angezeigt.

Program.cs

```
1  using System;
2
3  namespace CSV20.Namensregeln_Kommentierung
4  {
5      /// <summary>
6      /// Klasse des Hauptprogramms, welches die Main()-Funktion enthält.
7      /// </summary>
8      class Program
9      {
10         /// <summary>
11         /// Der Haupteinstiegspunkt der Konsolen-Anwendung.
12         /// </summary>
13         /// <param name="args">Die Parameter, welche dem Programm beim Start übergeben wurden.
   </param>
14         static void Main(string[] args)
15         {
16             Console.WriteLine(Addieren(6195, 153));
17
18             Console.ReadKey();
19         }
20
21         /// <summary>
22         /// Addiert 2 ganzzahlige 32bit Werte (vorzeichenbehaftet) miteinander.
23         /// </summary>
24         /// <param name="iZahl1">Die 1. Zahl welche für die Rechnung benötigt wird.</param>
25         /// <param name="iZahl2">Die 2. Zahl welche für die Rechnung benötigt wird.</param>
26         /// <returns>Das Ergebnis der Addition beider übergebener Zahlen.</returns>
27         static int Addieren(int iZahl1, int iZahl2)
28         {
29             return iZahl1 + iZahl2;
30         }
31     }
32 }
```

Das große Computer ABC

C# lernen

Tipps und Tricks: Abschließende Tipps

Wir haben in diesem C#-Kurs versucht, Ihnen die Programmiersprache C# und deren Komponenten sowie das .NET-Framework näherzubringen. Natürlich ist uns jedoch auch bewusst, dass wir keinesfalls den kompletten **Wortschatz** aller .NET-Libraries behandelt haben. Dies ist auch keinesfalls möglich und würde diesen Kurs sprengen. Also was tun, wenn wir die gewünschte Funktion nicht finden oder bei einem Problem nicht weiterkommen? Hier spielt in der heutigen Zeit natürlich das **Internet** die größte Rolle. Suchen wir etwas, so bemühen wir eine Suchmaschine unserer Wahl und kommen oft schnell zu einer Lösung. Hierbei empfiehlt es sich auch die Suche in englischer Sprache durchzuführen. Wollen wir uns lediglich die **Dokumentation** von Funktionen, Eigenschaften und Ereignissen einer bestimmten Klasse anschauen oder einfach durch die verschiedenen Namensräume und Klassen „blättern“, so ist die **C#-Referenz von Microsoft** zu empfehlen, welche unter https://msdn.microsoft.com/de-de/library/618ayhy6.aspx zu finden ist. Die Referenz ist sehr komplex und deckt alle .NET-Libraries ab. Zudem enthält die Microsoft-Webseite auch einige **Beispiele** zur Verwendung von verschiedenen Funktionen, Eigenschaften, Ereignissen und natürlich den Klassen selbst. Mit diesen Informationen sollten Sie also bestens vorbereitet sein große Projekte zu starten.

Ich wünsche Ihnen weiterhin viel Spaß, Erfolg und natürlich Freude mit C#.

Mit freundlichen Grüßen
Benjamin Jung
(Buchautor)

Impressum

Sowohl der Webseitenteil „C# lernen“ unter www.das-grosse-computer-abc.de/CSharp/ also auch dieses E-Book unterliegen dem Deutschen Urheberrecht.

Für den Inhalt dieses E-Books ist verantwortlich:

Name: Benjamin Jung
Anschrift: Krummstraße 9 /3
73054 Eislingen
Telefon: + 49 (0) 7161 87655
Telefax: + 49 (0) 7161 87385
Webseite: www.das-grosse-computer-abc.de/CSharp/
E-Mail: kontakt@das-grosse-computer-abc.de

Support vor Ort, telefonisch oder per Fax ist nicht möglich!
Keine der oberen Adressen darf für den Support bei C#-Problemen genutzt werden.
Nur Support zur Webseite, zum E-Book o. Ä. ist per E-Mail gestattet!

Haftungsausschluss

Wir als Team von „Das große Computer ABC“ und ich, Benjamin Jung als Administrator und Autor, übernehmen keine Haftung für die Inhalte. Alle Inhalte wurden mit größter Sorgfalt und bestem Wissen erstellt, jedoch übernehmen wir keine Haftung für Richtigkeit, Vollständigkeit und Aktualität.

Unser E-Book enthält außerdem Links zu Webseiten von Drittanbietern. Für die Inhalte dieser Webseiten können wir keine Haftung übernehmen, dafür sind ausschließlich die Anbieter der Ziel-Webseite verantwortlich.

Urheberrecht