walter DOBERENZ
thomas GEWINNUS

VISUAL C# 2012

GRUNDLAGEN UND PROFIWISSEN

// C#-Grundlagen, Techniken, OOP
// GUI-Programmierung mit Windows Forms und WPF
// Entwickeln von Windows Store Apps

E-Book

EXTRA: Kostenloses E-Book inkl. 1000 Seiten Bonuskapitel

HANSER

Doberenz/Gewinnus

Visual C# 2012
Grundlagen und Profiwissen

Walter Doberenz
Thomas Gewinnus

Visual C# 2012

Grundlagen und Profiwissen

HANSER

Die Autoren:

Professor Dr.-Ing. habil. Walter Doberenz, Wintersdorf

Dipl.-Ing. Thomas Gewinnus, Frankfurt/Oder

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

Lektorat: Sieglinde Schärl
Herstellung: Thomas Gerhardy
Satz: Ingenieurbüro Gewinnus
Sprachlektorat: Walter Doberenz
Umschlagdesign: Marc Müller-Bremer, *www.rebranding.de,* München
Umschlagrealisation: Stephan Rönigk
Druck und Bindung: Kösel, Krugzell
Ausstattung patentrechtlich geschützt. Kösel FD 351, Patent-Nr. 0748702
Printed in Germany

ISBN 978-3-446-43439-4

E-Book-ISBN 978-3-446-43521-6

Inhaltsverzeichnis

Teil II: Technologien

Teil III: WPF-Anwendungen

Anhang

Bonuskapitel im E-Book

Teil V: Weitere Technologien

– Diese Kapitel finden Sie nur im E-Book –

Teil VI: Windows Forms

Teil VII: ASP.NET

Teil VIII: Silverlight

Vorwort

C# ist eine noch junge Sprache, sie bietet Ihnen die Möglichkeiten und Flexibilität von C++ und erlaubt trotzdem eine schnelle und unkomplizierte Programmierpraxis wie Visual Basic. C# ist (fast) genauso mächtig wie C++, wurde aber komplett neu auf objektorientierter Basis geschrieben.

Damit ist C# das ideale Werkzeug zum Programmieren beliebiger Komponenten für das Microsoft .NET Framework, beginnend bei Windows Forms- über WPF-, ASP.NET- , WinRT- und Silverlight-Anwendungen bis hin zu systemnahen Applikationen.

Das vorliegende Buch ist ein faires Angebot für künftige wie auch für fortgeschrittene C#-Programmierer. Seine Philosophie knüpft an die vielen anderen Titel an, die wir in den vergangenen zwanzig Jahren zu verschiedenen Programmiersprachen geschrieben haben:

- Programmieren lernt man nicht durch lineares Durcharbeiten eines Lehrbuchs, sondern nur durch unermüdliches Ausprobieren von Beispielen, verbunden mit ständigem Nachschlagen in der Referenz.
- Der Umfang einer modernen Sprache wie C# in Verbindung mit Visual Studio ist so gewaltig, dass ein seriöses Programmierbuch das Prinzip der Vollständigkeit aufgeben muss und nach dem Prinzip "so viel wie nötig" sich lediglich eine "Initialisierungsfunktion" auf die Fahnen schreiben kann.

Gegenüber anderen Büchern zur gleichen oder ähnlichen Thematik nimmt unser Titel für sich in Anspruch, gleichzeitig Lehr- und Übungsbuch zu sein.

Zum Buchinhalt

Wie Sie bereits dem Buchtitel entnehmen können, wagt das vorliegende Werk den Spagat zwischen einem Grundlagen- und einem Profibuch. Sinn eines solchen Buches kann es nicht sein, eine umfassende Schritt-für-Schritt-Einführung in Visual C# 2012 zu liefern oder all die Informationen noch einmal zur Verfügung zu stellen, die Sie in der Produktdokumentation (MSDN) ohnehin schon finden und von denen Sie in der Regel nur ein Mausklick oder die F1-Taste trennt.

- Für den *Einsteiger* wollen wir den einzig vernünftigen und gangbaren Weg beschreiten, nämlich nach dem Prinzip "so viel wie nötig" eine schmale Schneise durch den Urwald der .NET-Programmierung mit Visual C# 2012 schlagen, bis er eine Lichtung erreicht hat, die ihm erste Erfolgserlebnisse vermittelt.

- Für den *Profi* wollen wir in diesem Buch eine Vielzahl von Informationen und Know-how bereitstellen, wonach er bisher in den mitgelieferten Dokumentationen, im Internet bzw. in anderen Büchern vergeblich gesucht hat.

Da mit Visual C# 2012 zahlreiche neue Features (vor allem die WinRT-Programmierung) neu hinzugekommen sind und wir auch viele Leserwünsche zusätzlich eingearbeitet haben, mussten einige Kapitel wesentlich erweitert bzw. neu hinzugefügt werden.

Ein Vergleich der Inhaltsverzeichnisse zeigt, dass aus den ursprünglichen 40 Kapiteln nunmehr 43 Kapitel geworden sind, die inzwischen 2200 Seiten umfassen.

Die Kapitel des Buchs haben wir in acht Themenkomplexen gruppiert:

- Grundlagen der Programmierung mit Visual C# 2012
- Technologien der Programmentwicklung
- WPF-Anwendungen
- Windows Store Apps
- Weitere Technologien
- Windows Forms
- ASP.NET
- Silverlight

Die Kapitel innerhalb eines Teils bilden einerseits eine logische Aufeinanderfolge, können andererseits aber auch quergelesen werden. Im Praxisteil eines jeden Kapitels werden anhand realer Problemstellungen die behandelten Programmiertechniken im Zusammenhang demonstriert.

Im gedruckten Teil dieses Buchs finden Sie die ersten vier Themenkomplexe, denn bereits hier sind wir an die Grenze des drucktechnisch Machbaren gestoßen. Die übrigen vier Themenkomplexe mussten wir in ein E-Book auslagern, welches Sie sich kostenlos aus dem Internet herunterladen können.

Zu den Neuigkeiten in Visual Studio 2012

Mit dem Erscheinen von Windows 8 bietet Microsoft erstmals ein Tablet-taugliches Betriebssystem an, für das Sie als C#-Entwickler eigene Anwendungen (neudeutsch Apps) entwickeln können. Während es für iPad und Android-Geräte mittlerweile viele tausende Apps gibt, herrscht bei Windows 8 noch ein riesengroßer Nachholbedarf.

Dieses Buch soll Ihnen helfen, die dafür nötigen Grundkenntnisse zu gewinnen und eigene Apps zu entwickeln. Dabei helfen Ihnen neben den fünf WinRT-Kapiteln auch die WPF-Kapitel über die entsprechenden Basistechnologien (XAML/Datenbindung).

Leider ist nicht alles Gold was glänzt, und so hat Microsoft mit der Visual Studio 2012, dem .NET-Framework 4.5 und Windows 8 zwar vieles anders, aber nicht alles besser gemacht. Wer seinen Blick durch diverse Foren streifen lässt, der wird schnell den Unmut über viele Entscheidungen bei den Entwicklern spüren:

- Das freudlose "Gruftilayout" der neuen Visual Studio-IDE und die Beschriftung des Hauptmenüs in Großbuchstaben sind nicht jedermanns Geschmack.
- Das Setup-Template wurde aus Visual Studio entfernt, Sie dürfen *InstallShield Limited Edition* verwenden (Download per Website).
- Das .NET-Framework 4.5 wird nicht mehr unter Windows XP unterstützt. Der XP-Marktanteil lag im September 2012 noch bei 42%, genauso hoch wie bei Windows 7. Auf diese Kundengruppe werden viele Entwickler nicht verzichten wollen. Damit aber bleibt nur das "alte" Framework 4.0 als Zielframework für neue Projekte.
- Windows 8-Projekte (WinRT) können nur unter Windows 8 erstellt werden und im Wesentlichen auch nur per Microsoft Store vertrieben werden.

Wir hoffen, dass der Leser dieses Buchs obige kritische Worte höher zu schätzen weiß als den euphorischen Lobgesang manch anderer Autoren auf jedwede Art von "Highlights".

Weitere Bücher von uns

Auf drei weitere von uns verfasste Buchtitel, die sich ebenfalls auf Visual Studio 2012 beziehen, wollen wir Sie hier noch hinweisen:

- Eine ideale Ergänzung zum vorliegenden Buch ist unser "Visual C# 2012 – Kochbuch". Mit mehr als 500 How-to-Problemlösungen zu allen hier behandelten Grundlagenthemen sind Sie bestens für die Anforderungen der Praxis gewappnet und können weitere Lücken schließen.
- Das Pendant zum vorliegenden Buch ist unser im gleichen Verlag erschienener Buchtitel "Visual Basic 2012 – Grundlagen und Profiwissen". Da es das gleiche Inhaltsverzeichnis hat (inklusive Beispielcode), lassen sich ideale Vergleiche zwischen beiden Sprachen anstellen. Eine solche "Übersetzungshilfe" scheint besonders wichtig zu sein, weil einerseits viele altgediente Visual Basic-Programmierer zu C# wechseln werden und man andererseits in einem .NET-Entwicklerteam durchaus in mehreren .NET-Sprachen zusammenarbeitet.
- Der Datenbank- und Web-Programmierung widmet sich ausführlich unser bei Microsoft Press erschienener Spezialtitel "Datenbankprogrammierung mit Visual C# 2012".

Zu den Codebeispielen

Alle Beispieldaten dieses Buchs können Sie sich unter der Adresse

LINK: `http://www.doko-buch.de`

herunterladen.

Beim Nachvollziehen der Buchbeispiele beachten Sie bitte Folgendes:

- Kopieren Sie die Buchbeispiele auf die Festplatte. Wenn Sie auf die Projektmappendatei (**.sln*) klicken, wird Visual Studio in der Regel automatisch geöffnet und das jeweilige Beispiel wird in die Entwicklungsumgebung geladen, wo Sie es z.B. mittels F5-Taste kompilieren und starten können.

- Einige wenige Datenbankprojekte verwenden absolute Pfadnamen, die Sie vor dem Kompilieren des Beispiels erst noch anpassen müssen.
- Für einige Beispiele sind ein installierter Microsoft SQL Server Express LocalDB sowie der Microsoft Internet Information Server (ASP.NET) erforderlich.
- Um mit den WinRT-Projekten arbeiten zu können, müssen Sie Visual Studio 2012 unter Windows 8 ausführen.
- Beachten Sie die zu einigen Beispielen beigefügten *Liesmich.txt*-Dateien, die Sie auf besondere Probleme hinweisen.

Nobody is perfect

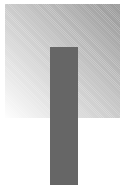
Sie werden – trotz der rund 2200 Seiten – in diesem Buch nicht alles finden, was Visual C# 2012 bzw. das .NET Framework 4.5 zu bieten haben. Manches ist sicher in einem anderen Spezialtitel noch besser oder ausführlicher beschrieben. Aber Sie halten mit unserem Buch einen optimalen und überschaubaren Breitband-Mix in den Händen, der sowohl vertikal vom Einsteiger bis zum Profi als auch horizontal von den einfachen Sprachelementen bis hin zu komplexen Anwendungen jedem etwas bietet, ohne dabei den Blick auf das Wesentliche im .NET-Dschungel zu verlieren.

Wenn Sie Vorschläge oder Fragen zum Buch haben, können Sie uns gern über unsere Website kontaktieren:

LINK: `http://www.doko-buch.de`

Wir hoffen, dass wir Ihnen mit diesem Buch einen nützlichen Begleiter bei der .NET-Programmierung zur Seite gestellt haben, der es verdient, seinen Platz nicht im Regal, sondern griffbereit neben dem Computer einzunehmen.

Walter Doberenz und Thomas Gewinnus *Wintersdorf/Frankfurt/O., im Oktober 2012*



Teil I: Grundlagen

- **Einstieg in Visual Studio 2012**
- **Grundlagen der Sprache C#**
- **Objektorientiertes Programmieren**
- **Arrays, Strings und Funktionen**
- **Weitere wichtige Sprachfeatures**
- **Einführung in LINQ**

Kapitel 1

Einstieg in Visual Studio 2012

Dieses Kapitel bietet Ihnen einen effektiven Schnelleinstieg in die Arbeit mit Visual Studio 2012. Gleich nachdem Sie die Hürden der Installation gemeistert haben, erstellen Sie Ihre ersten .NET-Anwendungen, werden dabei en passant mit den grundlegenden Features der Entwicklungsumgebung vertraut gemacht und nach dem Prinzip "soviel wie nötig" in die .NET-Philosophie eingeweiht. Nach der Lektüre dieses Kapitels und dem Nachvollziehen der abschließenden Praxisbeispiele sollte der Einsteiger über eine brauchbare Ausgangsbasis verfügen, um den sich vor ihm gewaltig auftürmenden Berg von Spezialkapiteln in Angriff zu nehmen.

Der erfahrene Visual Studio-Anwender erhält im Abschnitt 1.6 einen Überblick über die Neuerungen der Version 2012 gegenüber der Vorgängerversion Visual Studio 2010.

1.1 Die Installation von Visual Studio 2012

Ohne eine angemessen ausgestattete "Werkstatt" ist die Lektüre dieses Buchs nutzlos. Programmieren lernt man nur durch Beispiele, die man unmittelbar selbst am Rechner ausprobiert!

HINWEIS: Voraussetzung für ein erfolgreiches Studium dieses Buchs ist ein Rechner mit einer lauffähigen Installation von Visual Studio 2012!

1.1.1 Überblick über die Produktpalette

Alle im Handel angebotenen Visual-Studio-Pakete basieren auf dem .NET-Framework 4.5. Für welches der im Folgenden aufgeführten Produkte man sich entscheidet, hängt von den eigenen Anforderungen und Wünschen ab und ist nicht zuletzt auch eine Frage des Geldbeutels.

Visual Studio 2012 Express

Hier handelt es sich um abgespeckte, dafür aber kostenlose Versionen von Microsofts Entwicklungsumgebung. Wenn Sie als Hobby-Programmierer auf Features wie Berichte, Remote Debugging, ClickOnce etc. verzichten können, sind diese Minimalpakete in vielen Fällen ausreichend um eigene Anwendungen oder Webseiten zu erstellen. Folgende Editionen sind erhältlich:

- **Visual Studio Express 2012 für Windows 8**
  Mit dieser Edition können Sie WinRT-Applikationen für Windows 8 entwickeln. Enthalten sind neben C# auch Vorlagen für Visual Basic, JavaScript und C++, sowie das Windows 8 SDK und Blend für Visual Studio. Die von Ihnen geschriebenen Anwendungen können Sie anschließend im Windows Store Ihren potentiellen Kunden anbieten.
- **Visual Studio Express 2012 für Windows Desktop**
  Hiermit ist die einfache Entwicklung von Desktop- und Konsolenanwendungen für alle von Visual Studio 2012 unterstützten Windows-Versionen möglich. Neben C# stehen Ihnen dazu auch noch Visual Basic und C++ zur Verfügung. Enthalten sind auch einige Sprachtools (z.B. integrierter Unit-Test).
- **Visual Studio Express 2012 für Web**
  Bereits mit dieser Minimalausstattung ist die Entwicklung ansprechender interaktiver Webanwendungen möglich. Die Verteilung kann über den Webserver oder die Cloud unter Windows Azure erfolgen.
- **Visual Studio 2012 Express für Windows Phone**
  Unter Visual Studio 2012 wurde die Unterstützung für App-Entwickler nochmals stark erweitert. Eine entsprechende Visual Studio 2012 Express Edition wird allerdings erst mit der nächsten Version von Windows Phone verfügbar sein.
- **Visual Studio Team Foundation Server Express 2012**
  Teams mit bis zu fünf Entwicklern erhalten mit dieser Edition die Tools zur Quellcodeverwaltung, Buildautomatisierung undArbeitsaufgabennachverfolgung.

Visual Studio 2012 Professional

Wie es der Name bereits suggeriert, handelt es sich bei diesem Standard-Paket bereits um ein professionelles Werkzeug, denn es beinhaltet alle erforderlichen Kernfunktionen für die Entwicklung von Anwendungen für Windows, Office, das Web, die Cloud, Silverlight, SharePoint und Multi-Core-Szenarien.

Auch Visual Studio LightSwitch, die Entwicklungsumgebung für das Rapid Application Development (RAD), ist jetzt Bestandteil von Visual Studio Professional, Premium und Ultimate. Ähnliches gilt für die Team-Unterstützung und das Application Lifecycle Management (ALM), eine Sammlung von Tools und Prozessen zur Überwachung und Kontrolle des gesamten Entwicklungszyklus einer Applikation.

Sowohl ernstzunehmende Hobbyprogrammierer als auch professionelle Entwickler, die allein oder im kleinen Team an der Erstellung komplexer, mehrschichtiger Anwendungen arbeiten, sind mit dieser Edition gut beraten.

HINWEIS: Der Inhalt dieses Buches bezieht sich schwerpunktmäßig auf die Möglichkeiten der **Professional Edition**!

Visual Studio 2012 Premium

Bei diesem Paket handelt es sich um eine Vollausstattung für Softwareentwickler und -tester, um im Team Anwendungen auf Enterprise-Niveau zu entwickeln. Enthalten sind alle Funktionen der Professional-Version sowie weitere Funktionen, die komplexe Datenbankentwicklung und eine durchgängige Qualitätssicherung ermöglichen sollen. So finden sich

- Funktionen zur Verbesserung der Codequalität durch Codeüberprüfung mittels Peer-Workflow,
- bessere Entwicklungstools für den Entwurf von Multithreading-Anwendungen,
- Möglichkeiten zur Automatisieren von Benutzeroberflächentests
- Funktionen zum Suchen und Verwalten von doppeltem Code in der CodeBase zur Verbesserung der Architektur

Visual Studio 2012 Ultimate

Aufbauend auf dem Funktionsumfang von Visual Studio 2012 Premium finden sich zusätzlich folgende Funktionen:

- Zuverlässiges Erfassen und Reproduzieren von Fehlern, die während manueller und explorativer Tests gefunden werden, um nicht reproduzierbare Fehler zu vermeiden
- Verstehen der Abhängigkeiten und Beziehungen im Code durch Visualisierung
- Visualisieren der Auswirkung einer Änderung oder möglichen Änderung im Code
- Durchführen unbegrenzter Webleistungs- und Auslastungstests
- Entwerfen architektonischer Layerdiagramme zur Überprüfung des Codes und Implementierung in der Architektur

1.1.2 Anforderungen an Hard- und Software

Haben sich in der Vergangenheit die Hardwareanforderungen von Version zu Version in die Höhe geschraubt, so bleiben sie diesmal etwa auf dem gleichen Niveau wie beim Vorgänger Visual Studio 2010. Die folgende Auflistung kann lediglich eine Orientierungshilfe sein:

- Betriebssystem: Windows 8, Windows 7, Windows Server 2012, Windows Server 2008
- Unterstützte Architekturen: 32-Bit (x86) und 64-Bit (x64)
- Prozessor: 1,6-GHz-Pentium III+
- RAM: 1 GB verfügbarer physischer Arbeitsspeicher (x86) bzw. 2 GB (x64)
- Festplatte: 10 GB Speicherplatzbedarf
- Grafikkarte: DirectX 9-fähig mit einer Mindestauflösung von 1024 x 768 Pixeln
- DVD-ROM Laufwerk

Die Parameter von Prozessor und RAM sind als untere Grenzwerte zu verstehen, können aber für die Express-Editionen sicherlich noch etwas unterschritten werden.

Ganz wichtig:

HINWEIS: Wollen Sie WinRT-Anwendungen für Windows 8 entwickeln, so ist das Betriebssystem Windows 8 für das Entwicklungssystem unerlässlich!

Weiterhin ist zu beachten:

- Das neue .NET Framework 4.5 wird von Windows XP nicht mehr unterstützt – motten Sie also Ihren alten Computer ein.
- Das .NET-Framework 3.5 ist nicht mehr in Windows 8 enthalten, es muss nachinstalliert werden oder die Anwendungen müssen auf die Version 4 aktualisiert werden.
- Der SQL Server Express 2012 ist nicht mehr im Installationspaket enthalten, sondern muss separat heruntergeladen werden. Alternativ steht nach der Installation von Visual Studio der neue SQL Server Express 2012 LocalDB zur Verfügung

1.2 Unser allererstes C#-Programm

Jeder Weg, und ist er noch so weit, beginnt mit dem ersten Schritt! Nachdem die Mühen der Installation überstanden sind, wird es Zeit für ein allererstes C#-Programm. Wir verzichten allerdings auf das abgedroschene "Hello World" und wollen gleich mit etwas Nützlicherem beginnen, nämlich der Umrechnung von Euro in Dollar.

Auch allein mit dem .NET Framework SDK, also ohne das teure Visual Studio 2012, kann man vollwertige Programme entwickeln. Das wollen wir jetzt unter Beweis stellen, indem wir eine kleine Euro-Dollar-Applikation als so genannte *Konsolenanwendung* – dem einfachsten Anwendungstyp – schreiben.

1.2.1 Vorbereitungen

Voraussetzungen sind lediglich ein simpler Texteditor und der C#-Kommandozeilencompiler *csc.exe*.

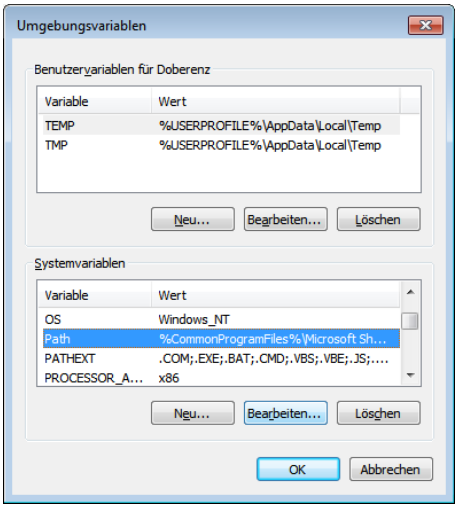
Compilerpfad eintragen

Der C#-Compiler *csc.exe* befindet sich, ziemlich versteckt, im Verzeichnis

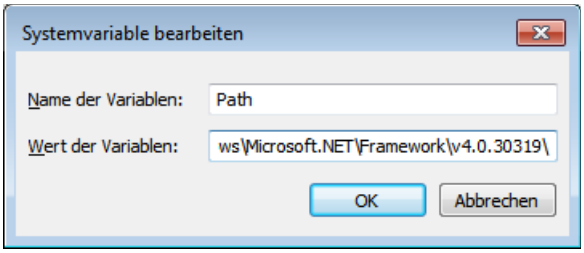
```
\\Windows\Microsoft.NET\Framework\v4.0.30319
```

Da das Kompilieren direkt an der Kommandozeile ausgeführt werden soll, werden wir *csc.exe* in den Windows-Pfad aufnehmen, um so seinen Aufruf von jedem Ordner des Systems aus zu ermöglichen:

- Sie finden den Dialog zur Einstellung der *Path*-Umgebungsvariablen in der Windows-Systemsteuerung unter dem Eintrag *System* im Aufgabenbereich "Erweiterte Systemeinstellungen".
- Im Dialog *Systemeigenschaften* klicken Sie auf der Registerkarte *Erweitert* auf die Schaltfläche *Umgebungsvariablen...*.



- Wählen Sie in der unteren Liste *Systemvariablen* den *Path*-Eintrag und klicken Sie auf die *Bearbeiten...*-Schaltfläche (siehe Abbildung).
- Hängen Sie den Namen des .NET Framework-Verzeichnisses, in welchem sich *csc.exe* befindet (*C:\Windows\Microsoft.NET\Framework\v4.0.30319*), durch ein Semikolon (;) getrennt hinten dran:

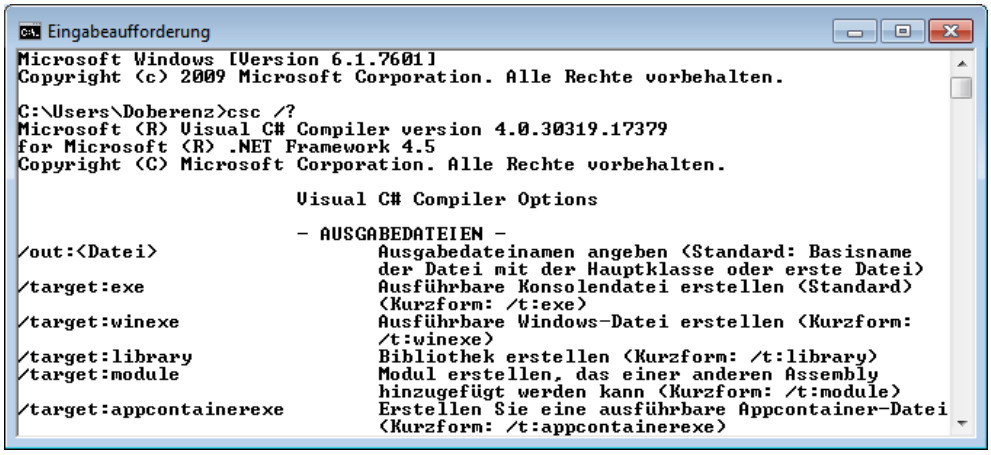


Die erfolgreiche Übernahme der Änderungen an den *Path*-Umgebungsvariablen können Sie in einem kleinen Test überprüfen, bei dem Sie sich als durchaus nützlichen Nebeneffekt gleich die vielfältigen Optionen des Compilers anzeigen lassen.

Wechseln Sie dazu über das Windows-Startmenü zur Eingabeaufforderung(*Start|Programme|Zubehör|Eingabeaufforderung*) und geben Sie (von einem beliebigen Verzeichnis aus) den folgenden Befehl ein, den Sie mit *Enter* abschließen:

```
csc /?
```

Aus der endlosen Parameterliste, die angezeigt wird, ist für uns die Option */target:exe* (abgekürzt */t:exe*) besonders interessant, da wir damit später unsere Konsolenanwendung kompilieren wollen (siehe Abbildung).



Vom Funktionieren des Compilers können Sie sich erst dann überzeugen, wenn Sie eine C#-Source-Datei erstellt haben (siehe folgender Abschnitt).

1.2.2 Quellcode schreiben

Öffnen Sie den im Windows-Zubehör enthaltenen Editor und tippen Sie, ohne lange darüber nachzudenken, einfach den folgenden Text ein:

```
using System;
class KonsolenDemo
{
  static void Main()
  {
    int i;
    Console.WriteLine("Umrechnung Euro in Dollar");
    do
    {
       float kurs, euro, dollar;
       Console.Write("Kurs 1 : ");
       kurs = Convert.ToSingle(Console.ReadLine());
       Console.Write("Euro: ");
       euro = Convert.ToSingle(Console.ReadLine());
       dollar = euro * kurs;
       Console.WriteLine("Sie erhalten " + dollar.ToString("0.00 $"));
       Console.Write("Programm beenden? (j/n)");
       string s = Console.ReadLine();
```

```
        i = string.Compare(s, "j");
      } while(i != 0);
    }
}
```

HINWEIS: Achten Sie auf die exakte Einhaltung der Groß-/Kleinschreibung!

Speichern Sie die Datei unter dem Namen *EuroDollar.cs* in ein extra dafür angelegtes Verzeichnis, z.B. *\EuroDollarKonsole*, ab.

1.2.3 Programm kompilieren und testen

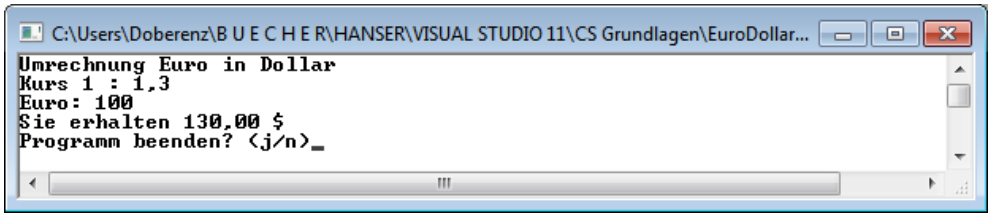
Um bequem an der Kommandozeile arbeiten zu können, kopieren Sie zunächst die Datei *cmd.exe* (Eingabeaufforderung aus *...\Windows\System32*) in dasselbe Verzeichnis, in welchem sich auch die Datei *EuroDollar.cs* befindet.

Klicken Sie doppelt auf *cmd.exe* und rufen Sie dann den C#-Compiler wie folgt auf:

```
csc /t:exe EuroDollar.cs
```

Nach dem erfolgreichen Kompilieren wird sich eine neue Datei *EuroDollar.exe* im Anwendungsverzeichnis befinden, ansonsten gibt der Compiler eine Fehlermeldung aus.

Klicken Sie doppelt auf die Datei *EuroDollar.exe* und führen Sie Ihr erstes C#-Programm aus!



1.2.4 Einige Erläuterungen zum Quellcode

Da wir auf die Grundlagen der Sprache C# erst in den späteren Kapiteln ausführlich zu sprechen kommen, sollen die folgenden Informationen zunächst nur allererste Eindrücke vermitteln.

Befehlszeilen und Gültigkeitsbereiche

Das Ende einer C#-Befehlszeile wird durch ein Semikolon (;) markiert. Der Zeilenumbruch spielt also keine Rolle! Gültigkeitsbereiche sind durch geschweifte Klammern { ... } eingegrenzt. In der Regel muss also die Anzahl der öffnenden Klammern gleich der Anzahl schließender Klammern sein.

using-Anweisung

Mit der ersten Anweisung

```
using System;
```

binden Sie den *System*-Namensraum (*Namespace*) ein. Das hat den Vorteil, dass Sie statt

```
System.Console.WriteLine("Umrechnung Euro in Dollar");
```

nur noch

```
Console.WriteLine("Umrechnung Euro in Dollar");
```

schreiben müssen.

class-Anweisung

Mit dieser Anweisung erzeugen Sie eine neue Klasse, in welcher in unserem Beispiel die *Main*-Prozedur deklariert wird. Diese definiert den Einsprungpunkt der Konsolenanwendung (also dort, wo das Programm startet).

HINWEIS: Ein C#-Programm besteht aus mindestens einer **.cs*-Textdatei mit einer Klasse.

WriteLine- und ReadLine-Methoden

Diese Methoden der *Console*-Klasse erlauben die Aus- und Eingabe von Text.

Während *Write* nur den Text an der aktuellen Position ausgibt, erzeugt *WriteLine* zusätzlich einen Zeilenumbruch.

ReadLine erwartet die Betätigung der *Enter*-Taste und liefert die vorher eingegebenen Zeichen als Zeichenkette zurück.

Assemblierung

Bei der vom C#-Compiler erzeugten Datei *EuroDollar.exe* handelt es sich **nicht** um eine herkömmliche Exe-Datei, sondern um eine so genannte *Assemblierung*, die erst in Zusammenarbeit mit der CLR (*Common Language Runtime*) des .NET-Frameworks in Maschinencode verwandelt wird (siehe Abschnitt 1.5.2).

1.2.5 Konsolenanwendungen sind langweilig

Zwar hat seit Visual C# 2005 die Klasse *System.Console* zahlreiche neue Mitglieder erhalten, mit denen auch verschiedenste farbliche Effekte möglich sind, trotzdem: Bei wem weckt das Outfit einer Konsolenanwendung noch positive Emotionen[1]?

---

[1] Von nostalgischen Erinnerungen an die DOS-Steinzeit einmal abgesehen.

Als einfaches Hilfsmittel zum Erlernen von C# und als Notnagel für all jene, die sich Visual Studio 2012 vorerst nicht leisten wollen oder können, hat dieser einfache Anwendungstyp aber durchaus noch seine Daseinsberechtigung.

Mit Visual Studio 2012 werden wir im Praxisteil dieses Kapitels (Abschnitt 1.7.2) das gleiche Problem nochmals lösen, dann allerdings mit einer attraktiven Windows-Oberfläche. Bevor es aber damit losgehen kann, sollten Sie sich ein wenig mit der Windows-Philosophie anfreunden.

1.3 Die Windows-Philosophie

Eine moderne Programmiersprache wie C# gibt Ihnen die faszinierende Möglichkeit, eigene Windows-Programme mit relativ geringem Aufwand und nach nur kurzer Einarbeitungszeit selbst zu entwickeln. Allerdings fällt der Einstieg umso leichter, je schneller man sich Klarheit über die zunächst fremdartig anmutende, aber dann doch einfach und gleichzeitig genial erscheinende Windows-Philosophie verschafft.

1.3.1 Mensch-Rechner-Dialog

Die Art und Weise, **wie** die Kommunikation mit dem Benutzer (Mensch-Rechner-Dialog) abläuft, dürfte wohl der gravierendste Unterschied zwischen einer klassischen Konsolenanwendung und einer typischen Windows-Anwendung sein. Wie Sie es bereits im Einführungsbeispiel 1.2 kennen gelernt haben, "wartet" das Konsolenprogramm auf eine Eingabe, indem die Tastatur zyklisch abgefragt wird.

Unter Windows werden hingegen Ein- und Ausgaben in so genannte "Nachrichten" umgesetzt, die zum Programm geschickt und dort in einer Nachrichtenschleife kontinuierlich verarbeitet werden. Daraus ergibt sich ein grundsätzlich anderes Prinzip der Interaktion zwischen Mensch und Rechner:

- Während bei einer Konsolenanwendung alle Initiativen für die Benutzerkommunikation vom Programm ausgehen, hat in einer Windows-Anwendung der Bediener den Hut auf. Er bestimmt durch seine Eingaben den Ablauf der Rechnersitzung.
- Während eine Konsolenanwendung in der Regel in einem einzigen Fenster läuft, erfolgt die Ausgabe bei einer Windows-Anwendung meist in mehreren Fenstern.

1.3.2 Objekt- und ereignisorientierte Programmierung

Vergleicht man den Programmaufbau einer Konsolenanwendung, welche aus einer langen Liste von Anweisungen besteht, mit einer Windows-Anwendung, so stellt man folgende Hauptunterschiede fest:

- Im Konsolenprogramm werden die Befehle sequenziell abgearbeitet, d.h. Schritt für Schritt hintereinander. Den Gesamtablauf kontrolliert in der Regel ein Hauptprogramm.

- In einer Windows-Anwendung laufen alle Aktionen objekt- und ereignisorientiert ab, eine streng vorgeschriebene Reihenfolge für die Eingabe und Abarbeitung der Befehle gibt es nicht mehr. Für jede Aktivität des Anwenders ist ein Programmteil zuständig, der weitestgehend unabhängig von anderen Programmteilen agieren kann und muss. Daraus folgt auch das Fehlen eines Hauptprogramms im herkömmlichen Sinn!

Ein Windows-Programmierer hat sich vor allem mit den folgenden Begriffen auseinander zu setzen:

Objekte (Objects)

Das sind zunächst die Elemente der Windows-Bedienoberfläche, denen wiederum Eigenschaften, Ereignisse und Methoden zugeordnet werden.

Beschränken wir uns der Einfachheit halber zunächst nur auf die visuelle Benutzerschnittstelle, so haben wir es in C# mit folgenden Objekten zu tun:

- **Formulare:**
  Das sind die Fenster, in welchen eine C#-Anwendung ausgeführt wird. In einem Formular (*Form*) können weitere untergeordnete Formulare, Komponenten (siehe unten), Text oder Grafik enthalten sein.
- **Steuerelemente:**
  Diese tauchen in vielfältiger Weise als Schaltflächen (*Button*), Textfelder (*TextBox)* etc. auf. Sie stellen die eigentliche Benutzerschnittstelle dar, über welche mittels Tastatur oder Maus Eingaben erfolgen oder die der Ausgabe von Informationen dienen.

Der Objektbegriff wird auch auf die nichtvisuellen Elemente (z.B *Timer, DataSet...*) ausgedehnt, und das geht schließlich so weit, dass innerhalb des .NET-Frameworks sogar alle Variablen als Objekte betrachtet werden. Natürlich dürfen auch Sie als Programmierer auch eigene Objekte/ Komponenten entwickeln und hinzufügen.

Eigenschaften (Properties)

Unter diesem Begriff versteht man die Attribute von Objekten, wie z.B. die Höhe (*Height*) und die Breite (*Width*) oder die Hintergrundfarbe (*BackColor*) eines Formulars. Jedes Objekt verfügt über seinen eigenen Satz von Eigenschaften, die teilweise nur zur Entwurfs- oder nur zur Laufzeit veränderbar sind.

Methoden (Methods)

Das sind die im Objekt definierten Funktionen und Prozeduren, die gewissermaßen das "Verhalten" beim Eintreffen einer Nachricht bestimmen. So säubert z.B. die *Clear*-Methode den Inhalt einer *ListBox*. Eine Methode kann z.B. auch das Verhalten des Objekts bei einem Mausklick, einer Tastatureingabe oder sonstigen Ereignissen (siehe unten) definieren. Im Unterschied zu den oben genannten Eigenschaften (Properties), die eine "statische" Beschreibung liefern, bestimmen Methoden die "dynamischen" Fähigkeiten des Objekts.

Ereignisse (Events)

Dies sind Nachrichten, die vom Objekt empfangen werden. Sie stellen die eigentliche Windows-Schnittstelle dar. So ruft z.B. das Anklicken eines Steuerelements mit der Maus in Windows ein *Click*-Ereignis hervor. Aufgabe eines Windows-Programms ist es, auf alle Ereignisse gemäß dem Wunsch des Anwenders zu reagieren. Dies geschieht in so genannten *Ereignisbehandlungsroutinen* (Eventhandler).

Diese (zugegebenermaßen ziemlich oberflächlichen und unvollständigen) Erklärungen zur objektorientierten Programmierung sollen vorerst zum Einstieg genügen, theoretisch sauber wird die OOP erst im Kapitel 3 erläutert.

1.3.3 Programmieren mit Visual Studio 2012

Nicht nur Konsolenanwendungen, sondern auch Windows- und Web-Anwendungen lassen sich rein theoretisch mit den (kostenlos erhältlichen) Werkzeugen des *Microsoft .NET Framework SDK* erstellen. Allerdings ist dies extrem umständlich, da dazu zeitaufwändige Überlegungen zur Gestaltung der Benutzerschnittstelle[1] und ständiges Nachschlagen in der Dokumentation erforderlich wären. Die intuitive Entwicklungsumgebung Visual Studio befreit Sie von diesem, besonders bei größeren Projekten sehr lästigen und nervtötenden Herumwursteln und erlaubt (unabhängig von der verwendeten Programmiersprache) eine systematische Vorgehensweise in vier Etappen:

1. Visueller Entwurf der Bedienoberfläche
2. Zuweisen der Objekteigenschaften
3. Verknüpfen der Objekte mit Ereignissen
4. Kompilieren und Testen der Anwendung

Bereits die *erste Etappe* weist einen deutlichen Unterschied zur Konsolenprogrammiertechnik auf: Am Anfang steht der Oberflächenentwurf!

Ausgangsbasis ist das vom Editor bereitgestellte Startformular (*Form1*), welches mit diversen Steuerelementen, wie Schaltflächen (*Buttons*) oder Editierfenstern (*TextBox*en), ausgestattet wird. Im Werkzeugkasten finden Sie ein nahezu komplettes Angebot der Windows-typischen Steuerelementen. Diese werden ausgewählt, mittels Maus an ihre endgültige Position gezogen und (falls nötig) in ihrer Größe verändert.

Bereits während der ersten Etappe hat man, mehr oder weniger unbewusst, Eigenschaften, wie Position und Abmessungen von Formularen und Steuerelementen, verändert. In der *zweiten Etappe* braucht man sich eigentlich nur noch um die Eigenschaften zu kümmern, die von den Standardeinstellungen (Defaults) abweichen.

Die *dritte Etappe* haucht Leben in unsere bislang nur mit statischen Attributen ausgestatteten Objekte. Hier muss in so genannten *Ereignisbehandlungsroutinen* (Eventhandlern) festgelegt werden, **wie** das Formular oder das betreffende Steuerelement auf bestimmte Ereignisse zu reagieren hat. Visual Studio stellt auch hier "vorgefertigten" Rahmencode (erste und letzte Anweisung)

[1] Das geht hin bis zum Abzählen von Pixeln!

für alle zum jeweiligen Objekt passenden Ereignisse zur Verfügung. Der Programmierer füllt diesen Rahmen mit C#-Quellcode aus. Hier können Methoden oder Prozeduren aufgerufen werden, aber auch Eigenschaften anderer Objekte lassen sich während der Laufzeit neu zuweisen.

In der *vierten Etappe* schlägt schließlich die Stunde der Wahrheit. Das von Ihnen geschriebene Programm wird vom C#-Compiler in einen Zwischencode übersetzt und läuft damit auf jedem Rechner, auf dem das .NET Framework installiert ist.

Allerdings ist die Arbeit des Programmierers nur in seltenen Fällen bereits nach einmaligem Durchlaufen aller vier Etappen getan. In der Regel müssen Fehler ausgemerzt und Ergänzungen vorgenommen werden, sodass sich der beschriebene Entwicklungszyklus auf ständig höherem Level so lange wiederholt, bis ein zufrieden stellendes Ergebnis erreicht ist.

Der in diesem Zyklus praktizierte visuelle Oberflächenentwurf, verbunden mit dem ereignisorientierten Entwurfskonzept, macht *Visual Studio 2012* zu einer hocheffektiven Entwicklungsumgebung für Windows- und Web-Anwendungen.

1.4 Die Entwicklungsumgebung Visual Studio 2012

Visual Studio 2012 ist eine universelle Entwicklungsumgebung (IDE[1]) für Windows- und für Web-Anwendungen, die auf Microsofts .NET-Technologie basieren. Alle notwendigen Tools, wie z.B. für den visuellen Oberflächenentwurf, für die Codeprogrammierung und für die Fehlersuche, werden bereitgestellt.

C# ist nur eine der möglichen objektorientierten Sprachen, die Sie unter Visual Studio 2012 einsetzen können. So werden z.B. noch Visual Basic, Visual C++ und Visual F# unterstützt.

HINWEIS: Der vorliegende Abschnitt soll lediglich einen allerersten Eindruck der IDE vermitteln, der sich erst durch die konkrete Arbeit mit den Praxisbeispielen am Ende dieses Kapitels verfestigen wird!

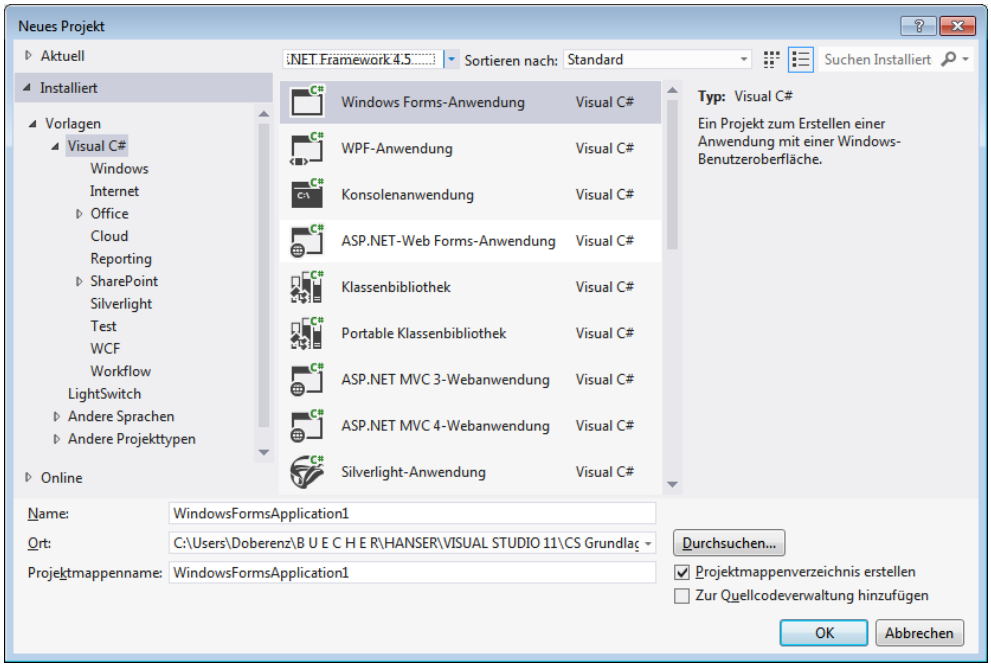
1.4.1 Neues Projekt

Wählen Sie auf der Startseite von Visual Studio 2012 den Menüpunkt *Neues Projekt...,* so öffnet sich der Startdialog *Neues Projekt* mit einem umfangreichen und zunächst verwirrenden Angebot an unterschiedlichen Vorlagen[2] für Visual C#-Projekttypen, wobei für den Einsteiger zunächst die klassische *Windows Forms-Anwendung* empfohlen wird.

Visual Studio 2012 erlaubt es, Programme für verschiedene .NET-Framework-Versionen zu entwickeln (Multi-Targeting, siehe obere Klappbox).

[1] *Integrated Developers Environment*

[2] Die Abbildung bezieht sich auf die Professional-Edition, bei den anderen Editionen von Visual Studio 2012 ist das Angebot an unterschiedlichen Projekttypen bzw. Vorlagen mehr oder weniger eingeschränkt.



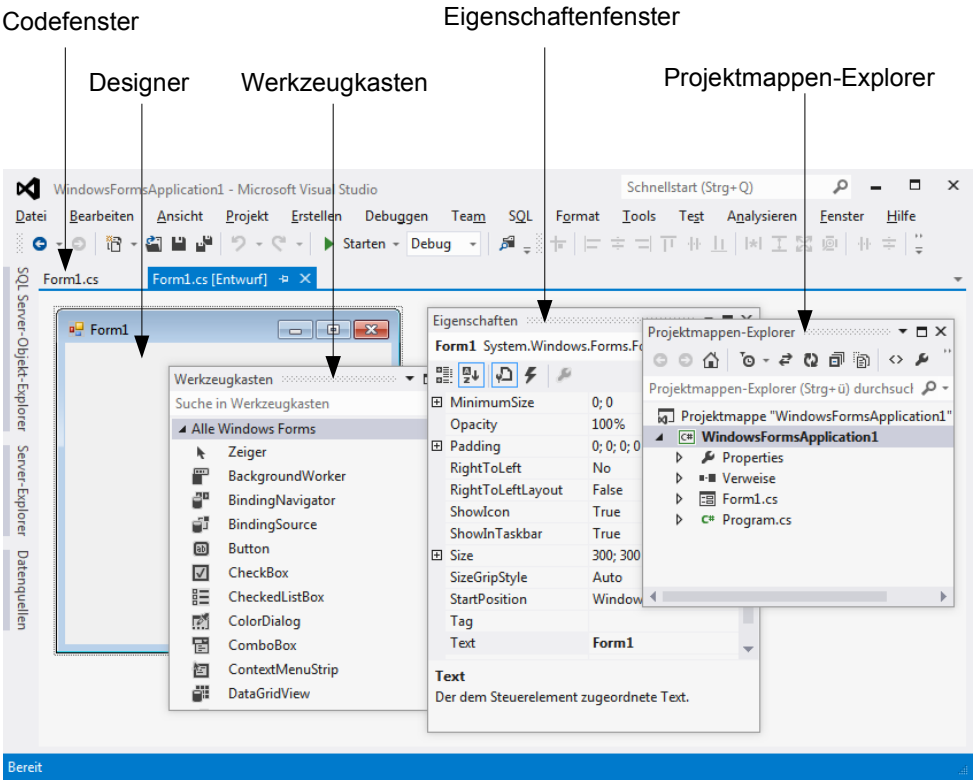
Haben Sie das Häkchen bei *Projektmappenverzeichnis erstellen* gesetzt, so erzeugt Visual Studio automatisch einen Unterordner mit dem Namen des Projekts in dem als Speicherort eingetragenen Hauptverzeichnis.

HINWEIS: Namen und Ort des Projekts sollten Sie unbedingt **vor** dem Klicken der *OK*-Schaltfläche eintragen, denn ein späteres Umbenennen ist recht umständlich.

1.4.2 Die wichtigsten Fenster

Haben Sie als Projekttyp beispielsweise *Windows Forms-Anwendung* gewählt, könnte Ihnen Visual Studio 2012 etwa den folgenden Anblick bieten, wobei auf die für den Einsteiger zunächst wichtigsten Fenster besonders hingewiesen wird.

HINWEIS: Falls sich eines der Fenster versteckt hat, können Sie es jederzeit über das *Ansicht*-Menü herbeiholen.



Der Projektmappen-Explorer

Da es unter Visual Studio 2012 möglich ist, mehrere Projekte gleichzeitig zu bearbeiten, gibt es eine Projektmappendatei mit der Extension *.sln* (Solution), deren Inhalt im Projektmappen-Explorer (*STRG+R*) übersichtlich angezeigt wird. Sie können dieses Fenster deshalb ohne Übertreibung als "Schaltzentrale" Ihres Projekts betrachten.

Die zu jedem einzelnen C#-Projekt gehörigen Dateien und Einstellungen werden in einer XML-Datei mit der Extension *.csproj* (C#-Projekt) verwaltet.

HINWEIS: Öffnen Sie Ihre Projekte immer über die *.sln*-Projektmappendatei, statt über die *.csproj*-Projektdatei, selbst wenn nur ein einziges Projekt enthalten ist!

Zur Bedeutung der einzelnen Einträge bzw. Dateien:

- *Properties*
  Hier sind verschiedene Dateien zusammengefasst, die die Projekteigenschaften bestimmen. *AssemblyInfo.cs* enthält z.B allgemeine Infos zur Assemblierung des Projekts, wie Titel, Beschreibung, Versionsnummer, Copyright. Weitere Dateien beziehen sich auf die Ressourcen und die Projekteinstellungen.

- *Verweise*
  Hier sind die aktuell für das Projekt gültigen Verweise auf Namensräume bzw. Assemblierungen enthalten. Standardmäßig hat Visual Studio bereits die wichtigsten Verweise eingestellt, weitere können Sie über das Kontextmenü der rechten Maustaste hinzufügen.
- *Form1.cs*
  Diese Datei enthält eine partielle Klasse[1], die den von Ihnen selbst hinzugefügten Code von *Form1* kapselt.
- *Form1.Designer.cs*
  Diese Datei enthält eine partielle Klasse, die den vom Windows Forms Designer automatisch generierten Code von *Form1* kapselt. Der gesamte Code von *Form1* ist also zwischen den partiellen Klassen in *Form1.cs* und *Form1.Designer.cs* aufgeteilt.
- *Program.cs*
  Eine statische Klasse, welche die *Main*-Methode (den Einsprungpunkt der Anwendung) enthält. In dieser Methode wird durch Aufruf von *Application.Run* eine Nachrichtenschleife gestartet, die ununterbrochen auf Ereignisse wartet, damit die Anwendung darauf reagieren kann.

Durch Doppelklick auf eine dieser Dateien können Sie diese im Designer bzw. im Codefenster zwecks Bearbeitung öffnen.

Der Designer

Im Designer-Fenster entwerfen Sie die Programmoberfläche bzw. Benutzerschnittstelle. Ähnlich wie bei einem Zeichenprogramm entnehmen Sie dem Werkzeugkasten Steuerelemente und ziehen diese per Drag & Drop auf ein Formular. Hier können Sie weitere Eigenschaften, wie z.B. Größe und Position, direkt mit der Maus und andere, wie z.B. Farbe und Schriftart, über das Eigenschaften-Fenster ändern.

Der Werkzeugkasten

Der Werkzeugkasten wird häufig benötigt (Menü *Ansicht/Werkzeugkasten* bzw. *STRG+W, X*). Auf verschiedenen Registerseiten, die später von Ihnen auch frei konfiguriert werden können, finden Sie eine umfangreiche Palette verschiedenster Steuerelemente für Windows-Forms-Anwendungen.

Das Eigenschaften-Fenster

Im Eigenschaften-Fenster (Menü *Ansicht/Eigenschaftenfenster* bzw. *F4*) werden die zur Entwurfszeit editierbaren Eigenschaften des gerade aktiven Steuerelements aufgelistet[2]. Normalerweise hat jede Eigenschaft bereits einen Standardwert, den Sie in vielen Fällen übernehmen können.

Das Aktivieren eines bestimmten Steuerelements geschieht entweder durch Anklicken desselben auf dem Formular, oder durch dessen Auswahl in der Klappbox am oberen Rand des Eigenschaften-Fensters.

---

[1] Das Konzept partieller Klassen wird im OOP-Kapitel (Abschnitt 3.7.3) erläutert.

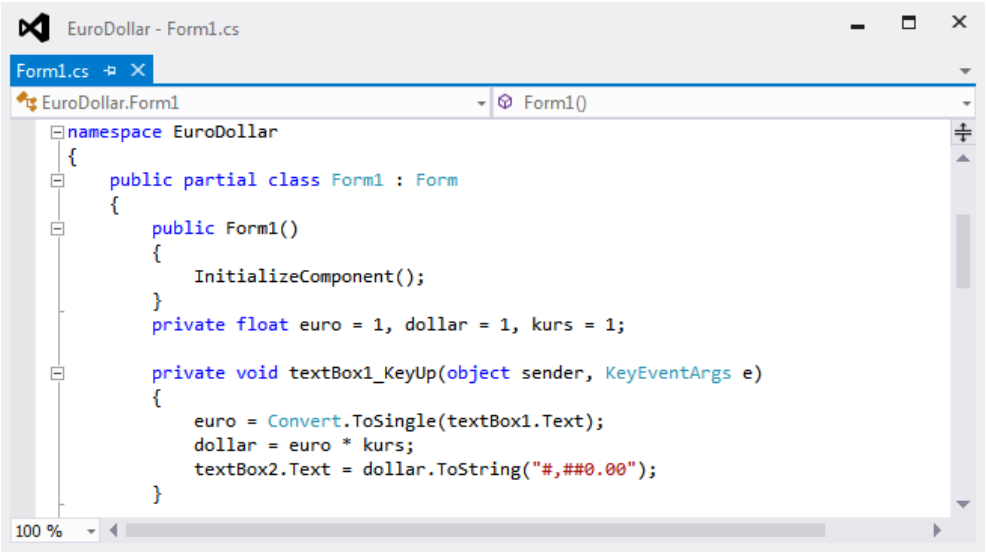
[2] Lassen Sie sich nicht davon irritieren, dass das Eigenschaftenfenster auf einer extra Registerseite auch die zum Steuerelement gehörigen Ereignisse anbietet.

Das Codefenster

Für die eigentliche Programmierung ist das Codefenster zuständig. Logischerweise wird dies damit auch zu Ihrem Hauptbetätigungsfeld als C#-Programmierer. Um zum Beispiel das zu einem Formular gehörige Codefenster zu öffnen, klicken Sie auf dasselbe mit der rechten Maustaste und wählen im Kontextmenü *Code anzeigen* (F7). Die folgende Abbildung zeigt das Codefenster für *Form1* im Praxisbeispiel 1.7.2.

Der Code-Editor unterstützt Sie auf vielfältige Weise beim Schreiben von Quellcode, so markiert er Wörter farblich, unterbreitet Ihnen Vorschläge, weist Sie auf Fehler hin oder rückt den Text automatisch ein.

Bei allem Verständnis für Ihre Ungeduld: bevor Sie mit praktischen Beispielen beginnen, empfehlen wir Ihnen zunächst eine kleine Exkursion in die Untiefen von .NET.



1.5 Microsofts .NET-Technologie

Ganz ohne Theorie geht nichts! In diesem leider etwas "trockenen" Abschnitt wollen wir den Einsteiger mit der grundlegenden .NET-Philosophie und den damit verbundenen Konzepten, Begriffen und Features vertraut machen. Dazu dürfen Sie Ihrem Rechner ruhig einmal eine Pause gönnen.

1.5.1 Zur Geschichte von .NET

Das Kürzel .NET ist die Bezeichnung für eine gemeinsame Plattform für viele Programmiersprachen. Beim Kompilieren von .NET-Programmen wird der jeweilige Quelltext in MSIL (*Microsoft Intermediate Language*) übersetzt. Es gibt nur ein gemeinsames Laufzeitsystem für alle Sprachen, die so genannte CLR (*Common Language Runtime*), das die MSIL-Programme ausführt.

Die im Jahr 2002 eingeführte .NET-Technologie wurde deshalb notwendig, weil sich die Anforderungen an moderne Softwareentwicklung dramatisch verändert hatten, wobei das rasant wachsende Internet mit seinen hohen Ansprüchen an die Skalierbarkeit einer Anwendung, die Verteilbarkeit auf mehrere Schichten und ausreichende Sicherheit der hauptsächliche Motor war, sich nach einer grundlegend neuen Sprachkonzeption umzuschauen.

Mit .NET fand ein radikaler Umbruch in der Geschichte der Softwareentwicklung statt. Nicht nur dass jetzt "echte" objektorientierte Programmierung zum obersten Dogma erhoben wird, nein, auch eine langjährig bewährte Sprache wie das alte Visual Basic wurde völlig umgekrempelt und die einst hoch gelobte COM-Technologie zum Auslaufmodell erklärt!

Warum eine extra Programmiersprache?

C# wurde ausschließlich für das .NET-Framework konzipiert, wobei versucht wurde, das Beste aus den etablierten Programmiersprachen Java, JavaScript, Visual Basic und C++ zu kombinieren, ohne aber deren Nachteile zu übernehmen.

Da sich die etablierten Sprachen nicht ohne größere Kompromisse an das .NET-Framework anpassen ließen, haben die .NET-Entwickler die Gelegenheit beim Schopf gepackt und eine "maßgeschneiderte" .NET-Sprache entwickelt. C# setzt sauber auf dem .NET-Framework auf und kommt ohne "faule" Kompatibilitäts-Kompromisse aus, wie sie z.B. teilweise bei Visual Basic erforderlich waren.

Ein großer Teil der .NET-Klassenbibliotheken ist selbst in C# programmiert, C# ist somit die Systemsprache von .NET und spielt gewissermaßen die gleiche Rolle wie C++ für Windows.

Als konsequent objektorientierte Sprache erfüllt C# folgende Kriterien:

- **Abstraktion**
  Die Komplexität eines Geschäftsproblems ist beherrschbar, indem eine Menge von abstrakten Objekten identifiziert werden können, die mit dem Problem verknüpft sind.
- **Kapselung**
  Die interne Implementation einer solchen Abstraktion wird innerhalb des Objekts versteckt.
- **Polymorphie**
  Ein und dieselbe Methode kann auf mehrere Arten implementiert werden.
- **Vererbung**
  Es wird nicht nur die Schnittstelle, sondern auch der Code einer Klasse vererbt (Implementations-Vererbung statt der COM-basierten Schnittstellen-Vererbung).

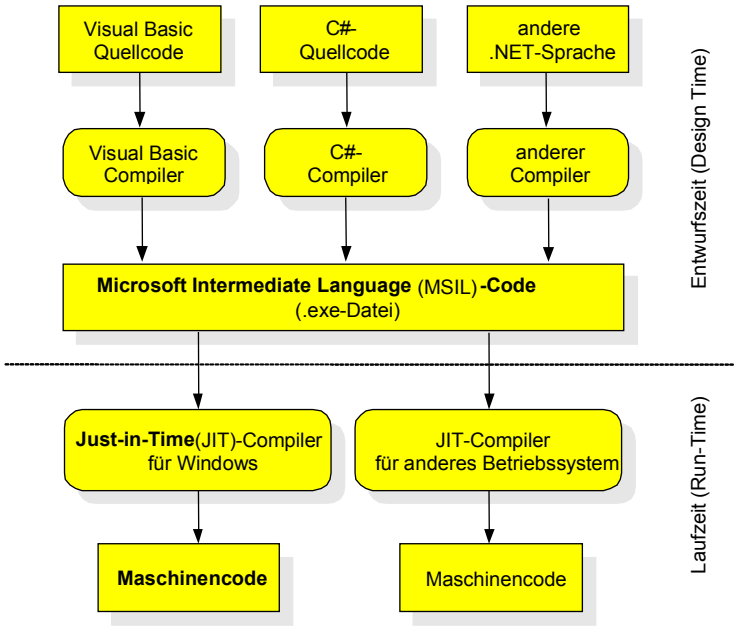
Microsoft kann natürlich nicht über Nacht die COM-Technologie auf die Müllkippe entsorgen, denn zu viele Programmierer würden dadurch auf immer und ewig verprellt werden und sich frustriert einer stabileren Entwicklungsplattform zuwenden. Aus diesem Grund wird COM auch in .NET noch einige Zeit sein Gnadenbrot erhalten.

Wie funktioniert eine .NET-Sprache?

Jeder in einer beliebigen .NET-Programmiersprache geschriebene Code wird beim Kompilieren in einen Zwischencode, den so genannten MSIL-Code (*Microsoft Intermediate Language Code*), übersetzt, der unabhängig von der Plattform bzw. der verwendeten Hardware ist und dem man es auch nicht mehr ansieht, in welcher Sprache seine Source geschrieben wurde.

HINWEIS: Das .NET-Konzept sieht fast wie ein Java-Plagiat aus, allerdings mit dem "feinen" Unterschied, dass es nicht an eine bestimmte Programmiersprache gebunden ist!

Erst wenn der MSIL-Code von einem Programm zur Ausführung genutzt werden soll, wird er vom *Just-in-Time(JIT)-Compiler* in Maschinencode übersetzt[1]. Ein .NET-Programm wird also vom Entwurf bis zu seiner Ausführung auf dem Zielrechner tatsächlich zweimal kompiliert (siehe folgende Abbildung).



HINWEIS: Für die Installation eines Programms ist in der Regel lediglich die Weitergabe des MSIL-Codes erforderlich. Voraussetzung ist allerdings das Vorhandensein der .NET-Laufzeitumgebung (CLR), die Teil des .NET Frameworks ist, auf dem Zielrechner.

[1] Der Begriff "jeder Code" schließt z.B. auch den Code der ASP.NET-Seiten ein.

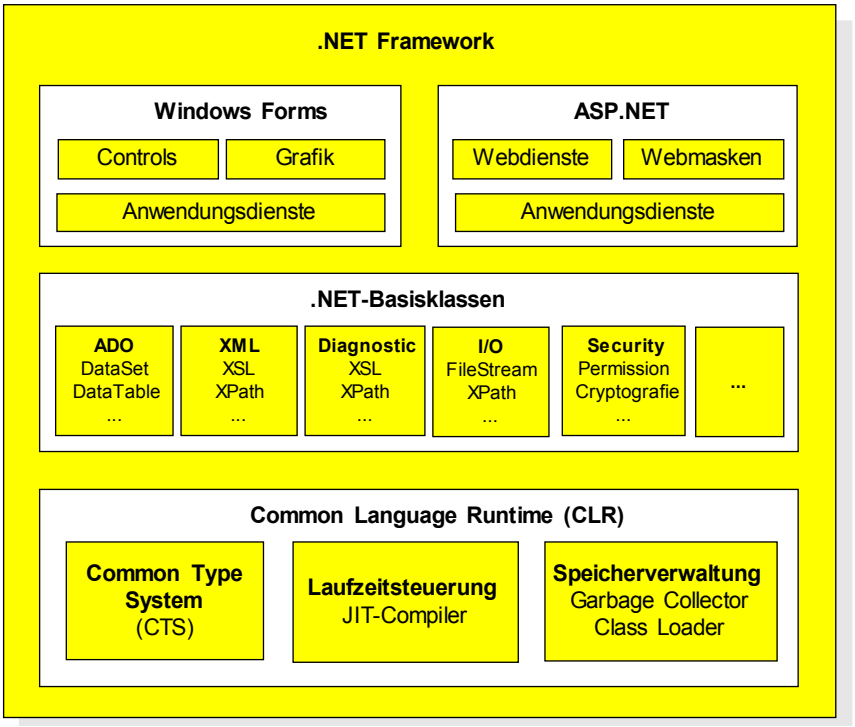
1.5.2 .NET-Features und Begriffe

Mit Einführung von Microsofts .NET-Technologie prasselte auch eine Vielzahl neuer Begriffe auf die Entwicklergemeinde ein. Wir wollen hier nur die wichtigsten erklären.

.NET-Framework

.NET ist die Infrastruktur für die gesamte .NET-Plattform, es handelt sich hierbei gleichermaßen um eine Entwurfs- wie um eine Laufzeitumgebung, in welcher Windows- und Web-Anwendungen erstellt und verteilt werden können.

Die nachfolgende Abbildung versucht, einen groben Überblick über die Komponenten des .NET Frameworks zu geben.



Zu den wichtigsten Komponenten des .NET-Frameworks und den damit zusammenhängenden Begriffen zählen:

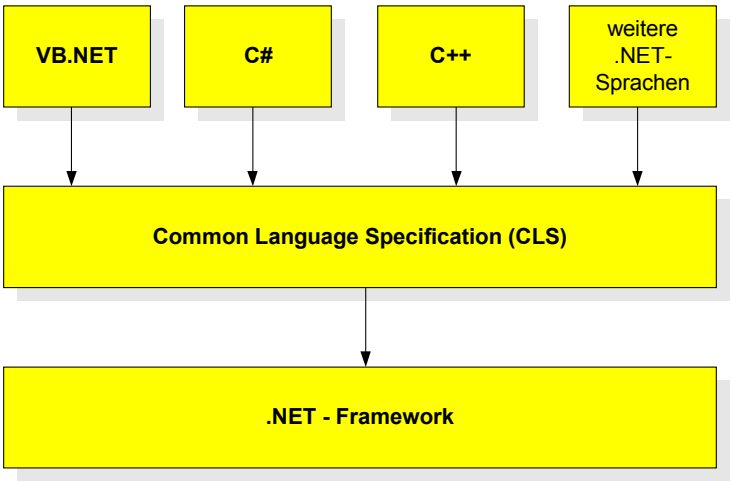
- Common Language Specification (CLS)
- Common Type System (CTS)
- Common Language Runtime (CLR)
- .NET-Klassenbibliothek

- diverse Basisklassenbibliotheken wie ADO.NET und ASP.NET
- diverse Compiler z.B. für C#, VB.NET ...

Im Folgenden sollen die einzelnen .NET-Bestandteile einer näheren Betrachtung unterzogen werden.

Die Common Language Specification (CLS)

Um den sprachunabhängigen MSIL-Zwischencode erzeugen zu können, müssen allgemein gültige Richtlinien und Standards für die .NET-Programmiersprachen existieren. Diese werden durch die *Common Language Specification* (CLS) definiert, die eine Reihe von Eigenschaften festlegt, die jede .NET-Programmiersprache erfüllen muss.



HINWEIS: Ganz egal, mit welcher .NET-Programmiersprache Sie arbeiten, der Quellcode wird immer in ein und dieselbe Intermediate Language (MSIL) kompiliert.

Besonders für die Entwicklung von .NET-Anwendungen im Team haben die Standards der CLS weitreichende positive Konsequenzen, denn es ist nun zweitrangig, in welcher .NET-Programmiersprache Herr Müller die Komponente X und Herr Meier die Komponente Y schreibt. Alle Komponenten werden problemlos miteinander interagieren!

Auf einen wichtigen Bestandteil des CLS kommen wir im folgenden Abschnitt zu sprechen.

Das Common Type System (CTS)

Ein Kernbestandteil der CLS ist das *Common Type System (CTS)*, es definiert alle Typen[1], die von der .NET-Laufzeitumgebung (CLR) unterstützt werden.

[1] Unter .NET spricht man allgemein von Typen und meint damit Klassen, Interfaces und Datentypen, die als Wert übergeben werden.

Alle diese Typen lassen sich in zwei Kategorien aufteilen:

- Wertetypen (werden auf dem Stack abgelegt)
- Referenztypen (werden auf dem Heap abgelegt)

Zu den Wertetypen gehören beispielsweise die ganzzahligen Datentypen und die Gleitkommazahlen, zu den Referenztypen zählen die Objekte, die aus Klassen instanziiert wurden.

HINWEIS: Dass unter .NET auch die Wertetypen letztendlich als Objekte betrachtet und behandelt werden, liegt an einem als *Boxing* bezeichneten Verfahren, das die Umwandlung eines Werte- in einen Referenztypen zur Laufzeit besorgt.

Warum hat Microsoft mit dem heiligen Prinzip der Abwärtskompatibilität gebrochen und selbst die fundamentalen Datentypen einer Programmiersprache neu definiert?

Als Antwort kommen wir noch einmal auf eine wesentliche Säule der .NET-Philosophie zu sprechen, auf die durch CLS/CTS manifestierte Sprachunabhängigkeit und auf die Konsequenzen, die dieses neue Paradigma nach sich zieht.

Microsofts .NET-Entwickler hatten gar keine andere Wahl, denn um Probleme beim Zugriff auf sprachfremde Komponenten zu vermeiden und um eine sprachübergreifende Programmentwicklung überhaupt zu ermöglichen, mussten die Spezifikationen der Programmiersprachen durch die CLS einander angepasst werden. Dazu müssen alle wesentlichen sprachbeschreibenden Elemente – wie vor allem die Datentypen – in allen .NET-Programmiersprachen gleich sein.

Da .NET eine Normierung der Programmiersprachen erzwingt, verwischen die Grenzen zwischen den verschiedenen Sprachen, und Sie brauchen nicht immer umzudenken, wenn Sie tatsächlich einmal auf eine andere .NET-Programmiersprache umsteigen wollen.

Als Lohn für die Mühen und den Mut, die eingefahrenen Gleise seiner altvertrauten Sprache zu verlassen, winken dem Entwickler wesentliche Vereinfachungen. So sind die Zeiten des alltäglichen Ärgers mit den Datentypen – wie z.B. bei der Übergabe eines Integers an eine C-DLL – endgültig vorbei. Bei so viel Licht gibt es natürlich auch Schatten:

HINWEIS: Mit einem der inzwischen zahlreich und kostenlos verfügbaren MSIL-Decompiler ist es sehr einfach möglich, aus einer EXE-Datei den Quellcode zu generieren. Wenn überhaupt, so ist es nur mit hohem Aufwand möglich, dass Sie als Programmierer Ihr Know-how vor der Konkurrenz schützen können.

Die Common Language Runtime (CLR)

Die Laufzeitumgebung bzw. *Common Language Runtime* (CLR) ist die Umgebung, in welcher .NET-Programme auf dem Zielrechner ausgeführt werden, sie muss auf einem Computer nur einmal installiert sein, und schon laufen sämtliche .NET-Anwendungen, egal ob sie in C#, VB.NET oder F# programmiert wurden. Die CLR zeichnet für die Ausführung der Anwendungen verantwortlich und kooperiert auf Basis des CTS mit der MSIL.

Mit ihren Fähigkeiten bildet die *Common Language Runtime* (CLR) gewissermaßen den Kern von .NET. Den Code, der von der CLR ausgeführt wird, bezeichnet man auch als verwalteten bzw. *Managed Code*.

Die CLR ist innerhalb des .NET-Frameworks nicht nur für das Ausführen von verwaltetem Code zuständig, der Aufgabenbereich der CLR ist weitaus umfangreicher und umfasst zahlreiche Dienste, die als Bindeglied zwischen dem verwalteten MSIL-Code und dem Betriebssystem des Rechners die Anforderungen des .NET-Frameworks sicherstellen, wie z.B.

- ClassLoader
- Just-in-Time(JIT)-Compiler
- ExceptionManager
- Code Manager
- Security Engine
- Debug Machine
- Thread Service
- COM-Marshaller

Die Verwendung der sprachneutralen MSIL erlaubt die Nutzung des CTS und der Basisklassen für alle .NET-Sprachen gleichermaßen. Einziger hardwareabhängiger Bestandteil des .NET-Frameworks ist der Just-in-Time Compiler. Deshalb kann der MSIL-Code im Prinzip frei zwischen allen Plattformen bzw. Geräten, für die ein .NET Framework existiert, ausgetauscht werden.

Namespaces ersetzen Registry

Alle Typen des .NET-Frameworks werden in so genannten Namensräumen (Namespaces) zusammengefasst. Unabhängig von irgendeiner Klassenhierarchie wird jede Klasse einem bestimmten Anwendungsgebiet zugeordnet.

Die folgende Tabelle zeigt beispielhaft einige wichtige Namespaces für die Basisklassen des .NET-Frameworks:

Namespace	... enthält Klassen für ...
System.Windows.Forms	... Windows-basierte Anwendungen
System.Collections	... Objekt-Arrays
System.Drawing	... die Grafikprogrammierung
System.Data	... den ADO-Datenbankzugriff
System.Web	... die HTTP-Webprogrammierung
System.IO	... Ein- und Ausgabeoperationen

Mit den Namespaces hat auch der Ärger mit der Registrierung von (COM-)Komponenten bei Versionskonflikten sein Ende gefunden, denn eine unter .NET geschriebene Komponente wird von

der .NET-Runtime nicht mehr über die ProgID der Klasse mit Hilfe der Registry lokalisiert, sondern über einen in der Runtime enthaltenen Mechanismus, welcher einen Namespace einer angeforderten Komponente sowie deren Version für das Heranziehen der "richtigen" Komponente verwendet.

Assemblierungen

Unter einer Assemblierung (*Assembly*) versteht man eine Basiseinheit für die Verwaltung von Managed Code und für das Verteilen von Anwendungen, sie kann sowohl aus einer einzelnen als auch aus mehreren Dateien (Modulen) bestehen. Eine solche Datei (*.dll* oder *.exe*) enthält MSIL-Code (kompilierter Zwischencode).

Die Klassenverwaltung in Form von selbst beschreibenden Assemblies vermeidet Versionskonflikte von Komponenten und ermöglicht vor allem dynamische Programminstallationen aus dem Internet. Statt der bei einer klassischen Installation bisher erforderlichen Einträge in die Windows-Registry genügt nunmehr einfaches Kopieren der Anwendung.

Normalerweise müssen Sie die Assemblierungen referenzieren, in welchen die von Ihrem Programm verwendeten Typen bzw. Klassen enthalten sind. Eine Ausnahme ist die Assemblierung *mscorlib.dll*, welche die Basistypen des .NET Frameworks in verschiedenen Namensräumen umfasst (siehe obige Tabelle).

Zugriff auf COM-Komponenten

Verweise auf COM-DLLs werden so eingebunden, dass sie zur Entwurfszeit quasi wie .NET-Komponenten behandelt werden können.

Über das Menü *Projekt|Verweis hinzufügen...* und Auswahl des "COM"-Registers erreichen Sie die Liste der verfügbaren COM-Bibliotheken. Nachdem Sie die gewünschte Bibliothek selektiert haben, können Sie die COM-Komponente wie gewohnt ansprechen.

HINWEIS: Wenn Sie COM-Objekte, wie z.B. alte ADO-Bibliotheken, in Ihre .NET-Projekte einbinden wollen, müssen Sie auf viele Vorteile von .NET verzichten. Durch die zusätzliche Interoperabilitätsschicht sinkt die Performance meist deutlich.

Metadaten und Reflexion

Das .NET-Framework stellt im *System.Reflection*-Namespace einige Klassen bereit, die es erlauben, die Metadaten (Beschreibung bzw. Strukturinformationen) einer Assembly zur Laufzeit auszuwerten, womit z.B. eine Untersuchung aller dort enthaltenen Typen oder Methoden möglich ist.

Die Beschreibung durch die .NET-Metadaten ist allerdings wesentlich umfassender als es in den gewohnten COM-Typbibliotheken üblich war. Außerdem werden die Metadaten direkt in der Assembly untergebracht, die dadurch selbstbeschreibend wird und z.B. auf Registry-Einträge verzichten kann. Metadaten können daher nicht versehentlich verloren gehen oder mit einer falschen Dateiversion kombiniert werden.

HINWEIS: Es gibt unter .NET nur noch eine einzige Stelle, an der sowohl der Programmcode als auch seine Beschreibung gespeichert wird!

Metadaten ermöglichen es, zur Laufzeit festzustellen, welche Typen benutzt und welche Methoden aufgerufen werden. Daher kann .NET die Umgebung an die Anwendung anpassen, sodass diese effizienter arbeitet.

Der Mechanismus zur Abfrage der Metadaten wird Reflexion (*Reflection*) genannt. Das .NET-Framework bietet dazu eine ganze Reihe von Methoden an, mit denen jede Anwendung – nicht nur die CLR – die Metadaten von anderen Anwendungen abfragen kann.

Auch Entwicklungswerkzeuge wie Microsoft Visual Studio verwenden die Reflexion, um z.B. den Mechanismus der IntelliSense zu implementieren. Sobald Sie einen Methodennamen eintippen, zeigt die IntelliSense eine Liste mit den Parametern der Methode an oder mit allen Elementen eines bestimmten Typs.

Weitere nützliche Werkzeuge, die auf der Basis von Reflexionsmethoden arbeiten, sind der IL-Disassembler (ILDASM) des .NET Frameworks oder ILSpy.

HINWEIS: Eine besondere Bedeutung hat Reflexion im Zusammenhang mit dem Auswerten von Attributen zur Laufzeit (siehe folgender Abschnitt).

Attribute

Wer sich noch an die älteren objektorientierten Sprachen (VB 6, Delphi 7, ...) erinnert, der kennt Attribute als Variablen, die zu einem Objekt gehören und damit seinen Zustand beschreiben.

Unter .NET haben Attribute eine grundsätzlich andere Bedeutung:

HINWEIS: .NET-Attribute stellen einen Mechanismus dar, mit welchem man Typen und Elemente einer Klasse schon beim Entwurf kommentieren und mit Informationen versorgen kann, die sich zur Laufzeit mittels Reflexion abfragen lassen.

Auf diese Weise können Sie eigenständige selbstbeschreibende Komponenten entwickeln, ohne die erforderlichen Infos separat in Ressourcendateien oder Konstanten unterbringen zu müssen. So erhalten Sie mobilere Komponenten mit besserer Wartbarkeit und Erweiterbarkeit.

Man kann Attribute auch mit "Anmerkungen" vergleichen, die man einzelnen Quellcode-Elementen, wie Klassen oder Methoden, "anheftet". Solche Attribute gibt es eigentlich in jeder Programmiersprache, sie regeln z.B. die Sichtbarkeit eines bestimmten Datentyps. Allerdings waren diese Fähigkeiten bislang fest in den Compiler integriert, während sie unter .NET nunmehr direkt im Quellcode zugänglich sind. Das heißt, dass .NET-Attribute typsichere, erweiterbare Metadaten sind, die zur Laufzeit von der CLR (oder von beliebigen .NET-Anwendungen) ausgewertet werden können.

Mit Attributen können Sie Design-Informationen definieren (z.B. zur Dokumentation), Laufzeit-Infos (z.B. Namen einer Datenbankspalte für ein Feld) oder sogar Verhaltensvorschriften für

die Laufzeit (z.B. ob ein gegebenes Feld an einer Transaktion teilnehmen darf). Die Möglichkeiten sind quasi unbegrenzt.

Wenn Ihre Anwendung beispielsweise einen Teil der erforderlichen Informationen in der Registry abspeichert, muss bereits beim Entwurf festgelegt werden, wo die Registrierschlüssel abzulegen sind. Solche Informationen werden üblicherweise in Konstanten oder in einer Ressourcendatei untergebracht oder sogar fest in die Aufrufe der entsprechenden Registrierfunktionen eingebaut. Wesentliche Bestandteile der Klasse werden also von der übrigen Klassendefinition abgetrennt. Der Attribute-Mechanismus macht damit Schluss, denn er erlaubt es, derlei Informationen direkt an die Klassenelemente "anzuheften", so dass letztendlich eine sich vollständig selbst beschreibende Komponente vorliegt.

Serialisierung

Fester Bestandteil des .NET-Frameworks ist auch ein Mechanismus zur Serialisierung von Objekten. Unter Serialisierung versteht man das Umwandeln einer Objektinstanz in sequenzielle Daten, d.h. in binäre oder XML-Daten oder in eine SOAP-Nachricht mit dem Ziel, die Objekte als Datei permanent zu speichern oder über Netzwerke zu verschicken.

Auf umgekehrtem Weg rekonstruiert die Deserialisierung aus den Daten wieder die ursprüngliche Objektinstanz.

Das .NET-Framework unterstützt zwei verschiedene Serialisierungsmechanismen:

- Die *Shallow-Serialisierung* mit der Klasse *System.Xml.Serialization.XmlSerializer*.
- Die *Deep-Serialisierung* mit den Klassen *BinaryFormatter* und *SoapFormatter* aus dem *System.Runtime.Serialization*-Namespace.

Aufgrund ihrer Einschränkungen (geschützte und private Objektfelder bleiben unberücksichtigt) ist die Shallow-Serialisierung für uns weniger interessant. Hingegen werden bei der Deep-Serialisierung alle Felder berücksichtigt, Bedingung ist lediglich die Kennzeichnung der Klasse mit dem Attribut *[Serializable]*.

Anwendungsgebiete der Serialisierung finden sich bei ASP.NET, ADO.NET, XML etc.

Multithreading

Multithreading ermöglicht es einer Anwendung, ihre Aktivitäten so aufzuteilen, dass diese unabhängig voneinander ausgeführt werden können, bei gleichzeitig besserer Auslastung der Prozessorzeit. Allgemein sind Threads keine Besonderheit von .NET, sondern auch in anderen Programmierumgebungen durchaus üblich.

Unter .NET laufen Threads in einem Umfeld, das Anwendungsdomäne genannt wird, Erstellung und Einsatz erfolgen mit Hilfe der Klasse *System.Threading.Thread*.

Nicht in jedem Fall ist die Aufnahme zusätzlicher Threads die beste Lösung, da man sich dadurch auch zusätzliche Probleme einhandeln kann. So ist beim Umgang mit mehreren Threads die Threadsicherheit von größter Bedeutung, d.h., aus Sicht der Threads müssen die Objekte stets in

einem gültigen Zustand vorliegen und das auch dann, wenn sie von mehreren Threads gleichzeitig benutzt werden.

Objektorientierte Programmierung

Last, but not least, wollen wir am Ende unseres Rundflugs über die .NET-Higlights noch einmal auf das allem zugrunde liegende OOP-Konzept verweisen, denn .NET ist komplett objektorientiert aufgebaut – unabhängig von der verwendeten Sprache oder der Zielumgebung, für die programmiert wird (Windows- oder Web-Anwendung).

Jeder .NET-Code ist innerhalb einer Klasse verborgen, und sogar einfache Variablen sind zu Objekten mutiert, die Eigenschaften und Methoden bereitstellen. Es macht deshalb wenig Sinn, mit der Einführung in die Sprache C# fortzufahren ohne sich vorher mit dem Konzept der OOP vertraut gemacht zu haben (siehe Kapitel 3).

1.6 Wichtige Neuigkeiten in Visual Studio 2012

All unseren Lesern, die bereits mit der Vorgängerversion (Visual Studio 2010, C# 4.x) gearbeitet haben, sind wir einen kleinen Überblick über die wichtigsten Neuerungen der Version 2012 (bzw. C# 5.0) schuldig.

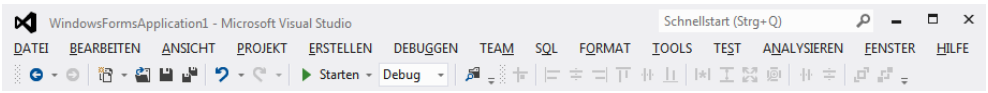
1.6.1 Die neue Visual Studio 2012 Entwicklungsumgebung

Das User Interface von Visual Studio 2012 wurde, aus welchen Gründen auch immer, deutlich umgestaltet und ist – unter weitgehendem Verzicht auf 3D-Effekte – vornehmlich in tristes Grau gehüllt[1]. Der Umsteiger wird einige Zeit brauchen, um altbekannte Funktionen an anderer Stelle wiederzufinden.

Neues Outfit der Toolbar

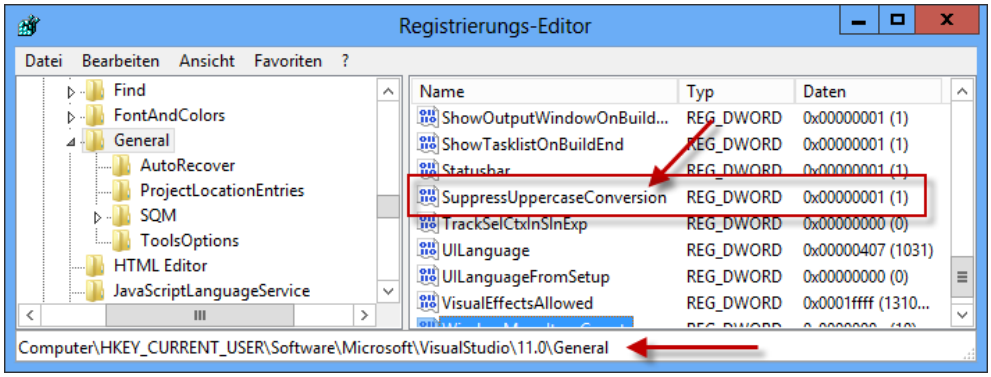
Die neue Schnellstart-Box (oben rechts) soll die die bequeme Suche nach momentan verfügbaren Befehlen und deren Auswahl in einer Dropdown-Liste ermöglichen.

Sehr gewöhnungsbedürftig ist die komplette Großschreibung der Menü-Oberpunkte:

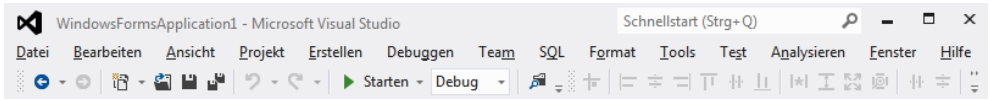


Wem dieser aufdringliche Stil nicht gefällt, für den gibt es eine Lösung: Rufen Sie die Registry auf (*regedit* im Suchfeld des Windows-Startmenüs eingeben). Unter dem Schlüssel *HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\11.0\General* erzeugen Sie einen neuen DWORD-Eintrag mit dem Namen *SuppressUppercaseConversion* und weisen diesem den Wert 1 zu.

[1] Die Alternative, die Sie unter *Tools|Optionen* wählen können, wäre eine gruselige schwarze Bedienoberfläche.

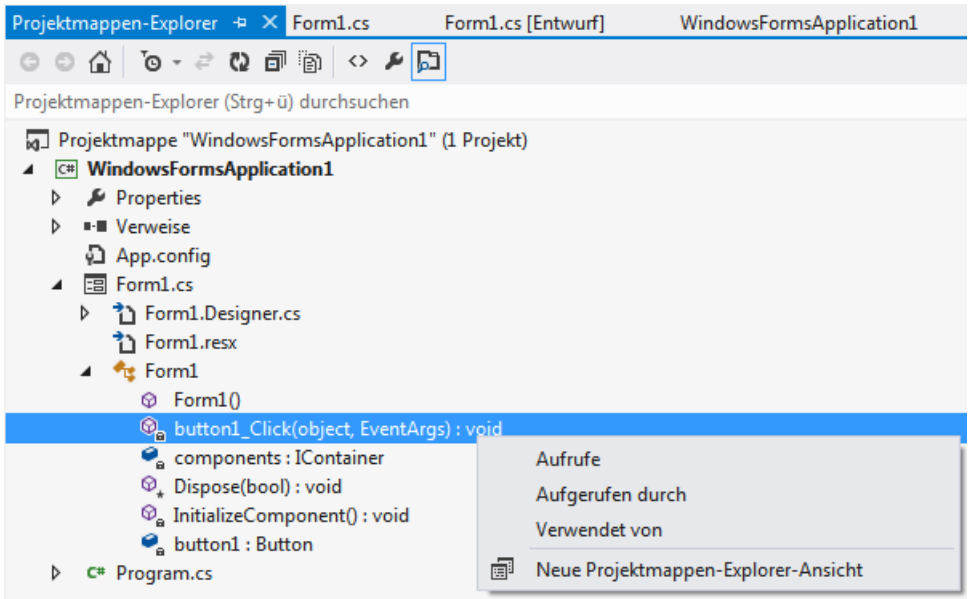


Nach dem Neustart von Visual Studio werden Sie ab jetzt von einem viel freundlicheren Menü begrüßt:



Veränderter Projektmappen-Explorer

Der Projektmappen-Explorer ist zum "Mädchen für alles" mutiert und unterstützt jetzt auch die Navigation durch das Objektmodell, Volltextsuche und mehr. Zum Beispiel können Sie eine *.cs*-Datei expandieren, um die Klassen innerhalb der Datei zu betrachten, dann die Klasse weiter expandieren, um ihre Mitglieder und deren Aufrufhierarchien zu untersuchen (siehe Abbildung).



Registerkartenverwaltung der geöffneten Dateien

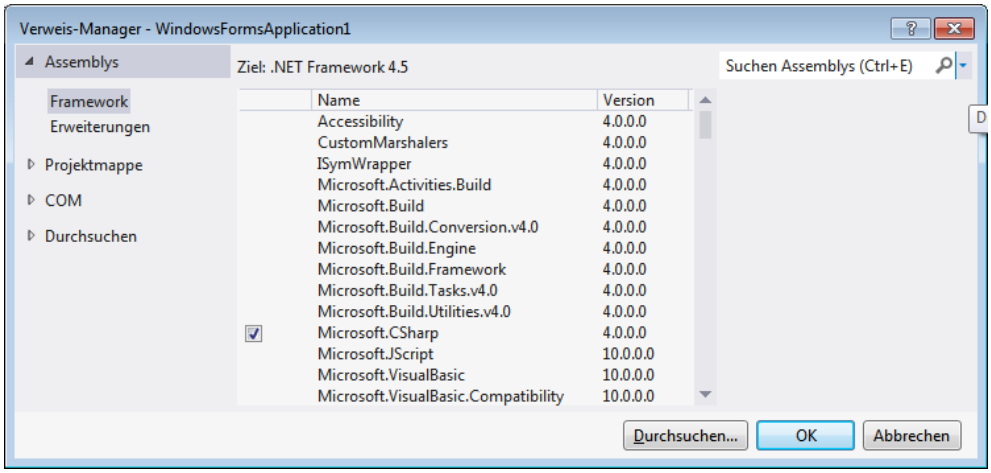
Hier sollten Sie sich an eine interessante Neuigkeit gewöhnen: Haben Sie bisher eine Datei per Doppelklick im Projektmappen-Explorer geöffnet, so gibt es nunmehr die zusätzliche Möglichkeit, die Datei per Einfachklick als Vorschau temporär zu öffnen.

Suchen überall

In Visual Studio 2012 wird jetzt die (mehr oder weniger sinnvolle) Suche nach zahlreichen Features an zahlreichen Orten fast bis zur Perversion ausgeweitet: Projektmappen-Explorer, Verweis hinzufügen, Integrierte Schnellsuche, Test Explorer, Fehlerliste, Parallel Watch, Werkzeugkasten, Team Foundation Server (TFS) u.a.

Neuer Verweis Manager-Dialog

Der Dialog *Verweis-Manager* wurde bereits in den Power Tools für Visual Studio 2010 erneuert, jetzt findet man ihn fest integriert in Visual Studio 2012. Der sich öffnende Verweis-Manager bietet eine Übersicht über .NET-Komponenten im Global Assembly Cache und in den in der Registry gespeicherten Suchpfaden, zu Projekten in der gleichen Projektmappe und COM-Komponenten. Enthalten ist auch ein Browser zur Suche nach Assemblies.



Projekt-Kompatibilität

Visual Studio 2012 enthält jetzt die Projekt-Kompatibilität als spezielles Feature, sodass ein Projekt-Upgrade als gemeinsamer Schritt im Team nicht erforderlich ist. Projektdateien, die unter Visual Studio 2010 erzeugt wurden, bleiben auch nach ihrem Öffnen in Visual Studio 2012 unverändert. Wenn also ein Entwickler im Team ein Visual Studio 2010 Projekt in Visual Studio 2012 öffnet und den gemeinsamen Code ändert, können andere Entwickler dasselbe Projekt unter Visual Studio 2010 SP1 öffnen. Umgekehrt können auch Entwickler für Visual Studio 2012-Projekte daran gemeinsam mit anderen Entwicklern arbeiten, die Visual Studio 2010 SP1 verwenden.

HINWEIS: Die Projekt-Kompatibilität in Visual Studio 2012 funktioniert nur mit Visual Studio 2010 SP1, alternativ werden Sie aufgefordert das Projekt zu konvertieren.

Neue Projekttypen

Die meisten neuen Projekttypen gibt es bei den neuen Windows Store-Anwendungen und bei JavaScript, sie sind allerdings nur verfügbar, wenn Visual Studio 2012 auf einem Windows 8-Computer läuft. Diese Applikationen benötigen die Windows Runtime (WinRT) und können in C#, Visual Basic, C++ und JavaScript entwickelt werden. Achten Sie bei den einzelnen Projekttypen immer auch auf die jeweils unterstützte Framework-Version:

HINWEIS: Visual Studio 2012 kann bis zurück zur Version 2.0 kompilieren. Bei älteren Versionen fehlen dann allerdings viele Projekt-Vorlagen.

Multi-Monitor -Unterstützung

Visual Studio 2012 hat jetzt einen stark verbesserten Multi-Monitor-Support. Auf elegante Weise ist es möglich, die IDE auf mehrere Monitore zu verteilen.

Zusätzliche Tools und Features

Hervorzuheben ist die Integration von Expression Blend in die Visual Studio 2012 IDE. Auch Visual Studio LightSwitch, die Entwicklungsumgebung für das Rapid Application Development (RAD), und das Application Lifecycle Management (ALM) gehören jetzt dazu.

1.6.2 Neuheiten im .NET Framework 4.5

Auch hier wollen wir nur die unserer Meinung nach wichtigsten Features hervorheben:

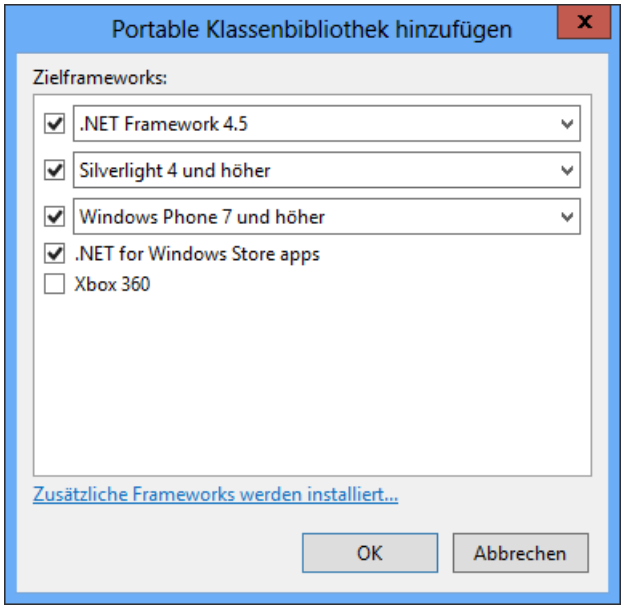
WinRT-Anwendungen

Unter Windows 8 kann eine Teilmenge von .NET Framework 4.5 zum Erstellen von Windows-Apps im WinRT-Stil verwendet werden (siehe Kapitel 17 bis 21).

Mit dem *Resource File Generator*-Tool (*Resgen.exe*) können Sie aus einer RESOURCES-Datei, die in einer .NET Framework-Assembly eingebettet ist, eine RESW-Datei für Windows-Apps im WinRT-Stil erstellen.

Portable Klassenbibliotheken

Ein *Portable Klassenbibliothek*-Projekt erlaubt das Erstellen von verwalteten Assemblies für mehrere .NET-Framework-Plattformen. Nachdem Sie sich für die Zielplattform (Windows Phone, .NET für Windows Store-Apps) entschieden haben, werden die verfügbaren Typen und Mitglieder automatisch auf diese Plattformen beschränkt.



Parallele Computervorgänge

Das Framework 4.5 enthält mehrere neue Funktionen und Erweiterungen für parallele Berechnungen. Dazu gehören die verbesserte Unterstützung für asynchrone Programmierung, die optimierte Unterstützung für paralleles Debuggen und eine neue Datenflussbibliothek.

Internet

Hinzugekommen sind einige neue Funktionen für ASP.NET 4.5: Unterstützung für neue HTML5-Formulartypen, für das asynchrone Lesen und Schreiben von HTTP-Anforderungen und -Antworten, für asynchrone Module, für die E-Mail-Adressen-Internationalisierung (EAI) und für das WebSockets-Protokoll.

WPF

Mit dem *ibbon*-Steuerelement können Sie eine Menüband-Benutzeroberfläche programmieren, die ein Anwendungsmenü und eine Symbolleiste für den Schnellzugriff enthält.

Die synchrone und asynchrone Datenvalidierung wird durch eine neue *INotifyDataErrorInfo*-Schnittstelle unterstützt.

Auch die Datenbindung an statische Eigenschaften und an benutzerdefinierte Typen, die die *ICustomTypeProvider*-Schnittstelle implementieren, ist möglich geworden.

WCF

Hervorzuheben ist hier vor allem die Unterstützung für die Contract-First-Entwicklung sowie für asynchrones Streaming.

1.6.3 C# 5.0 – Sprache und Compiler

Wie die folgende Tabelle zeigt, sind die Neuerungen im Vergleich zu denen der Vorgängerversionen relativ bescheiden.

Version	IDE	Wichtigste Neuerungen					
C# 1.0	VS 2002	Managed Code					
C# 2.0	VS 2005	Generics	Anonyme Methoden	Nullable Types			
C# 3.0	VS 2008	Lambda Ausdrücke	Erweiterungs-methoden	Expression Tree	Anonyme Typen	LINQ	Typinferenz (var)
C# 4.0	VS 2010	Late Binding (dynamisch)	Benannte Argumente	Optionale Parameter	Mehr COM-Support	Parallele Programmierung (TPL) PLINQ	
C# 5.0	VS 2012	Asynchrone Features	Caller Information				

Asynchrone Methoden

Im Zusammenhang mit der asynchronen Programmierung wurden zwei neue Schlüsselwörter eingeführt: der *async*-Modifizierer und der *await*-Operator. Eine mit *async* markierte Methode heißt "asynchrone Methode". Es ergeben sich dadurch teilweise erhebliche Vereinfachungen für den Programmierer (siehe Abschnitt 9.9).

Caller Information

Dieses neue Feature kann hilfreich sein beim Debugging und beim Entwickeln von Diagnose-Tools. So kann doppelter Code vermieden werden, wie zum Beispiel beim Logging und Tracing (siehe Abschnitt 11.3).

1.7 Praxisbeispiele

Bereits im Abschnitt 1.3.3 hatten wir Ihnen die vier Etappen der Programmentwicklung in Visual Studio ganz allgemein erklärt. Jetzt wollen wir Nägel mit Köpfen machen und diese Schritte anhand von zwei Beispielen (ein ganz einfaches und ein etwas anspruchsvolleres) praktisch nachvollziehen.

Für diese kleinen Applikationen sind nicht die geringsten Programmierkenntnisse erforderlich, es geht vielmehr darum, ein erstes Gefühl für die Anwendungsentwicklung unter Visual Studio 2012 zu gewinnen.

1.7.1 Windows Forms-Anwendung für Einsteiger

Die bescheidene Funktionalität beschränkt sich auf ein Fensterchen mit einer Schaltfläche, über welche per Mausklick die Beschriftung der Titelleiste in "Hallo C#-Freunde" geändert werden kann. Das Beispiel demonstriert, mit welch geringem Aufwand man in Visual Studio eigene

Windows-Anwendungen erstellen kann. Der damit ausgelöste Aha-Effekt wird Sie sicher ausreichend motivieren, manche Durststrecken der nächsten Kapitel zu überstehen.

1. Etappe: Visueller Entwurf der Bedienoberfläche

Der Programmstart von *Microsoft Visual Studio 2012* erfolgt entweder über das Windows-Startmenü oder noch schneller über eine Desktop- bzw. Taskleisten-Verknüpfung.

Auf der Startseite klicken Sie den Link *Neues Projekt...*. Im sich daraufhin öffnenden Dialogfenster *Neues Projekt* wählen Sie links in der Baumstruktur unter *Vorlagen* zunächst *Visual C#* und *Windows* aus (siehe Abbildung Seite 69).

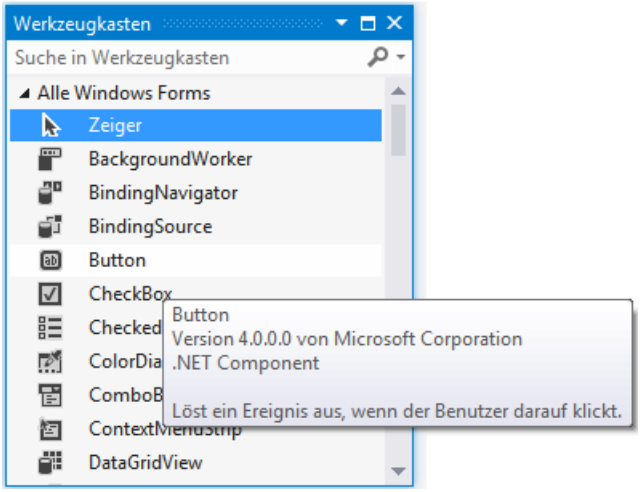
Im Mittelteil klicken Sie auf *Windows Forms-Anwendung* (ganz oben in der Liste). Nehmen Sie im unteren Teil die folgenden Einträge vor bzw. belassen es bei den Standardvorgaben:

Name: *WindowsFormsApplication1*
Ort: z.B.: *C:\CS\Beispiele*
Projektmappenname: *WindowsFormsApplication1*

HINWEIS: Im Dialogfenster rechts oben sehen Sie, dass Sie mit Visual Studio 2012 sowohl Projekte für das .NET Framework 4.5 als auch für die Vorgängerversionen 4, 3.5, 3.0 und 2.0 entwickeln können. Entsprechend der eingestellten Version ändert sich auch das Vorlagen-Angebot.

Nach dem "OK" dauert es ein kleines Weilchen, bis die Entwicklungsumgebung mit dem Startformular *Form1* erscheint. Darauf platzieren Sie ein Steuerelement vom Typ *Button*. Die dazu notwendige Vorgehensweise unterscheidet sich kaum von der bei einem normalen Zeichenprogramm.

Klicken Sie im Menü *Ansicht* auf den Eintrag *Werkzeugkasten* und wählen Sie dann einfach das gewünschte Steuerelement aus:



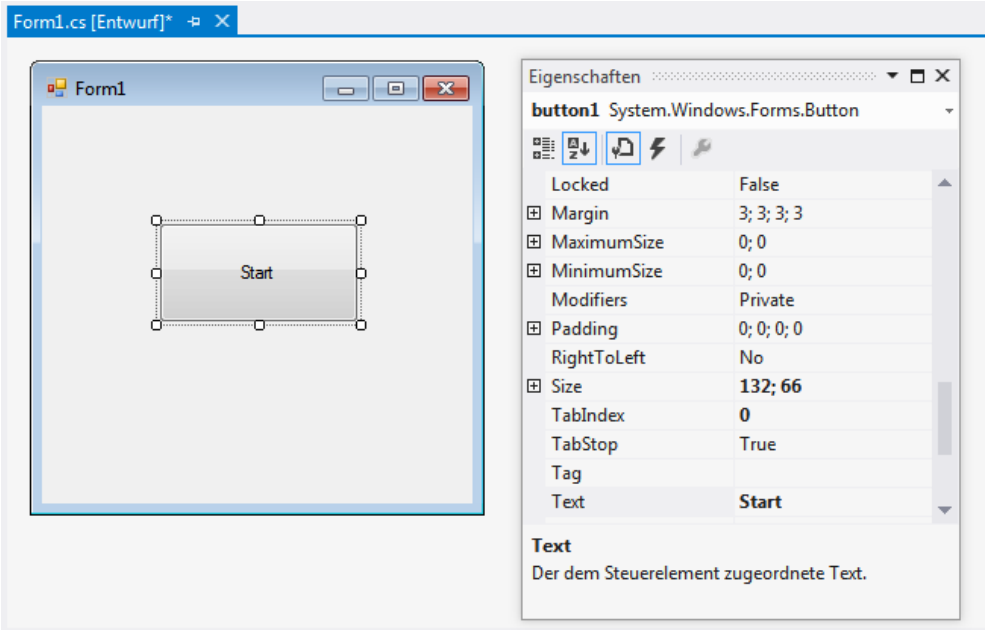
Ein schneller Doppelklick befördert das Steuerelement direkt auf das Formular. Sie können aber auch den gewünschten *Button* in gewohnter Windows-Manier auf das Formular ziehen, dort absetzen und auf die gewünschte Größe zoomen.

2. Etappe: Zuweisen der Objekteigenschaften

Der *Button* trägt noch seine standardmäßige Beschriftung *button1*. Um diese in "Start" zu ändern, muss die *Text*-Eigenschaft geändert werden. Markieren Sie dazu das Steuerelement mit der Maus und rufen Sie mit F4 (bzw. über das Menü *Ansicht*) das Eigenschaftenfenster auf. Ändern Sie im Eigenschaften-Fenster die *Text*-Eigenschaft von ihrem Standardwert "button1" in "Start".

Verwechseln Sie die *Text*-Eigenschaft nicht mit der *Name*-Eigenschaft. Immer wenn Sie ein neues Steuerelement platzieren, setzt Visual Studio standardmäßig den Wert von *Text* zunächst auf den von *Name*.

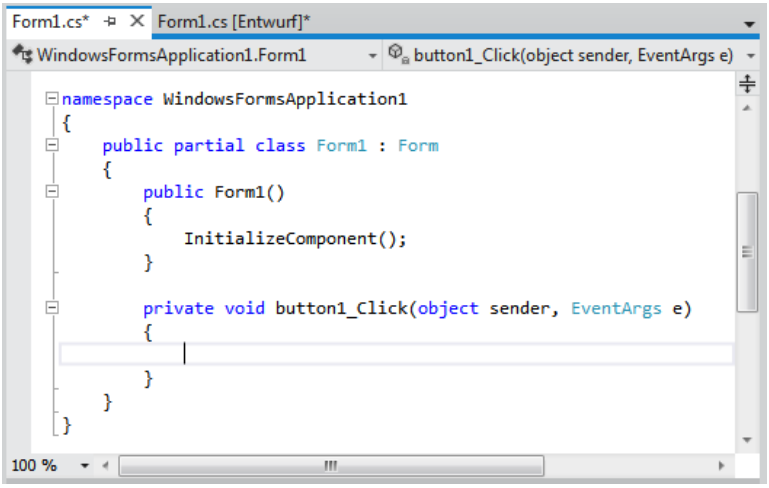
Es dürfte Ihnen nun auch keine Schwierigkeiten bereiten, über die *Font*-Eigenschaft von *button1* auch noch die Schriftgröße etc. zu ändern.



3. Etappe: Verknüpfen der Objekte mit Ereignissen

Klicken Sie doppelt auf die Komponente *button1*, so öffnet sich das Code-Fenster. Richten Sie nun Ihr Augenmerk auf die Schreibmarke, welche im vorgefertigten Rahmencode innerhalb der *Click*-Ereignisbehandlungsroutine (Eventhandler) blinkt. Hier tragen Sie Ihren C#-Code ein, der festlegt, **was** passieren soll, wenn zur Programmlaufzeit (also nicht jetzt zur Entwurfszeit!) der Anwender auf diese Schaltfläche klickt.

In unserem Fall wollen wir erreichen, dass sich die Beschriftung der Titelleiste des Formulars ändert. Das bedeutet, dass wir die *Text*-Eigenschaft des Objekts *Form1*, dessen Standardwert bislang ebenfalls "Form1" lautete, neu zuweisen müssen. Diesmal aber tun wir das nicht im Eigenschaftenfenster, sondern per C#-Code.



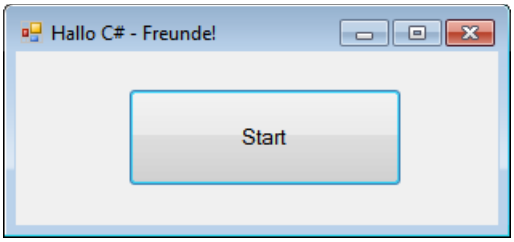
Fügen Sie die fett hervorgehobene Zeile in den bereits vorhandenen Rahmencode ein:

```
private void button1_Click(object sender, EventArgs e)
{
    this.Text = "Hallo C# - Freunde!";
}
```

4. Etappe: Programm kompilieren und testen

Kompilieren Sie das Programm durch Klicken auf das kleine Dreieck in der Symbolleiste (bzw. Menü *Debuggen|Debugging starten* oder *F5*). Sie befinden sich nun im Ausführungsmodus. Ihr Programm "lebt" jetzt, denn die Schaltfläche lässt sich klicken, und die Beschriftung der Titelleiste ändert sich tatsächlich.

Das Programm beenden Sie, indem Sie auf das kleine Quadrat in der Symbolleiste klicken (bzw. Menü *Debuggen/Debugging beenden* oder *Umschalt+F5*) oder aber das Formular einfach in altbekannter Windows-Manier schließen.



Gratulation – Sie haben soeben Ihre erste Windows Forms-Anwendung geschrieben!

Bemerkungen

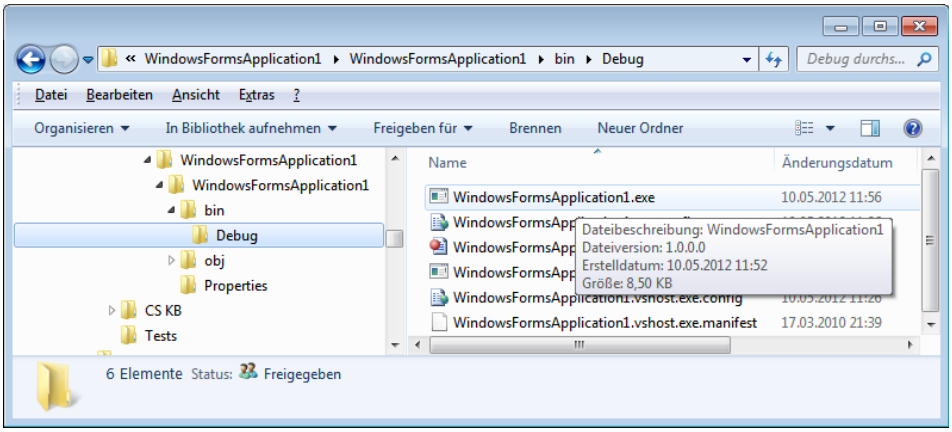
- In diesem Beispiel haben Sie ganz nebenbei auch gelernt, dass man Eigenschaften (Properties) nicht nur zur Entwurfszeit im Eigenschaften-Fenster zuweist, sondern dies auch zur Laufzeit per Code tun kann. Im letzteren Fall wird der Name der Eigenschaft (*Text*) vom zugehörigen Objekt (*this*) durch einen Punkt getrennt.
- Die Properties, die Sie im Eigenschaften-Fenster zuweisen, bezeichnet man auch als *Start-eigenschaften*. Zur Laufzeit können diese – wie im Beispiel für die *Text*-Eigenschaft des Formulars gezeigt – durchaus ihren Wert ändern.
- Das .NET-SDK empfiehlt, dass alle Klassen innerhalb eines Namensraums (*namespace*) definiert werden[1]. Visual Studio verwendet automatisch den Namen Ihres Projekts (wir haben das bei der Standardvorgabe *WindowsFormsApplication1* belassen) als oberste Ebene des Namensraums. Davor wurden über den *using*-Befehl automatisch weitere Namensräume standardmäßig eingebunden. Insgesamt hat also der Quellcode Ihrer ersten Windows-Anwendung folgendes Aussehen, wobei nur die fett hervorgehobene Zeile von Ihnen selbst getippt werden musste:

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

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            this.Text = "Hallo C# - Freunde!";
        }
    }
}
```

HINWEIS: Beachten Sie die durch die {}-Klammern eingegrenzten Gültigkeitsbereiche!

[1] Mehr zum Konzept der Namensräume erfahren Sie im Kapitel 5.

- Die als Ergebnis des Kompilierprozesses generierte *.exe*-Datei finden Sie im Unterverzeichnis *...\WindowsFormsApplikation1\bin\Debug* Ihres Projektordners. Es handelt sich hierbei allerdings **nicht** um eine klassische Exe-Datei, sondern um eine so genannte *Assemblierung* (siehe Abschnitt 1.5). Da die EXE im MSIL-Code vorliegt, ist sie nur in Zusammenarbeit mit dem .NET-Framework lauffähig. Haben Sie die Programmentwicklung erfolgreich abgeschlossen und Visual Studio beendet, so können Sie später jederzeit in dieses Verzeichnis wechseln, um durch Doppelklick auf die Datei *WindowsFormsApplication1.exe* das Programm zu starten.



Direkt im Projektverzeichnis befindet sich die Projektmappendatei *WindowsFormsApplication1.sln*. Wenn Sie auf diese Datei klicken[1], so wird standardmäßig Visual Studio geöffnet und das Programm wird in die Entwicklungsumgebung geladen.

1.7.2 Windows-Anwendung für fortgeschrittene Einsteiger

Diesmal soll es keine Spielerei, sondern ein durchaus nützliches Prögrämmchen sein – die Umrechnung von Euro in Dollar, ein simpler Währungsrechner also. Durch Vergleichen mit der im Abschnitt 1.2 beschriebenen ersten C#-Anwendung dürften auch die Unterschiede der klassischen Konsolentechnik zur visualisierten, objekt- und ereignisorientierten Windows-Programmierung ganz deutlich zu Tage treten.

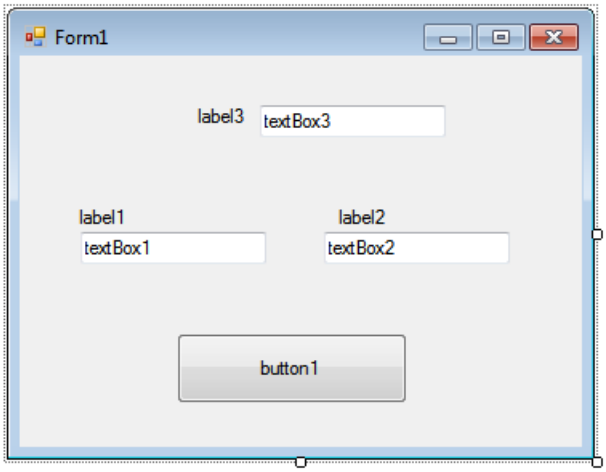
1. Etappe: Visueller Entwurf der Bedienoberfläche

Öffnen Sie ein neues Visual C#-Projekt vom Typ "Windows Forms-Anwendung" und geben Sie ihm den Namen "EuroDollar".

Ziel ist die folgende Bedienoberfläche, die Sie jetzt mühelos im Designer-Fenster erstellen (siehe folgende Abbildung)[2].

[1] Das werden Sie z.B. häufig beim Laden von Beispielprojekten tun.

[2] Der Inhalt der drei Textboxen ist standardmäßig leer und wurde nur hier aus Übersichtlichkeitsgründen mit deren Namen beschriftet.



Sie brauchen außer dem bereits vorhandenen Startformular *Form1* drei *Label* zur Beschriftung, drei *TextBox*en für die Eingabe und einen *Button* zum Beenden des Programms. Für die Namensgebung sorgt Visual Studio automatisch, es sei denn, Sie möchten den Objekten eigene Namen verleihen.

HINWEIS: Konzentrieren Sie sich in der ersten Etappe nur auf Lage und Abmessung der Komponenten, nicht auf deren Beschriftung, da Eigenschaften erst in der nächsten Etappe angepasst werden!

Beim Platzieren und bei der Größenanpassung der Komponenten gehen Sie ähnlich vor, wie Sie es bereits von vektororientierten Zeichenprogrammen (Visio, PowerPoint) gewöhnt sind:

- Im Werkzeugkasten klicken Sie auf das Symbol für die *TextBox*-Komponente. Der Mauszeiger wechselt sein Aussehen.
- Danach bewegen Sie den Mauszeiger zu der Stelle von *Form1*, an welcher sich die linke obere Ecke von *textBox1* befinden soll, drücken die Maustaste nieder und zoomen (bei gedrückt gehaltener Maustaste) die *TextBox* auf ihre endgültige Größe. Analog verfahren Sie mit *textBox2* und *textBox3*.
- Nun klicken Sie im Werkzeugkasten auf das Symbol für die *Label*-Komponente und erzeugen auf die gleiche Weise *label1*, *label2* und *label3*.
- Schließlich bleibt noch *button1*, den Sie am unteren Rand von *Form1* absetzen.

2. Etappe: Zuweisen der Objekteigenschaften

Unser Programm besteht nun aus insgesamt acht Komponenten: einem Formular und sieben Steuerelementen. Alle Eigenschaften haben bereits ihre Standardwerte. Einige davon müssen wir allerdings noch ändern. Dies geschieht mit Hilfe des Eigenschaften-Fensters. Wenn Sie mit der

Maus auf eine Komponente klicken und danach die *F4*-Taste betätigen, erscheint das Eigenschaften-Fenster der Komponente mit der Liste aller zur Entwurfszeit verfügbaren Eigenschaften.

- Beginnen Sie mit *label1*, das die Beschriftung "Euro" tragen soll. Die Beschriftung kann mit der *Text*-Eigenschaft geändert werden. Standardmäßig entspricht diese der *Name*-Property, in unserem Fall also "*label1*". Um das zu ändern, klicken Sie auf das *Label* und tragen anschließend in der Spalte rechts neben dem *Text*-Feld die neue Beschriftung ein (die alte ist vorher "wegzuradieren"). Analog verfahren Sie mit den beiden anderen *Label*s (Beschriftung "Dollar" und "Kurs 1: ").
- Auch *button1* muss natürlich seine neue *Text*-Eigenschaft ("Beenden") erhalten.
- Schließlich klicken Sie auf eine leere Fläche von *Form1*, um anschließend mit *F4* das Eigenschaften-Fenster für das Formular aufzurufen und dessen *Text*-Eigenschaft entsprechend der gewünschten Beschriftung der Titelleiste zu modifizieren.

Die Tabelle gibt eine Zusammenstellung aller Objekteigenschaften, die wir geändert haben:

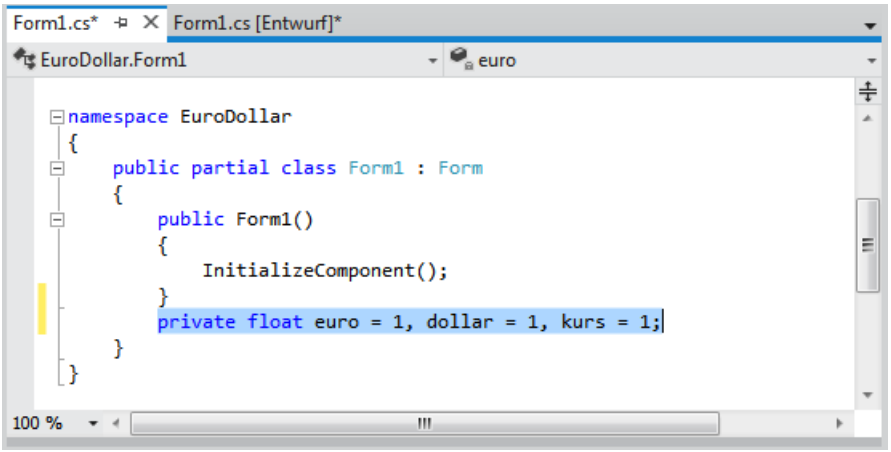
Name des Objekts	Eigenschaft	Neuer Wert
Form1	*Text* *Font.Size*	Währungsrechner *10*
label1	*Text*	Euro
label2	*Text*	Dollar
label3	*Text*	Kurs 1:
textBox1	*TextAlign*	*Right*
textBox2	*TextAlign*	*Right*
textBox3	*TextAlign*	*Center*
button1	*Text*	Beenden

3. Etappe: Verknüpfen der Objekte mit Ereignissen

Während Sie die beiden Vorgängeretappen noch getrost Ihrer Sekretärin überlassen konnten, beginnt jetzt Ihre Hauptarbeit als C#-Programmierer. Wechseln Sie zum Code-Fenster *Form1.cs* (auch mit *F7*, *Ansicht|Code* oder dem Kontextmenü des Formulars möglich). Was Sie erwartet, ist die von Visual Studio vorbereitete Klassendeklaration von *Form1*. Wie Sie sehen, können Sie einzelne Bereiche (Regionen) durch das Plus- bzw. Minus-Symbol am linken Rand auf- bzw. zuklappen.

Zunächst fügen Sie eine Anweisung ein, um drei Variablen des *float*-Datentyps zu deklarieren. Gleichzeitig werden diese Variablen mit dem Wert 1 initialisiert:

```
private float euro=1, dollar=1, kurs=1;
```

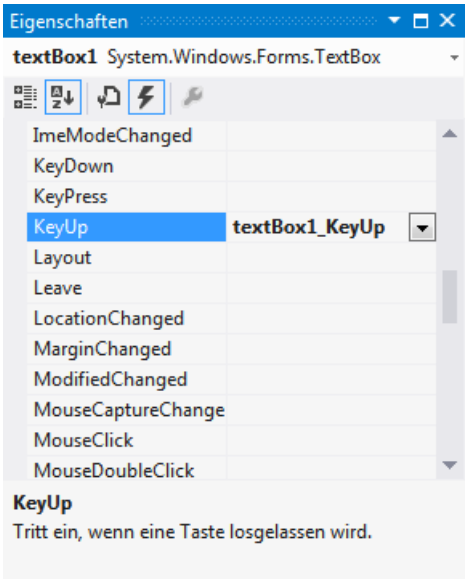


Im Unterschied zur einfachen Konsolenanwendung, bei welcher uns das Programm die Einhaltung einer bestimmten Eingabereihenfolge aufgezwungen hat, soll in unserer Windows Forms-Anwendung die Berechnung immer dann neu gestartet werden, wenn wir bei der Eingabe in eine der drei Textboxen irgendeine Taste losgelassen haben. Wir müssen also für jede der Textboxen einen eigenen Eventhandler für das *KeyUp*-Ereignis schreiben!

Dabei ist eine fast schon rituelle Erstellungsreihenfolge zu beachten, die Sie mit fortschreitender Programmierpraxis sehr bald auch im Schlaf ausführen können:

- **Objekt auswählen**
  Zur Objektauswahl klicken Sie auf das Objekt im Designer-Fenster und öffnen mit *F4* das Eigenschaften-Fenster.
  Klicken Sie im Eigenschaften-Fenster oben auf das ⚡-Symbol, um die Ereignisliste zur Anzeige zu bringen.
- **Ereignis auswählen**
  Zur Ereignisauswahl doppelklicken Sie auf das gewünschte Ereignis. Als Resultat werden automatisch die erste und die letzte Zeile (Rahmencode) des Eventhandlers generiert und im Code-Fenster angezeigt.
- **Ereignisbehandlungen programmieren**
  Füllen Sie den Eventhandler mit den gewünschten C#-Anweisungen aus.

Wir beginnen in unserem Beispiel mit dem *KeyUp*-Eventhandler für *textBox1*, der immer dann ausgeführt wird, wenn der Benutzer den Euro-Betrag ändert. Doppelklicken Sie also auf *KeyUp* und das Codefenster öffnet sich mit dem automatisch erzeugten Rahmencode des Eventhandlers (siehe folgende Abbildung).



Füllen Sie nun den Rahmencode wie folgt aus:

```
private void textBox1_KeyUp(object sender, KeyEventArgs e)
{
    euro = Convert.ToSingle(textBox1.Text);
    dollar = euro * kurs;
    textBox2.Text = dollar.ToString("#,##0.00");
}
```

HINWEIS: Sie müssen nur die Anweisungen innerhalb der geschweiften Klammern selbst einfügen, der übrige Rahmencode wird automatisch erzeugt, wenn Sie die oben erläuterte Erstellungsreihenfolge beachten!

Der Prozedurkopf des Eventhandlers verweist standardmäßig vor dem Unterstrich (_) auf den Namen des Objekts und danach auf das entsprechende Ereignis. Das vorangestellte *private* verdeutlicht, dass auf die Prozedur nur innerhalb des *Form1*-Klasse zugegriffen werden kann.

Auf analoge Weise erzeugen Sie die Eventhandler für die Steuerelemente *textBox2* und *textBox3*, geben aber dann den jeweils geänderten Umrechnungscode ein.

Ändern des Dollar-Betrags:

```
private void textBox2_KeyUp(object sender, KeyEventArgs e)
{
     dollar = Convert.ToSingle(textBox2.Text);
     euro = dollar / kurs;
     textBox1.Text = euro.ToString("#,##0.00");
}
```

Ändern des Umrechnungskurses:

```
private void textBox3_KeyUp(object sender, KeyEventArgs e)
{
     kurs = Convert.ToSingle(textBox3.Text);
     dollar = euro * kurs;
     textBox2.Text = dollar.ToString("#,##0.00");
}
```

HINWEIS: Grübeln Sie jetzt noch nicht über den tieferen Sinn der einzelnen Anweisungen nach, denn dazu haben Sie in den späteren Kapiteln noch genug Zeit!

Damit Sie bereits unmittelbar nach dem Programmstart sinnvolle Werte in den drei Textboxen sehen, ist folgender Eventhandler für das *Load*-Ereignis des *Form1*-Objekts hinzuzufügen:

```
private void Form1_Load(object sender, EventArgs e)
{
     textBox1.Text = euro.ToString();
     textBox2.Text = dollar.ToString();
     textBox3.Text = kurs.ToString();
}
```

Beim Klick auf den *Beenden*-Button soll das Formular entladen werden. Wählen Sie in der Objektauswahlliste des Eigenschaften-Fensters den Eintrag *button1* und anschließend in der Ereignisauswahlliste das *Click*-Event:

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

4. Etappe: Programm kompilieren und testen

Klicken Sie auf den ▶ Starten-Button in der Symbolleiste (oder *F5*-Taste), und im Handumdrehen ist Ihre C#-Windows-Anwendung kompiliert und ausgeführt!

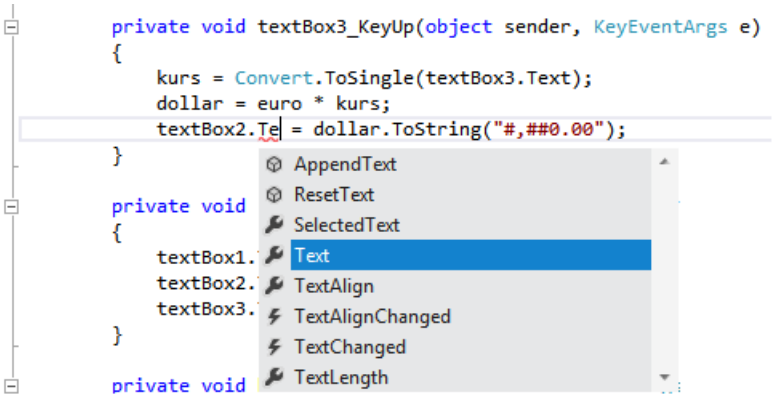
Spielen Sie ruhig ein wenig mit verschiedenen Werten herum, um sich den Unterschied zwischen Konsolen- und Windows-Programmen zu verinnerlichen. Da es keine vorgeschriebene Reihenfolge für die Benutzereingaben mehr gibt, werden die anderen Felder sofort aktualisiert. Eine spezielle "="-Schaltfläche (etwa wie bei einem Taschenrechner) ist deshalb überflüssig.

HINWEIS: Achten Sie darauf, dass Sie als Dezimaltrennzeichen das Komma und nicht den Punkt eingeben. Letzterer dient als Tausender-Separator.

IntelliSense – die hilfreiche Fee

Eines der bemerkenswertesten Features des Code-Editors ist seine so genannte *IntelliSense*, auf die Sie mit Sicherheit bereits beim Eintippen des Quellcodes aufmerksam geworden sind. Sobald Sie den Namen eines Objekts bzw. eines Steuerelements mit einem Punkt abschließen, erscheint wie von Geisterhand eine Liste mit allen Eigenschaften und Methoden des Objekts. Das befreit Sie von dem lästigen Nachschlagen in der Hilfe und bewahrt Sie vor Schreibfehlern.

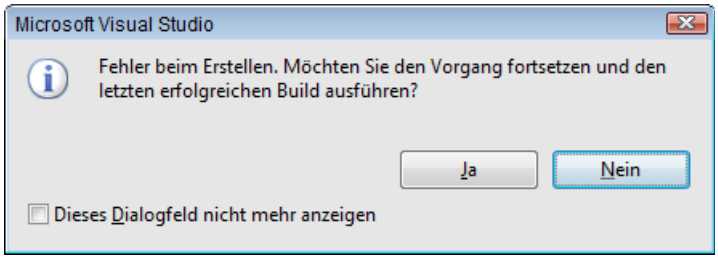
HINWEIS: Wenn Sie das markierte Element übernehmen wollen, brauchen Sie den Namen nicht zu Ende zu schreiben, da die IntelliSense den kompletten Text automatisch ergänzt.



Hilfe ein Fehler!

Bereits beim Schreiben des Quellcodes werden Sie von Visual Studio auf grundsätzliche Syntaxfehler (z.B. ein vergessenes Semikolon) hingewiesen, im Allgemeinen geschieht dies durch Unterstreichen mit einer wellenförmigen (roten) Linie. Wenn Sie mit der Maus auf diese Linie zeigen, erhalten Sie meistens auch noch einen, mehr oder weniger brauchbaren, Hinweis (z.B. "; wird erwartet").

Andere Fehler treten erst beim Kompilieren ans Tageslicht, wobei Sie durch folgende Meldung aufgeschreckt werden:



In der Regel sollten Sie *Nein* klicken, um unverzüglich den (oder die) Übeltäter im Quellcode zu suchen und dingfest zu machen. Das Lokalisieren ist meist sehr einfach, da die fehlerhaften Ausdrücke durch eine Wellenlinie unterschlängelt sind. Auch hier erhalten Sie Hinweise zur Fehlerursache, wenn Sie mit der Maus auf die betreffende Passage zeigen.

HINWEIS: Wenn das Programm sich partout nicht kompilieren lassen will und weit und breit keine Wellenlinie bzw. ein anderer Hinweis auf den Übeltäter in Sicht ist, hilft meist ein Blick in die Fehlerliste am unteren Rand des Hauptfensters (oder Menü *Ansicht|Fehlerliste*).

Auf einen typischen Fehler, der manchen Anfänger zur Verzweiflung bringen kann, soll Sie das folgende Beispiel hinweisen.

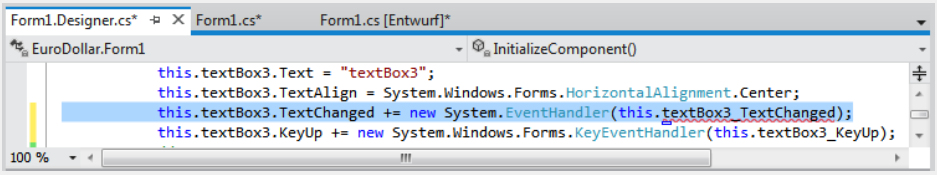
BEISPIEL 1.1: Fehler

Durch einen versehentlichen Doppelklick auf *textBox3* haben Sie im Codefenster unbewusst einen Eventhandler für das *TextChanged*-Ereignis (das ist das Standardereignis für dieses Steuerelement) erzeugt. Sie haben das zwar sofort bemerkt und den Rahmencode des Eventhandlers gleich wieder gelöscht. Trotzdem werden Sie beim anschließenden Versuch, wieder zum Designer umzuschalten, durch folgenden Anblick erschreckt:

Erst ein Blick auf die Fehlerliste bringt Sie auf die richtige Fährte, denn Sie haben zwar den Rahmencode des *textBox3_TextChanged*-Eventhandlers wieder gelöscht, nicht aber seine (von

BEISPIEL 1.1: Fehler

Visual Studio automatisch angelegte) Deklaration. Diese ist nach wie vor vorhanden und verweist (für Sie zunächst unsichtbar) auf den nun nicht mehr vorhandenen Code. Klicken Sie deshalb auf den Link "Gehe zu Code" (oder doppelklicken Sie auf den Eintrag in der Fehlerliste), worauf sich das Codefenster der Datei *Form1.Designer.cs* öffnet[1]:



Der Name des "falschen" Eventhandlers ist mit einer roten Linie unterschlängelt. Entfernen Sie die blau hinterlegte Zeile komplett.

Falls sich anschließend das Designer-Fenster zwar öffnen lässt, auf dem Formular *Form1* aber die Steuerelemente fehlen, so müssen Sie Visual Studio komplett schließen und neu starten.

HINWEIS: Weitere hilfreiche Hinweise zum Debuggen finden Sie im Kapitel 11.

[1] In dieser Datei haben Sie normalerweise nichts zu suchen, da die Einträge von Visual Studio vorgenommen werden.

Kapitel 2

Grundlagen der Sprache C#

In diesem Kapitel wollen wir Ihnen den für den Einstieg wichtigen Sprachkern von C# erklären[1]. Wir zeigen Ihnen, wie Sie Anweisungen schreiben, mit Datentypen umgehen, Schleifen und Verzweigungen einsetzen, Arrays definieren und Funktionen bzw. Prozeduren aufrufen. Mit Rücksicht auf den Einsteiger und um die Übersichtlichkeit nicht zu gefährden, folgen die etwas anspruchsvolleren Bestandteile von C# erst in späteren Kapiteln (objektorientiertes Programmieren in Kapitel 3, fortgeschrittenere Sprachelemente in den Kapiteln 4 und 5).

HINWEIS: Testen Sie möglichst viele der kleinen Codeschnipsel des vorliegenden Kapitels am eigenen PC auf Herz und Nieren. Als Codegerüst kann entweder eine Konsolenanwendung oder aber eine Windows Forms-Anwendung dienen.

2.1 Grundbegriffe

Zur Vorbereitung empfehlen wir ein Rückblättern zum Kapitel 1 (Einführungsbeispiel 1.2, Praxisbeispiele 1.7), wo solch grundlegende Begriffe wie Anweisungen, Klassen, Namensraum und Gültigkeitsbereiche bereits grob erklärt wurden.

2.1.1 Anweisungen

Wir verstehen unter einer Anweisung einen Befehl, der eine bestimmte Aktion ausführt. Wie in jeder anderen Programmiersprache müssen auch die C#-Anweisungen bestimmten Regeln entsprechen, die man in ihrer Gesamtheit als *Syntax*[2] bezeichnet.

Eine der wichtigsten Syntaxregeln kennen Sie bereits aus den Einführungsbeispielen, nämlich dass jede Anweisung mit einem Semikolon abzuschließen ist und dass der Zeilenumbruch dabei keinerlei Rolle spielt.

---

[1] Der Profi, der bereits mit Java, C oder C++ gearbeitet hat, wird dieses Kapitel natürlich mit Siebenmeilenstiefeln durcheilen.

[2] Im Unterschied zur *Syntax* versteht man unter der *Semantik* einer Sprache die Beschreibung dessen, *was* die einzelnen Anweisungen bewirken.

Da C# eine so genannte formatfreie Sprache ist, haben neben dem Zeilenumbruch auch Leerzeichen, Tabulatoren etc. keine Bedeutung, es sei denn, Sie verwenden sie bewusst, um die optische Lesbarkeit Ihres Codes zu verbessern.

Durch gute Strukturierung Ihres Quellcodes, wie z.B. blockweises Einrücken, machen Sie Ihre Programme übersichtlicher und verringern somit die Fehlerquote.

HINWEIS: Die Entwicklungsumgebung von Visual Studio erleichtert Ihnen das blockweise Einrücken unter anderem durch das Menü *Bearbeiten/Erweitert/Zeileneinzug vergrößern* bzw. *verkleinern* oder durch die entsprechenden Schaltflächen der Symbolleiste.

2.1.2 Bezeichner

Für die Namensgebung von Elementen Ihres Programms, wie Variablen, Methoden, Klassen, ... verwenden Sie Bezeichner. Bei der Namensgebung müssen Sie sich an folgende Regeln halten:

- Als Zeichen sind Groß- und Kleinbuchstaben, der Unterstrich "_" sowie die Ziffern 0...9 zulässig.
- Jeder Bezeichner muss mit einem Buchstaben (oder einem Unterstrich) beginnen.
- Als case-sensitive Sprache unterscheidet C# penibel zwischen Groß- und Kleinschreibung.

BEISPIEL 2.1: Zulässige Bezeichner

C#
```
euro
 _radius
zwerg7
```

BEISPIEL 2.2: Unzulässige Bezeichner

C#
```
%Anteil
7Zwerge
Gehalt$
```

Bezüglich der Verwendung von Groß- und Kleinschreibung gibt es zwar keine Verbote, aber folgende Empfehlung:

HINWEIS: Verwenden Sie möglichst keine Bezeichner, die sich lediglich durch die Groß- und Kleinschreibung voneinander unterscheiden!

BEISPIEL 2.3: Beide Bezeichner sollten Sie nicht nebeneinander verwenden:

C#
```
MeineAdresse
meineAdresse
```

2.1.3 Schlüsselwörter

Bei Schlüsselwörtern handelt es sich um vordefinierte reservierte Bezeichner, die den Kern der Sprache C# ausmachen und die im Code-Fenster von Visual Studio normalerweise blau eingefärbt werden. Die folgende (nicht ganz vollständige) Übersicht zeigt die Schlüsselwörter von C#.

abstract	*as*	*async*	*await*	*base*	*bool*
break	*byte*	*case*	*char*	*class*	*const*
continue	*decimal*	*default*	*delegate*	*do*	*double*
else	*enum*	*event*	*explicit*	*extern*	*false*
finally	*fixed*	*float*	*for*	*foreach*	*goto*
if	*implicit*	*in*	*int*	*interface*	*internal*
is	*lock*	*long*	*namespace*	*new*	*null*
object	*operator*	*out*	*override*	*params*	*private*
protected	*public*	*readonly*	*ref*	*return*	*sbyte*
sealed	*short*	*sizeof*	*stackalloc*	*static*	*string*
struct	*switch*	*this*	*throw*	*true*	*try*
typeof	*uint*	*ulong*	*unchecked*	*unsafe*	*ushort*
using	*var*	*virtual*	*void*	*volatile*	*while*

HINWEIS: Schlüsselwörter dürfen Sie **nicht** für selbst definierte Bezeichner verwenden!

Allerdings gibt es zu obigem Hinweis eine gewisse Ausnahme: Wenn Schlüsselwörter ein @ als Präfix enthalten, können sie als Bezeichner im Programm verwendet werden.

BEISPIEL 2.4: Schlüsselwörter mit Präfix

C# *@if* stellt z.B. einen gültigen Bezeichner dar, *if* jedoch nicht, da es sich um ein Schlüsselwort handelt.

2.1.4 Kommentare

Kommentaranweisungen (im Editor standardmäßig grün eingefärbt) dienen als zusätzliche Erläuterungen für den Programmierer, sie sollen die Lesbarkeit des Quellcodes verbessern. Für das Kennzeichnen von Kommentaren können Sie zwei unterschiedliche Verfahren verwenden.

Einzeilige Kommentare

Um eine Zeile (nicht Befehlszeile) als Kommentar zu markieren, leiten Sie diese mit einem doppelten Slash // ein.

BEISPIEL 2.5: Eine Anweisung mit Kommentar

C#
```
private float euro=1, dollar=1, kurs=1;    // Variablendeklaration
```

HINWEIS: Geizen Sie in Ihren Quelltexten nicht mit Kommentaren, damit Sie (oder andere) auch später noch den von Ihnen geschriebenen Code verstehen können!

Mehrzeilige Kommentare

Um einen mehrzeiligen Bereich als Kommentar zu kennzeichnen, wird dieser mit /* und */ eingegrenzt.

BEISPIEL 2.6: Mehrzeiliger Kommentar

C#
```
/* Dieser Kommentar
besteht aus zwei Zeilen */
```

Mehrzeilige Kommentare können Sie auch vorteilhaft beim Testen von Code einsetzen, indem Sie (in der Regel nur vorübergehend) bestimmte Codeabschnitte außer Kraft setzen, d.h. "auskommentieren".

HINWEIS: Die Visual Studio Entwicklungsumgebung erleichtert Ihnen das Auskommentieren von Codeabschnitten mit dem Menü *Bearbeiten/Erweitert/Auswahl kommentieren* (Strg+E, C) bzw. *Auskommentierung der Auswahl aufheben* (Strg+E, U) oder mit den entsprechenden Schaltflächen der Symbolleiste.

2.2 Datentypen, Variablen und Konstanten

Jedes Programm "lebt" in erster Linie von seinen Variablen und Konstanten, die bestimmten Datentypen entsprechen. Es ist daher logisch, dass wir dieses Thema an den Anfang unserer Betrachtungen zur Sprache C# stellen müssen.

2.2.1 Fundamentale Typen

Die folgende Tabelle gibt eine Übersicht der einfachen (fundamentalen) Datentypen, die C# zur Verfügung stellt[1].

Wie Sie der Tabelle entnehmen können, entsprechen alle C#-Datentypen einer Klassendefinition im .NET Framework. Die CLR[2]-Datentypen sind im *System*-Namensraum angeordnet. Bei der Deklaration (siehe unten) ist es egal, welchen der beiden möglichen Typbezeichner Sie angeben.

[1] Auf "anspruchsvollere" Datentypen wie *var* oder *dynamic*, gehen wir erst an späterer Stelle ein.

[2] Die .NET-Laufzeitumgebung (*Common Language Runtime*)

C#-Datentyp	.NET-CLR-Typ	Erläuterung	Länge [Byte][1]
byte	*System.Byte*	positive Ganzzahl zwischen 0 ... 255	1
sbyte	*System.SByte*	vorzeichenbehaftete Ganzzahl zwischen -128 ... 127	1
short	*System.Int16*	kurze Ganzzahl zwischen -2^{15} ... $2^{15}-1$ (-32.768 ... 32.767)	2
ushort	*System.UInt16*	vorzeichenlose Ganzzahl zwischen 0 ... 65.535	2
int	*System.Int32*	Ganzzahl zwischen -2^{31} ... $2^{31}-1$ (-2.147.483.648 ... 2.147.483.647)	4
uint	*System.UInt32*	vorzeichenlose Ganzzahl zwischen 0 ... 4.294.967.295	4
ulong	*System.UInt64*	vorzeichenlose Ganzzahl zwischen 0 ... 18.446.744.073.709.551.615	8
long	*System.Int64*	lange Ganzzahl -2^{63} ... $2^{63}-1$ (-9.223.372.036.854.775.808 ... 9.223.372.036.854.775.807)	8
float	*System.Single*	einfachgenaue Gleitkommazahl mit 7-stelliger Genauigkeit zwischen ca. +/- 3.4E-45 und +/-3.4E+38	4
double	*System.Double*	doppeltgenaue Gleitkommazahl mit 16-stelliger Genauigkeit zwischen ca. +/- 4.9E-324 und +/- 1.8E+308	8
decimal	*System.Decimal*	hochgenaue Gleitkommazahl zwischen 0 ... +/- 79E+27 (ohne Dezimalpunkt) und ca. +/-1.0E-29 ...7.9E+27 (mit Dezimalpunkt)	16
char	*System.Char*	ein beliebiges Unicode-Zeichen	2
bool	*System.Boolean*	Wahrheitswert (*true*, *false*)	2
string	*System.String*	beliebige Unicode-Zeichenfolge mit einer maximalen Länge von ca. 2.000.000.000 Zeichen	2 pro Zeichen plus 10
object	*System.Object*	universeller Datentyp	4

2.2.2 Wertetypen versus Verweistypen

Mit Ausnahme des *string*- und des *object*-Datentyps, die so genannte *Verweistypen* sind, gehören die übrigen Datentypen in obiger Tabelle zu den *Wertetypen*. Das Verständnis des Unterschieds zwischen diesen beiden fundamentalen Gruppen ist enorm wichtig für das tiefere Eindringen in die Sprache C#.

[1] Systemintern sind auch die ersten vier Typen 4-Byte lang.

Wertetypen

Dazu zählen all die einfachen Datentypen wie *byte*, *int*, *double* ..., hinzu kommen später noch andere wie beispielsweise *struct* (siehe 2.6.2) und *DateTime* (siehe Kapitel 4). Beim Abarbeiten des Programms wird für die lokalen Variablen und die übergebenen Parameter Speicherplatz benötigt, der immer dem Stack entnommen wird. Nach Beendigung einer Methode wird der Speicher automatisch an den Stack[1] zurückgegeben.

Verweistypen

Bislang kennen wir nur die Verweistypen *string* und *object*, im Kapitel 4 kommen später noch Datenfelder (Arrays) hinzu. Aber das ist nur die Spitze des Eisbergs, denn in der objektorientierten Programmierung, in welche wir ab Kapitel 3 tiefer einsteigen werden, sind alle Objekte Verweistypen. Das bedeutet, dass auf dem Stack nicht der Wert des Objekts abgespeichert wird, sondern lediglich ein Verweis (Referenz, Zeiger) auf eine Speicheradresse des Heap, wo das eigentliche Objekt gespeichert ist. Das Anlegen des Objekts auf dem Heap wird auch als Instanziierung bezeichnet, in der Regel muss dazu der *new*-Operator verwendet werden[2]. Das Entsorgen des Speichers übernimmt sporadisch der so genannte Garbage Collector. Doch zu all dem kommen wir erst im nachfolgenden OOP-Kapitel.

2.2.3 Benennung von Variablen

Variablen sind benannte Speicherplatzstellen, der Variablenname dient dazu, die Speicheradresse im Programmcode quasi wie über einen Alias anzusprechen.

Zusätzlich zu den unter 2.1.2 aufgeführten Regeln für selbst definierte Bezeichner sollten Sie folgenden Empfehlungen gemäß *Common Language Specification* (CLS) folgen:

- Beginnen Sie den Namen einer Variablen mit einem Kleinbuchstaben.
- Vermeiden Sie Unterstriche (_).
- Falls Bezeichner aus mehreren Wörtern zusammengesetzt sind, so sollten ab dem zweiten Wort alle Wörter mit einem Großbuchstaben beginnen.

BEISPIEL 2.7: Ein Variablenname, der sich aus mehreren Wörtern zusammensetzt.

```
C# meinHaushaltskassenSaldo
```

2.2.4 Deklaration von Variablen

Variablen werden in C# wie folgt deklariert:

SYNTAX: *Datentyp Variablenname*;

[1] Stack und Heap sind bestimmte Bereiche im Arbeitsspeicher jedes Computers.

[2] Der *string*-Datentyp bildet hier eine gewisse Ausnahme (siehe 4.2).

BEISPIEL 2.8: Drei Variablen unterschiedlichen Datentyps werden deklariert.

C#
```
int Anzahl; double Breite;
string nachName;
```

Wollen Sie mehrere Variablen vom gleichen Datentyp deklarieren, so werden diese durch Kommas separiert.

BEISPIEL 2.9: Drei int-Variablen werden deklariert.

C#
```
int i, j, k;
```

Als Datentyp kann man auch den CLR-Typ angeben (siehe obige Tabelle).

BEISPIEL 2.10: Die folgenden drei Deklarationen sind gleichwertig.

C#
```
int i;
System.Int32 i;
Int32 i;
```

Initialisierte Variablen

Zusammen mit der Deklaration können Sie den Variablen auch gleich Anfangswerte zuweisen (im Fachjargon heißt das "initialisieren").

BEISPIEL 2.11: Initialisierte Variablen

C#

Statt

```
int anzahl;
anzahl = 99;
```

können Sie kürzer formulieren:

```
int anzahl = 99;
```

Typinferenz

Dieses in Zusammenhang mit der LINQ-Technologie (siehe Kapitel 6 ab Seite 347) neu eingeführte Sprachmerkmal erlaubt es, dass der Datentyp von Variablen bei der Deklaration vom Compiler automatisch ermittelt wird, ohne dass Sie explizit den Typ angegeben müssen. Als Ersatz für einen konkreten Typ wird das Schlüsselwort *var* verwendet.

HINWEIS: Damit der Compiler den Typ der Variablen feststellen kann, muss eine mit *var* deklarierte Variable unbedingt bei der Deklaration initialisiert werden.

BEISPIEL 2.12: Typinferenz

C#

Die Initialisierung der Variablen *a* wird vom Compiler ausgewertet und der Typ aufgrund des Wertes 35 auf *Integer* festgelegt.

```
var a = 35;
```

Obige Zeile ist semantisch identisch mit folgendem Ausdruck:

```
int a = 35;
```

Der Datentyp wird einmalig bei der ersten Deklaration der Variablen vom Compiler festgelegt und kann danach nicht mehr verändert werden.

BEISPIEL 2.13: Keine Datentypänderung möglich

C#

Da die Variable *b* vom Compiler als *Integer* festgelegt wurde, kann ihr später kein *Double*-Wert zugewiesen werden.

```
var b = 7;
b = 12.3;            // Fehler!
```

HINWEIS: Typinferenz funktioniert nur bei lokalen Variablen, also nicht auf globaler Ebene (siehe Abschnitt 2.2.10).

2.2.5 Typsuffixe

Wenn Sie Variablen im Quellcode direkte Zahlenwerte (Literale) zuweisen wollen, so werden diese vom Compiler standardmäßig als Datentyp *int* bzw. *double* interpretiert. Bei Datentypen wie *long*, *float* oder *decimal* müssen Sie ein so genanntes Typsuffix (*L, F, M*) anhängen, ansonsten werden die Literale wie *int* oder *double* behandelt, und es gibt einen Compilerfehler. Eine Übersicht enthält die folgende Tabelle.

Typsuffix	Gleitkommatyp
f oder *F*	*float*
d oder *D*	*double*
m oder *M*	*decimal*

BEISPIEL 2.14: Richtige und falsche Literalzuweisungen.

C#

```
float max = 99.99;        // Fehler!
float max = 99.99F;       // richtig
decimal geld = 300.50;    // Fehler!
decimal geld = 300.50M;   // richtig
```

2.2.6 Zeichen und Zeichenketten

Die Datentypen *char* und *string* basieren auf dem Unicode-Zeichensatz, der pro Zeichen 2 Byte beansprucht (im Unterschied zum klassischen ASCII- bzw. ANSI-Zeichensatz mit 1 Byte pro Zeichen). Mit dem Unicode können deshalb nicht mehr nur maximal 255, sondern bis zu 65535 (!) verschiedene Zeichen gespeichert werden.

char

Variablen vom *char*-Datentyp können Sie Zeichenliterale, hexadezimale Escape-Sequenzen oder Unicode-Darstellungen zuweisen.

HINWEIS: *char*-Literale sind in Hochkommata (') einzufassen.

BEISPIEL 2.15: char

C#

Drei gleichwertige Anweisungen deklarieren eine *char*-Variable und initialisieren diese mit dem Zeichen *A*:

```
char c = 'A';           // Zeichenliteral
char c = '\x0041';      // hexadezimal
char c = '\u0041';      // Unicode
```

Als weitere Möglichkeit kommt eine explizite Typkonvertierung direkt aus dem (ganzzahligen) Zeichencode in Betracht.

BEISPIEL 2.16: Eine weitere Ergänzung zum Vorgängerbeispiel

C#

```
char c = (char) 65;    // Typcasting liefert 'A'
```

string

HINWEIS: Stringliterale werden in doppelten Hochkommata ("Gänsefüßchen") eingefasst.

BEISPIEL 2.17: Zuweisen einer Stringvariablen

C#

```
string s = "Hallo";
```

Einen einzelnen *char* können Sie direkt aus einem *string* herauskopieren, indem Sie den Index in eckige Klammern schreiben. Dabei hat das erste Zeichen den Index 0.

BEISPIEL 2.18: Das erste Zeichen eines Strings ermitteln

C#

```
string s = "Hallo";
char c = s[0];           // liefert "H"
```

HINWEIS: Ausführliches zur Stringverarbeitung finden Sie in Kapitel 4.

Der umgekehrte Schrägstrich bzw. Backslash (\) spielt innerhalb eines Strings eine besondere Rolle, denn nachfolgende Zeichen werden vom C#-Compiler als Befehl interpretiert[1].

Die folgende Tabelle zeigt die häufigsten in C# benutzten Escapesequenzen.

Escape Sequenz	Bedeutung
\'	einfaches Anführungszeichen
\"	doppeltes Anführungszeichen
\\	Backslash
\a	Signalton
\b	Backspace (BS)
\f	Seitenvorschub
\n	Neue Zeile (LF)
\r	Wagenrücklauf CR)
\t	Horizontaler Tabulator (TAB)

HINWEIS: Bei einem Unicode-Zeichen folgt dem Backslash ein kleines u und die vierstellige Nummer des Zeichens, z.B. '\u0013'.

BEISPIEL 2.19: Korrekte Schreibweise für Dateipfade

C#

Die Anweisung zur Definition eines Dateipfads

```
string path = "C:\Benutzer\Doberenz";
```

... wird bereits von der Entwicklungsumgebung als "nicht erkannte Escapesequenz" zurückgewiesen, da der Backslash als Beginn einer Escapesequenz interpretiert wird.

Die folgende Anweisung wäre korrekt:

```
string path = "C:\\Benutzer\\Doberenz";
```

oder in diesem Fall auch die Kurzform:

```
string path = @"C:\Benutzer\Doberenz";
```

BEISPIEL 2.20: Hinzufügen eines Zeilenvorschubs mit Signalton

C#

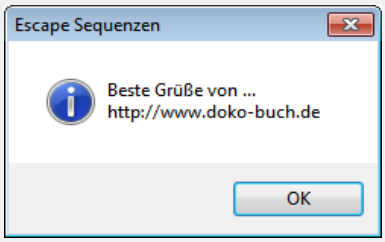
```
MessageBox.Show("Beste Grüße von ...\r\n\ahttp://www.doko-buch.de", "Escape Sequenzen",
                MessageBoxButtons.OK, MessageBoxIcon.Information);
```

[1] Siehe dazu auch Kapitel 4, Abschnitt 4.3 (Reguläre Ausdrücke).

BEISPIEL 2.20: Hinzufügen eines Zeilenvorschubs mit Signalton

Ergebnis

Ein Signalton ertönt und folgendes Meldungsfenster erscheint:



2.2.7 object-Datentyp

Der *object*-Datentyp ist weitaus mehr, als es der Name vermuten lässt. Alle Klassen des .NET-Frameworks sind von *System.Object* abgeleitet, und *object* ist dafür lediglich ein Alias.

Eine *object*-Variable ist ein so genannter *Verweis- oder Referenztyp*, denn sie speichert nicht den tatsächlichen Wert, sondern lediglich einen 4 Byte großen Zeiger auf die Adresse der zugewiesenen Variablen.

Variablen des Typs *object* können Sie alles zuweisen.

BEISPIEL 2.21: Einer Objektvariablen *o* wird eine *int*-Zahl zugewiesen.

C#

```
int i = 5;
object o = i;
```

Etwas komplizierter ist der umgekehrte Weg, nämlich wenn Sie den Wert der Objektvariablen einer anderen Variablen zuweisen wollen. In diesem Fall ist man auf explizite Typumwandlung (Typecasting) angewiesen (siehe Abschnitt 2.3.1). Eine direkte Zuweisung (implizite Typkonvertierung) führt zu einem Compilerfehler.

BEISPIEL 2.22: Die Fortsetzung des Vorgängerbeispiels

C#

```
int j = o;          // Fehler!
int j = (int) o;    // mit Typecasting ok!
```

Wichtig in diesem Zusammenhang ist das Verständnis des Unterschieds zwischen Werte- und Referenztypen und dem Prinzip des Boxing/Unboxing (siehe 2.3.6).

2.2.8 Konstanten deklarieren

Im Unterschied zu Variablen bleibt der Wert einer Konstanten während der gesamten Laufzeit eines Programms unverändert. Sie legen ihn einmalig mit dem *const*-Schlüsselwort fest. Die Deklaration ist ähnlich wie bei initialisierten Variablen.

SYNTAX: `const Datentyp Konstantenname = Wert;`

BEISPIEL 2.23: Verschiedene Konstantendeklarationen

C#
```
const int c = 119;
const float PI  = 3.1415F;
const double X1  = 3 * 0.4, X2 = 5.3 + 0.68;
const string s = "Hallo";
```

Sammlungen von Konstanten werden üblicherweise in so genannten *Enumerationen* "zusammengehalten" (siehe Abschnitt 2.6.1).

2.2.9 Nullable Types

C# erfordert seit eh und je die explizite Initialisierung von Variablen.

BEISPIEL 2.24: Falsche und richtige Verwendung von Variablen

C#
```
int z;
z++;              // falsch, weil z nicht initialisiert ist
...
int z = 0;
z++;              // richtig
```

Initialisieren von Wertetypen mit null

Etwas komplizierter wird es aber, wenn man einen Wert mit "nichts" (*null*) initialisieren möchte.

BEISPIEL 2.25: Das funktioniert nicht!

C#
```
int z = null;       // falsch
```

Durch ein der Typdeklaration nachgestelltes Fragezeichen (*?*) kann der Compiler jetzt einen Wertetyp in die generische *System.Nullable*-Struktur verpacken, wodurch es möglich wird, einer Variablen den Wert *null* zuzuweisen.

BEISPIEL 2.26: Aber das funktioniert!

C#
```
int? z = null;
z = 1;
```

Oder als explizite Deklaration:

```
System.Nullable<Int32> z = null;
z = 1;
```

Zuweisungen mit dem ??-Operator

Um einen *Nullable Type* (werteloser Typ) einer anderen Variablen zuweisen zu können, muss vorher eine *null*-Abfrage erfolgen, wie sie mittels *HasValue*-Eigenschaft möglich ist.

BEISPIEL 2.27: Die Variable *y* wird mit der Zahl 0 initialisiert, da *x* den Wert null hat.

C#
```
int? x = null;
int y;
if (x.HasValue) y = (int) x;
else y = 0;
```

Deutlich eleganter und kürzer ist eine solche Zuweisung aber, wenn man dazu das doppelte Fragezeichen (*??*) verwendet, es liefert den Wert des vorangestellten Ausdrucks falls dieser nicht *null* ist, anderenfalls den Wert des nachfolgenden Ausdrucks.

BEISPIEL 2.28: Das Vorgängerbeispiel wird einfacher realisiert.

C#
```
int? x = null;
int y = x ?? 0;
```

BEISPIEL 2.29: Im *Label* erscheint der Text "nichts zugewiesen".

C#
```
string s = null;
label1.Text = s ?? "nichts zugewiesen!";
```

2.2.10 Typinferenz

Dieses in Zusammenhang mit der LINQ-Technologie (siehe Kapitel 6) eingeführte Sprachfeature erlaubt es, dass der Datentyp einer Variablen bei der Deklaration vom Compiler automatisch ermittelt wird, ohne dass Sie explizit den Typ angeben müssen. Als Ersatz für einen konkreten Typ wird das Schlüsselwort *var* verwendet.

HINWEIS: Damit der Compiler den Typ der Variablen feststellen kann, muss eine mit *var* deklarierte Variable unbedingt bei der Deklaration initialisiert werden.

BEISPIEL 2.30: *var*-Deklaration

C#

Die Initialisierung der Variablen *a* wird vom Compiler ausgewertet und der Typ aufgrund des Wertes 35 auf *Integer* festgelegt.

```
var a = 35;
```

Obige Zeile ist semantisch identisch mit folgendem Ausdruck:

```
int a = 35;
```

Der Datentyp wird einmalig bei der ersten Deklaration der Variablen vom Compiler festgelegt und kann danach nicht mehr verändert werden.

BEISPIEL 2.31: Keine Änderung möglich ...

C#

Da die Variable *b* vom Compiler als *Integer* festgelegt wurde, kann ihr später kein *double*-Wert zugewiesen werden.

```
var b = 7;
b = 12.3;            // Fehler!
```

HINWEIS: Typinferenz funktioniert nur bei lokalen Variablen (siehe folgender Abschnitt)!

2.2.11 Gültigkeitsbereiche und Sichtbarkeit

Obwohl sich in C# alles innerhalb von Klassen abspielt, werden wir erst im OOP-Kapitel 3 ausführlicher auf diese Thematik zu sprechen kommen.

Trotzdem sollten Sie bereits jetzt Folgendes wissen:

- Lokale Variablen gelten standardmäßig nur innerhalb ihres – in geschweiften Klammern eingerahmten – Bereichs und der untergeordneten Bereiche. Ein Zugriff von außerhalb ist nicht möglich.
- Sie können die Schlüsselwörter *private* und *public* verwenden um festzulegen, ob auf die Variablen auch von außerhalb zugegriffen werden kann.
- *public*-Konstanten sind nicht empfehlenswert, weil das leicht zu Namenskonflikten führen kann. Wenn Sie auf Klassen- oder Namespace-Ebene mit (globalen) Konstanten arbeiten möchten, verwenden Sie am besten eine Enumeration (siehe Abschnitt 2.6.1).
- Wenn Sie eine Variable nicht als *public* oder *private* deklarieren, ist sie standardmäßig *private*. Man bezeichnet die Schlüsselwörter *private* und *public* auch als *Zugriffsmodifizierer*, sie gelten nicht nur für Variablen, sondern auch für Klassen, Objekte, Eigenschaften und Methoden (siehe Kapitel 3, Abschnitt 3.1.3).
- Im Unterschied zu Visual Basic gibt es in C# keine *Static*-Variablen.

2.3 Konvertieren von Datentypen

C# ist eine typsichere Sprache und nimmt es deshalb mit den Datentypen sehr genau. Schon bei den geringsten Nachlässigkeiten schlagen Ihnen IDE oder Compiler gnadenlos auf die Finger.

2.3.1 Implizite und explizite Konvertierung

Unabhängig vom tatsächlichen Wert, wie er in der Variablen gespeichert ist, lassen sich verschiedene Datentypen nur dann gegenseitig zuweisen, wenn der Wertebereich des rechten Datentyps in den linken "passt". In einem solchen Fall findet eine so genannte *implizite Konvertierung* statt, die der Compiler automatisch vornimmt.

BEISPIEL 2.32: Die Zuweisung *Byte* zu *Integer* funktioniert problemlos.

C#
```
byte b = 100;
int i = b;              // implizite Typkonvertierung
```

Geradezu oberlehrerhaft verhält sich C# im umgekehrten Fall. Egal ob der Wert in den kleineren Datentyp passt oder nicht – es wird halt gemeckert.

BEISPIEL 2.33: Das geht nicht

C#

Obwohl der Wert 100 problemlos in eine *Byte*-Variable passt, erscheint die Fehlermeldung "Implizite Konvertierung des Typs 'int' zu 'byte' nicht möglich!"

```
int i = 100;
byte b = i;             // Fehler!
```

Um den meckernden Compiler zu beschwichtigen, ist eine so genannte *explizite Typkonvertierung* (auch *Typecasting* genannt) erforderlich.

SYNTAX: *neueVariable* = (*Neuer Datentyp*) *alteVariable*;

BEISPIEL 2.34: Das Vorgängerbeispiel wird fehlerfrei ausgeführt

C#
```
int i = 100;
byte b = (byte) i;      // explizite Typkonvertierung
```

Implizite Konvertierungen sind sicher, Datenverluste sind deshalb ausgeschlossen. Dabei kann stets nur der kleinere der beiden Datentypen direkt in einen größeren umgewandelt werden[1].

Explizite Typkonvertierungen sollten stets mit Vorsicht angewendet werden, wobei man sicher sein muss, dass die Wertebereiche zur Laufzeit nicht überschritten werden.

HINWEIS: Man muss sich immer bewusst sein, dass eine explizite Typkonvertierung dann zu Datenverlusten führen kann, wenn der Wertebereich durch die Konvertierung verkleinert wird.

BEISPIEL 2.35: Da der *byte*-Datentyp nur den Bereich 0 ... 255 abdeckt, entsteht ein falsches Ergebnis

C#
```
int i = 300;
byte b = (byte) i;      // liefert falsches Resultat (44)
```

[1] Sie können sich das bildlich so vorstellen, dass jeder Datentyp einem Kochtopf mit unterschiedlichem Fassungsvermögen entspricht, und Sie dürfen immer nur etwas aus einem kleineren in einen größeren Topf füllen. Verboten wäre es beispielsweise, aus einem 1-Liter-Topf etwas in einen 0,5-Liter-Topf zu gießen, obwohl im 1-Liter-Topf nur 0,1 Liter enthalten sind!

BEISPIEL 2.36: Implizite und explizite Typkonvertierung *float* in *int* gegenübergestellt

```
C#
int i;
float f = 12.5F;
i = f;              // implizite Konvertierung ergibt Fehler!
i = (int) f;        // explizite Konvertierung ergibt Datenverlust: i erhält den Wert 12
```

BEISPIEL 2.37: Implizite Typkonvertierung *char* in *int*

```
C#
char c = 'A';
int i = c;                  // i erhält den Wert 65  (Zeichencode von 'A')
```

BEISPIEL 2.38: Explizite Typkonvertierung des Ergebnisses einer Division

```
C#
int i = 3;
float x = i / 10;             // x erhält den Wert 0
float x = (float) i / 10;     // x erhält den Wert 0,3
```

as-Konverierungsoperator

Eine Alternative zur expliziten Typumwandlung mittels *()*-Konvertierung ist der *as*-Operator, der allerdings nur auf Verweis- und nicht auf Wertetypen anwendbar ist. Auch alle Steuerelemente gehören zu den Verweistypen!

BEISPIEL 2.39: Konvertieren des *sender*-Parameters eines Eventhandlers

```
C#
this.Text = (sender as TextBox).Text;
```

2.3.2 Welcher Datentyp passt zu welchem?

Der folgenden Tabelle entnehmen Sie alle möglichen impliziten und expliziten Typkonvertierungen. Die impliziten Konvertierungen sind fett hervorgehoben.

Quell-Datentyp	Zieldatentypen
bool	*object*
byte	*ushort, short, uint, int, ulong, long, float, double, decimal, object, sbyte, char*
sbyte	***short, int, long, float, double, decimal, object,*** *byte, ushort, uint, ulong, char*
short	***int, long, float, double, decimal, object,*** *sbyte, byte, ushort, uint, ulong, char*
ushort	***uint, int, ulong, long, float, double, decimal, object,*** *sbyte, byte, short, char*
char	***ushort, uint, int, ulong, long, float, double, decimal, object,*** *sbyte, byte, short*
int	***long, float, double, decimal, object,*** *sbyte, byte, short, ushort, uint, ulong, char*
uint	***long, float, double, decimal, object,*** *sbyte, byte, short, ushort, int, char*

Quell-Datentyp	Zieldatentypen
long	***float, double, decimal, object,*** *sbyte, byte, short, ushort, int, uint, ulong, char*
ulong	***float, double, decimal, object,*** *sbyte, byte, short, ushort, int, uint, long, char*
float	***double, object,*** *sbyte, byte, short, ushort, int, uint, long, ulong, char, decimal*
double	***object,*** *sbyte, byte, short, ushort, int, uint, long, ulong, char, float, decimal*
decimal	***object,*** *sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double*
string	*object*
object	alle Datentypen (Unboxing, siehe Abschnitt 2.3.6)

2.3.3 Konvertieren von string

Laut obiger Konvertierungstabelle lässt sich der *string*-Datentyp nur in den universellen *object*-Datentyp umwandeln und in umgekehrter Richtung scheint gar nichts zu gehen Doch die Entwarnung folgt zugleich.

ToString-Methode

Der *object*-Datentyp – gewissermaßen die "Mutter aller Objekte" – vererbt an alle Nachkommen die *ToString*-Methode, auf welche Sie bereits hin und wieder in den bisherigen Beispielen gestoßen sind, nämlich dann, wenn es darum ging, Zahlenwerte zur Anzeige zu bringen.

HINWEIS: Jeder Datentyp kann mittels seiner *ToString*-Methode in den Datentyp *string* umgewandelt werden!

Und noch ein Hinweis, den sich besonders die von Visual Basic kommenden Umsteiger hinter die Ohren schreiben sollten:

HINWEIS: Vergessen Sie nicht die Klammern hinter *ToString()*!

BEISPIEL 2.40: Anzeige einer Gleitkommazahl

C#
```
double d = 12.75;
MessageBox.Show(d.ToString);     // Fehler
MessageBox.Show(d.ToString());   // ok
```

BEISPIEL 2.41: Konvertieren eines *bool* in *string*

C#
```
bool b = true;
string s = b.ToString();         // "True"
```

String in Zahl verwandeln

Zwar können wir mit der *ToString()*-Methode alle Datentypen in den *string*-Typ konvertieren, wie aber sieht es umgekehrt aus?

Für bestimmte andere Datentypen gibt es spezifische Lösungen, z.B. zum Umwandeln von *string* in *char*.

BEISPIEL 2.42: Einem *char* wird das zweite Zeichen eines *string* zugewiesen.

```
C#  string name = "Max";
    char c = name[1];
    MessageBox.Show(c.ToString());      // zeigt "a"
```

Damit enden vorerst unsere Erfolgserlebnisse, denn das übliche Typecasting scheint bei den anderen Datentypen zu versagen.

BEISPIEL 2.43: Das geht leider nicht.

```
C#  string s = "5";
    int i = (int) s;          // Fehler!
```

Rettung naht auch hier in Gestalt der *Convert*-Klasse (siehe auch nächster Abschnitt). Als Alternative zu den expliziten Typkonvertierungen bietet diese Klasse für nahezu jeden Datentyp eine spezielle (statische) Konvertierungsmethode, der man den zu konvertierenden Ausdruck als Argument übergibt.

BEISPIEL 2.44: Das Vorgängerbeispiel kann wie folgt gelöst werden.

```
C#  string s = "5";
    int i = Convert.ToInt32(s);
    MessageBox.Show(i.ToString());      // zeigt "5"
```

BEISPIEL 2.45: Ein *string* wird in eine *double*-Zahl konvertiert.

```
C#  string s = "23,50";
    double d = Convert.ToDouble(s);     // d erhält den Wert 23,50
```

Alternativ kann auch die *Parse*-Methode eingesetzt werden.

BEISPIEL 2.46: Konvertieren eines Stringliterals in eine Ganzzahl.

```
C#  int nr = Int32.Parse("12");
```

EVA-Prinzip

Auch für (fast) jedes Programm gilt nach wie vor das uralte EVA-Prinzip (Eingabe, Verarbeitung, Ausgabe). In diesem Zusammenhang sei nochmals auf die besondere Bedeutung der Typkonvertierung von und in den *string*-Datentyp hingewiesen. Da unter Windows sehr häufig die Übergabe-

werte als Zeichenketten vorliegen (*Text*-Eigenschaft der Ein- und Ausgabefelder), müssen sie zunächst in Zahlentypen umgewandelt werden, um dann nach ihrer Verarbeitung wieder in Zeichenketten rückverwandelt und (formatiert) zur Anzeige gebracht zu werden.

BEISPIEL 2.47: Ein Ausschnitt aus dem Einführungsbeispiel 1.7.2

C#
```
euro = Convert.ToSingle(textBox1.Text);         // Eingabe: string => float
dollar = euro * kurs;                           // Verarbeitung
textBox2.Text = dollar.ToString("#,##0.00");    // Ausgabe: float => string
```

2.3.4 Die Convert-Klasse

Diese statische Klasse bietet für jeden einfachen Datentyp eine spezielle Konvertierungsmethode, der man den zu konvertierenden Ausdruck als Argument übergibt.

SYNTAX: `Convert.`*`typMethode`*`(object `*`expr`*`);`

typeMethode = Konvertierungsmethode (*ToBoolean*, *ToByte, ToInt32, ToDouble* ...)

expr = zu konvertierender Ausdruck

BEISPIEL 2.48: *string* wird in *double* konvertiert

C#
```
string s = "55,7";
double d = Convert.ToDouble(s);          // 55,7
```

BEISPIEL 2.49: *bool* wird in *int* und in *string* konvertiert

C#
```
bool b = true;
int i = Convert.ToInt32(b);             // 1
b = false;
i = Convert.ToInt32(b);                 // 0
string s = Convert.ToString(b);         // "False"
```

2.3.5 Die Parse-Methode

Die numerischen Typen *Byte*(*byte*), *Int16*(*short)*, *Int32(int)*, *Int64*(*long)*, *Single*(*float)* und *Double*(*double*) verfügen u.a. über die (statische) *Parse*-Methode, welche die Stringdarstellung einer Zahl in den entsprechenden Typ konvertieren kann.

BEISPIEL 2.50: Der Inhalt einer *TextBox* wird in eine Gleitkommazahl konvertiert.

C#
```
double z = Double.Parse(textBox1.Text);
```

HINWEIS: Die *Parse*-Methode hat den Vorteil, dass zusätzlich Kulturinformationen eines bestimmten Landes mit übergeben werden dürfen.

2.3.6 Boxing und Unboxing

Die Begriffe *Boxing/Unboxing* gehören zu den häufig strapazierten .NET-Schlagwörtern, die manchem Einsteiger Ehrfurcht einflößen. Was verbirgt sich dahinter? Sie wissen bis jetzt, dass Sie dem universellen *object*-Datentyp jeden Wert direkt zuweisen können, d.h. durch implizite Typkonvertierung. Umgekehrt kann, falls es der *object*-Inhalt erlaubt, jeder Datentyp durch explizite Typkonvertierung (Typecasting) aus *object* wieder "herausgezogen" werden. Das direkte Zuweisen funktioniert in diesem Fall nicht.

BEISPIEL 2.51: Boxing und Unboxing

C# Eine *bool*-Variable wird in ein *object* "verpackt" (Boxing) und dieses anschließend einer zweiten *bool*-Variablen zugewiesen (Unboxing).

```
bool b1 = true;
object o = b1;          // ok, implizite Konvertierung (Boxing)
bool b2 = o;            // Fehler, implizite Konvertierung
bool b2 = (bool) o;     // ok, explizite Konvertierung (Unboxing, b2 ist true)
```

Um den tieferen Sinn von Boxing/Unboxing zu verstehen, sollten Sie sich nochmals den Unterschied zwischen den beiden fundamentalen Arten von Datentypen vergegenwärtigen, d.h., zwischen den Wertetypen und den Verweis- bzw. Referenztypen (siehe Abschnitt 2.2.2).

Boxing

Es erhebt sich nun die Frage, was denn passiert, wenn man einer *object*-Variablen, d.h. einem Verweistyp, einen Wertetyp zuweist, der naturgemäß im Stack gespeichert ist.

BEISPIEL 2.52: Ein Integer wird einem object-Datentyp zugewiesen.

C#
```
int i = 25;
object o = i;
```

Die genauere Fragestellung ist, worauf zeigt die *object*-Variable o? Der Zeiger *o* darf doch keinesfalls auf den Stack verweisen (das würde die Stabilität des Programms massiv gefährden)!

Die Antwort: Es findet ein automatischer Kopiervorgang statt, d.h., eine Kopie der Variablen *i* wird auf dem Heap abgelegt, auf die dann die *object*-Variable *o* zeigt.

Unboxing

Wie greift man nun aber wieder auf den in der *object*-Variablen "eingepackten" Wert zu? Eine einfache (implizite) Zuweisung funktioniert nicht. Richtig ist eine explizite Typkonvertierung (Typecasting).

BEISPIEL 2.53: Das Vorgängerbeispiel wird fortgesetzt.

C#
```
int j = o;                  // Fehler!
int j = (int)  o;           // ok
```

Allerdings funktioniert das Typecasting nur dann, wenn die Objektvariable tatsächlich auf den gewünschten Typ verweist, Trickserei – wie im folgenden Beispiel – nützt Ihnen also nichts.

BEISPIEL 2.54: Das geht nicht!

C#

Die Hoffnung, bei der Umwandlung *string* nach *int* vielleicht ohne *Convert*-Klasse (siehe oben) auszukommen, geht leider nicht in Erfüllung.

```
string s = "5";
object o = s;
int i = (int) o;    // Fehler!
```

HINWEIS: Das Boxing ist mit ein wesentlicher Grund, warum in .NET "alles ein Objekt" ist, denn auch Wertetypen können damit quasi wie Objekte behandelt werden.

BEISPIEL 2.55: Ja, auch das funktioniert!

C#

```
int i = new int();
i = 12;
```

2.4 Operatoren

Operatoren verknüpfen Variablen bzw. Operanden miteinander und führen Berechnungen durch. Wir unterscheiden zwischen

- arithmetischen Operatoren,
- Zuweisungsoperatoren,
- logischen Operatoren und
- Vergleichsoperatoren.

Die meisten Operatoren in C# benötigen zwei Operanden.

BEISPIEL 2.56: Operanden

C#

Im Ausdruck

```
i = 12;
```

ist der *Operator* das Gleichheitszeichen (=), die beiden *Operanden* sind die Variable *i* und die Literalkonstante *12*.

HINWEIS: C# erlaubt auch das Überladen von Operatoren, auf das wir aber erst an späterer Stelle eingehen wollen (siehe Kapitel 5).

2.4.1 Arithmetische Operatoren

Standard-Operatoren

Es gibt zunächst die üblichen Operatoren für die Grundrechenarten:

Operator	Beispielausdruck	Erklärung
+	*x + y*	Addition
-	*x - y*	Subtraktion
*	*x * y*	Multiplikation
/	*x / y*	Division
%	*x % y*	Modulo-Division (liefert Restwert)

BEISPIEL 2.57: Standard-Operatoren

```
C#
int i, j = 6;
i = 3 *(4 + 5) * j;    // 162
i = 7 % 3;             // 1 (Rest!)
```

HINWEIS: Achten Sie bei der Division von Literalen darauf, dass das Ergebnis abgerundet wird, wenn nicht mindestens einer der Operanden als Gleitkommatyp gekennzeichnet ist.

BEISPIEL 2.58: Nur die beiden letzten Divisionen liefern das exakte Ergebnis.

```
C#
double d;
d = 7 / 3;             // 2   (Ergebnis wird abgerundet!)
d = 7D / 3;            // 2,3333333333333
d = 7.0 / 3;           //       dto.
```

Inkrement- und Dekrement-Operatoren

Mit den Kurz-Operatoren ++ und -- lässt sich das schrittweise Erhöhen (Inkrementieren) bzw. Erniedrigen von Variablen (Dekrementieren) vereinfachen.

Operator	Beispielausdruck	Erklärung
++	x++ ++x	Postfix-Inkrement Präfix-Inkrement
--	x-- --x	Postfix-Dekrement Präfix-Dekrement

Wie Sie den Beispielen in obiger Tabelle entnehmen, können Sie die Kurz-Operatoren ++ und -- nicht nur hinter den Namen der Variablen (Postfix), sondern auch davor (Präfix) schreiben.

BEISPIEL 2.59: Postfix-Inkrement und Postfix-Dekrement

C#
```
int i = 10;
i++;                // i erhält den Wert 11
double d = 2.5;
d--;                //  d erhält den Wert 1,5
```

BEISPIEL 2.60: Äquivalente Version des Vorgängerbeispiels mit Präfixoperationen

C#
```
int i = 10;
++i;                // i erhält den Wert 11
double d = 2.5;
--d;                //  d erhält den Wert 1,5
```

Wie Sie sehen, haben beide Schreibweisen keinerlei Einfluss auf den Wert der Variablen, diese wird in jedem Fall um 1 inkrementiert bzw. dekrementiert, wozu also sollen Postfix- und Präfix-Notationen dann gut sein?

Um den "feinen" Unterschied zu verstehen, muss man wissen, dass nicht nur die Variable (z.B. *i*) einem bestimmten Wert entspricht, sondern auch die mit dem Kurz-Operator verknüpfte Variable (z.B. *i*++). Letztere hat bei einer Postfix-Operation den Wert **vor** der Inkrementierung bzw. Dekrementierung, bei einer Präfix-Operation hingegen den Wert **nach** Inkrementierung bzw. Dekrementierung.

BEISPIEL 2.61: Ergebnisse einer Postfix-Inkrementation werden im Meldungsfenster angezeigt.

C#
```
int i = 10;
MessageBox.Show((i++).ToString());   // i++ hat den Wert 10
MessageBox.Show(i.ToString());       // i hat den Wert 11
```

BEISPIEL 2.62: Dasselbe für eine Präfix-Inkrementation.

C#
```
int i = 10;
MessageBox.Show((++i).ToString());     // ++i hat den Wert 11
MessageBox.Show(i.ToString());         // i hat den Wert 11
```

Eine besondere Rolle spielen Postfix- und Präfix-Schreibweise bei der Steuerung von *while*- und *do*-Schleifen (siehe Abschnitt 2.5.2).

2.4.2 Zuweisungsoperatoren

Die Tabelle zeigt, dass es neben dem simplen Zuweisungsoperator (=) noch fünf andere mit arithmetischen Operatoren verknüpfte Kurz-Operatoren gibt.

Operator	Beispielausdruck	Erklärung
=	x = y	x wird der Wert von y zugewiesen
+=	x += y	x ergibt sich zu x + y

Operator	Beispielausdruck	Erklärung
-=	x -= y	x ergibt sich zu x - y
*=	x *= y	x ergibt sich zu x * y
/=	x /= y	x ergibt sich zu x / y
%=	x %= y	x ergibt sich als Restbetrag aus x/y

Die Kurz-Operatoren bringen relativ bescheidene Verbesserungen, sie erleichtern das Schreiben von Quellcode und erhöhen die Übersichtlichkeit.

BEISPIEL 2.63: Kurz-Operatoren

C#

Statt

```
i = i + 3;
```

kann man auch schreiben

```
i += 3;
```

BEISPIEL 2.64: Kurz-Operatoren

C#

Für

```
string s = "Hallo";
```

gibt es diese

```
s = s + " .NET-Freunde!";
```

oder diese Möglichkeit zum Anhängen einer weiteren Zeichenkette:

```
string s = "Hallo";
s += " .NET - Freunde!";              // "Hallo .NET - Freunde!"
```

2.4.3 Logische Operatoren

Logische Operatoren basieren auf Ja-/Nein- bzw. *true-/false*-Werten. In C# ist dazu ein reichhaltiges Angebot enthalten.

Vergleichsoperatoren

Vergleichs- oder relationale Operatoren vergleichen zwei Ausdrücke miteinander und liefern als Ergebnis einen *true/false*-Wahrheitswert.

Operator	Erklärung
==	x gleich y?
!=	x ungleich y?
<	x kleiner y?

Operator	Erklärung
<=	x kleiner oder gleich y?
>	x größer y?
>=	x größer oder gleich y?

BEISPIEL 2.65: Vergleich von zwei Integer-Zahlen

C#
```
bool b = 10 < 5;                // b ist false
```

Besondere Bedeutung haben Vergleichsoperatoren im Zusammenhang mit Verzweigungsbefehlen, wie wir sie in Abschnitt 2.5.1 kennen lernen werden. Das folgende Beispiel liefert einen Vorgeschmack.

BEISPIEL 2.66: Vergleichsoperatoren im Zusammenhang mit Verzweigungsbefehlen,

C#

Wenn der Wert der Variablen min gleich 59 ist, wird ihr Wert auf 0 gesetzt, ansonsten um 1 erhöht.

```
if (min == 59) min = 0; else min++;
```

Obwohl *string* ein Verweistyp ist, werden die Gleichheitsoperatoren (== und !=) so definiert, dass die Werte von *string*-Objekten und keine Verweise verglichen werden.

BEISPIEL 2.67: Zwei Strings werden verglichen.

C#
```
string a = "Hallo";
string b = "H";
b += "allo";                          // b wird zu "Hallo"
Console.WriteLine( a == b );          // liefert true, da die Werte gleich sind
Console.WriteLine( (object) a == b ); // liefert false, da es verschiedene Objekte sind
```

Boolesche Operatoren

Diese Operatoren werden auf boolesche Variablen (*true/false*) angewendet:

Operator	Erklärung
&	**Und:** liefert *true*, wenn beide Operanden *true* sind
|	**Oder:** liefert *true*, wenn mindestens einer der Operanden *true* ist
^	**Exklusiv-Oder (XOR):** liefert *true*, wenn genau nur einer der beiden Operanden *true* ist
&&	**intelligentes Und:** wie &-Operator, aber ist der erste Operand *false*, wird der zweite nicht ausgewertet
||	**intelligentes Oder:** wie |-Operator, aber ist der erste Operand *true*, wird der zweite nicht ausgewertet
!	**Negation:** aus *true* wird *false* und aus *false* wird *true*.

BEISPIEL 2.68: Boolesche Operatoren

C#
```
bool b = (true & true) | (false & true);      // b wird true
bool z = (10 < 5) ^ (11 > 11)                 // z wird false
bool schalter = true;
schalter = !schalter;                         // schalter wird false
```

Kurzschlussauswertung

Ist bei einer UND-Verknüpfung der linke Teil *false*, so kann auf die Auswertung des rechten Teils verzichtet werden, da das Ergebnis sowieso *false* ist. Ist bei einer ODER-Verknüpfung der linke Teil *true*, so steht ebenfalls das Ergebnis bereits fest (*true*).

Diese Gesetzmäßigkeit machen sich die "intelligenten" Verknüpfungsoperatoren && und || zunutze, indem sie ein auch als *Kurzschlussauswertung* bekanntes Verfahren verwenden. Wenn der linke Teil bereits zu einem eindeutigen Ergebnis führt, wird der rechte Teil gar nicht erst ausgewertet.

BEISPIEL 2.69: Kurzschlussauswertung

C#

Es wird das gleiche Ergebnis wie im Vorgängerbeispiel erzielt, aber es wird weniger Rechenzeit benötigt, da der rechte Teil nicht ausgewertet wird (der linke Teil ist *true*).

```
bool b = (true && true) || (false && true);         // b wird true
```

HINWEIS: Verwenden Sie die Operatoren && und || anstatt & und |, da Sie dadurch Rechenzeit einsparen!

Bitweise Operationen

Mit den folgenden Operatoren (von denen Ihnen die ersten drei bereits bekannt sind) lassen sich bitweise Verknüpfungen durchführen. Sie verknüpfen also nicht mehr die booleschen Variablen *true* und *false*, sondern die einzelnen Bits (0 bzw. 1) von zwei Zahlen.

Operator	Erklärung
&	bitweise "UND"-Verknüpfung der beiden Operanden
|	bitweise "ODER"-Verknüpfung der beiden Operanden
^	bitweise "XOR"-Verknüpfung der beiden Operanden
>>	Rechtsverschiebung aller Bits eines Operanden um eine bestimmte Anzahl
<<	Linksverschiebung aller Bits eines Operanden um eine bestimmte Anzahl

BEISPIEL 2.70: Die XOR-Verknüpfung der Integer-Zahlen 1 und 7 ergibt 6.

C#
```
int a, b;
a = 1;                  // Bitmuster = 001
b = 7;                  // Bitmuster = 111
```

BEISPIEL 2.70: Die XOR-Verknüpfung der Integer-Zahlen 1 und 7 ergibt 6.

```
a = a ^ b;              // Bitmuster = 110  (a erhält den Wert 6)
```

BEISPIEL 2.71: Die Bitfolge der Zahl 1 wird um zwei Stellen nach links "geshiftet" und ergibt 4.

C#
```
int a = 1;             // Bitmuster = 001
a = a << 2;            // Bitmuster = 100    (a erhält den Wert 4)
```

2.4.4 Rangfolge der Operatoren

Es ist klar, dass bei einem Zuweisungsoperator (=) immer erst die rechte Seite ausgerechnet und dann der linken Seite zugewiesen wird. Aber in welcher Reihenfolge werden die Operationen auf der rechten Seite ausgeführt? Antwort gibt die folgende Tabelle, welche die Operatoren in ihrer hierarchischen Rangfolge zeigt.

Operator	Bedeutung
( )	Klammern
!	logisches NOT
* / %	Multiplikation, Division, Modulo
+ -	Addition, Subtraktion
< <= > >=	kleiner als, kleiner gleich als, größer als, größer gleich als
== !=	gleich, ungleich

Die weiter oben in der Hierarchie stehenden Operationen werden immer **vor** den weiter unten stehenden ausgeführt.

HINWEIS: Durch Einschließen in runde Klammern ( ) kann die hierarchische Reihenfolge außer Kraft gesetzt werden.

BEISPIEL 2.72: Arithmetische Operationen

C#
```
double x = 2.0;
double y  = x * x + 1 + x / 4;              // y = 5,5
```

aber

```
double y  = x * (x + 1 + x / 4);            // y = 7
```

BEISPIEL 2.73: Boolesche Operationen

C#
```
bool b = !true && false || 5 > 6;           // b = false
int z = 50;
bool numeric = z > 47 && z < 58;            // numeric = true
```

2.5 Kontrollstrukturen

Verzweigungs- und Schleifenanweisungen unterbrechen den linearen Programmablauf und gehören zum Einmaleins des Programmierens.

2.5.1 Verzweigungsbefehle

"Programmweichen" werden durch Verzweigungsanweisungen bzw. -funktionen oder auch durch Sprungbefehle gestellt.

Klassische Entscheidungsanweisungen

Die folgende Tabelle gibt einen Überblick[1].

Verzweigungsanweisung	Erklärung

`if (Bedingung)`
`  Anweisung1;`
`else`
`  Anweisung2;`
	bedingte Verzweigung
	Wenn die *Bedingung* zutrifft, wird *Anweisung1* ausgeführt, ansonsten *Anweisung2*.
	Der *else*-Zweig kann auch weggelassen werden.

`if (Bedingung1)`
`  Anweisung1;`
`else if (Bedingung2)`
`  Anweisung2;`
`else if (Bedingung3)`
`  Anweisungen ...;`
	verschachtelte Verzweigung
	Zweige werden so lange ausgewertet, bis eine der Bedingungen *true* ergibt

`switch (Ausdruck)`
`{`
`  case Ausdruck1 :`
`  Anweisungen;`
`  break;`
`  case Ausdruck2 :`
`  Anweisungen;`
`  break;`
`  ...`
`  default :`
`  Anweisungen;`
`  break;`
`}`
	Blockstruktur
	Der Ausdruck wird mit den hinter *case* angeführten Ausdrücken verglichen. Nach erstem Erfolg wird der Block verlassen, ansonsten werden die *default*-Anweisungen ausgeführt.
	Der *default*-Zweig kann auch weggelassen werden, in diesem Fall erfolgt eine Fortsetzung nach Blockende.
	Die *break*-Befehle verhindern das "Durchfallen".

In vielen Fällen werden Sie zum Prüfen von Bedingungen die *if*-Anweisung verwenden.

[1] Statt der einzelnen Anweisungen können auch Anweisungsblöcke stehen, die dann in geschweifte Klammern einzuschließen sind.

BEISPIEL 2.74: In den Labels wird "Verbessern" und "Du bekommst nichts!" angezeigt

C#
```
int zensur = 3;
if (zensur == 1)
{
    label1.Text = "Gratuliere!";
    label2.Text = "Du bekommst einen Blumenstrauß!";
}
else
{
    label1.Text = "Verbessern!" ;
    label2.Text = "Du bekommst nichts!";
}
```

Optional können Sie innerhalb des *if*-Blocks noch *else*- oder *else if*-Zweige verwenden, wobei die *else if*-Bedingung nur dann geprüft wird, wenn keine der vorstehenden *if*-Bedingungen erfüllt war.

BEISPIEL 2.75: Im Label wird "Befriedigend" angezeigt

C#
```
int zensur = 3;
if (zensur == 1)
    label1.Text =  "Sehr gut!";
else if (zensur == 2)
    label1.Text =  "Gut";
else if (zensur == 3)                          // zutreffende Bedingung
    label1.Text =  "Befriedigend";
    //(usw.)
else
  label1.Text = "Nicht erlaubte Zensur!";
```

Mit dem *switch*-Konstrukt wird ein Ausdruck auf mehrere mögliche Ergebnisse hin überprüft. Im Testausdruck kann ein beliebiger arithmetischer oder logischer Ausdruck stehen.

BEISPIEL 2.76: Diese Kontrollstruktur leistet das Gleiche wie das Vorgängerbeispiel

C#
```
int zensur = 3;
switch (zensur) {
    case 1: label1.Text = "Sehr gut"; break;
    case 2: label1.Text =  "Gut"; break;
    case 3: label1.Text =  "Befriedigend"; break;     // zutreffende Bedingung
    //(usw.)
    default : label1.Text = "Nicht erlaubte Zensur!"; break;
}
```

HINWEIS: Sie können *switch* nur bei einfachen Datentypen wie *byte* und *int* sowie *string* verwenden. In allen anderen Fällen müssen Sie *if*-Konstrukte nehmen.

Um eine identische Aktion bei mehreren möglichen Vergleichswerten auszuführen, schreiben Sie die einzelnen *case*-Zweige einfach hintereinander und lassen dabei das Schlüsselwort *break* weg.

BEISPIEL 2.77: Das *Label* zeigt "Frühling" an

C#
```
byte monat = 5;
switch (monat) {
    case 12 : case 1 :
    case 2 : label1.Text =  "Winter"; break;
    case 3 : case 4 :
    case 5 : label1.Text =  "Frühling"; break;       // zutreffende Bedingung
    case 6 : case 7 :
    case 8 : label1.Text =  "Sommer"; break;
    case 9 : case 10 :
    case 11: label1.Text =  "Herbst"; break;
    default :
    label1.Text =  "kein gültiger Monat!"; break;
}
```

HINWEIS: Sie sollten, wo immer es geht, statt einer *if*-Anweisung mit eingeschachtelten *else if*-Verzweigungen eine *switch*-Anweisung verwenden. Diese wird wesentlich schneller ausgeführt, da die Prüfbedingung nur einmal auszuwerten ist.

In der Prüfbedingung für das *if*-Konstrukt wird auch oft vom Negations-Operator (*!*) Gebrauch gemacht:

BEISPIEL 2.78: An den Verzeichnisnamen *myPath* wird ein Slash (/) angehängt, falls keiner vorhanden ist

C#
An den Verzeichnisnamen *myPath* wird ein Slash (/) angehängt, falls keiner vorhanden ist.
```
if (!myPath.EndsWith("/")) myPath += "/";
```

Ergänzung

Ein weniger gebräuchlicher Verzweigungsbefehl basiert auf dem Fragezeichen (?) und durch Doppelpunkt (:) getrennten Zielanweisungen. Dadurch lässt sich eine kompaktere Schreibweise erzwingen.

BEISPIEL 2.79: Der Verzweigungsbefehl

C#
```
label1.Text = checkBox1.Checked ? "Ja"  : "Nein";
```
entspricht
```
if (checkBox1.Checked) label1.Text = "Ja";
else label1.Text = "Nein";
```

BEISPIEL 2.80: In Abhängigkeit von einer booleschen Variablen erhält i den Wert 1 oder 2

C#
```
int i = checkBox1.Checked ? 1 : 2;
```

Sprungbefehle

Verzweigungen können auch mit Sprunganweisungen realisiert werden. Innerhalb von Sprunganweisungen werden die Schlüsselwörter *continue*, *default*, *goto* und *return* eingesetzt.

So sind innerhalb eines *switch*-Konstrukts auch absolute Sprünge mittels *goto* möglich. Die *case*- oder *default*-Anweisungen sind die Sprungziele.

BEISPIEL 2.81: Eine Alternative zum Vorgängerbeispiel (auszugsweise)

C#
```
switch (monat)
{
  case 12: goto case 2;
  case 1:  goto case 2;
  case 2 : label1.Text =  "Winter"; break;
  case 3 : goto case 5;
  case 4 : goto case 5;
  case 5 : label1.Text =  "Frühling"; break;
// usw.
```

2.5.2 Schleifenanweisungen

Die wichtigsten Grundtypen sind *for*-, *while*- und *do*-Schleifen (siehe Tabelle).

Schleifenanweisung	Erklärung

`for (Zähler=Anfangswert; Abbruchbedingung; Zähler=neuerWert)`
`{`
`  Anweisungen;`
`}`
	***for*-Zählschleife,**
	wird so lange durchlaufen, bis Abbruchbedingung *false* ist

`while (Abbruchbedingung)`
`{`
`  Anweisungen;`
`}`
	***while*- Bedingungsschleife,**
	Abbruchbedingung am Schleifenanfang (kopfgesteuert)

`do`
`{`
`  Anweisungen;`
`}`
`while (Abbruchbedingung)`
	***do*- Bedingungsschleife,**
	Abbruchbedingung am Schleifenende (fußgesteuert)

HINWEIS: Ein weiterer Schleifentyp, die *foreach*-Schleife, spielt im Zusammenhang mit Arrays und Auflistungen eine wichtige Rolle (siehe Kapitel 4).

for-Schleifen

In diesem klassischen Schleifentyp wird die Zählervariable pro Durchlauf aktualisiert (meist inkrementiert bzw. dekrementiert), bis eine Abbruchbedingung erfüllt ist. Die Initialisierung, der boolesche Ausdruck und die Anweisung zur Aktualisierung der Zählvariablen müssen durch Semikolons voneinander getrennt sein.

BEISPIEL 2.82: Die Schleife gibt zehnmal untereinander den laufenden Index und einen Text in einer *ListBox* aus

C#
```
for (int i = 1; i<=10; i++)
       listBox1.Items.Add(i.ToString() + " Viele Wege führen nach Rom!");
```

Sie können jedes der drei Elemente (Initialisierung, Abbruchbedingung, Aktualisierung) im Schleifenkopf auch weglassen, müssen sich aber dann anderweitig um Ersatz bemühen.

BEISPIEL 2.83: Eine äquivalente Version des Vorgängerbeispiels

C#
```
int i = 1;                              // Ersatz für Initialisierung der Zählvariablen
for (; i<=10; )
{
    listBox1.Items.Add(i.ToString() + " Viele Wege führen nach Rom!");
    i++;                                // Ersatz für Aktualisierung der Zählvariablen
}
```

while-Schleife

Bei diesem Schleifentyp steht die Organisation einer Zählervariablen nicht im Mittelpunkt, wodurch eine etwas flexiblere Programmierung möglich wird. In Abhängigkeit davon, ob die Abbruchbedingung am Schleifenanfang oder an deren Ende kontrolliert wird, spricht man auch von *kopfgesteuerten* bzw. *fußgesteuerten* Schleifen. Die *while*-Schleife ist – ebenso wie die *for*-Schleife – kopfgesteuert.

BEISPIEL 2.84: Ein völlig identisches Resultat wie obige for-Schleifen

C#
```
int i = 1;
while (i <= 10)
{
   listBox1.Items.Add(i.ToString() + " Viele Wege führen nach Rom!");
   i++;
}
```

do-Schleife

Dieser dritten Schleifentyp ähnelt der *while*-Schleife, allerdings wird die Abbruchbedingung erst am Ende getestet.

BEISPIEL 2.85: Eine weitere äquivalente Version der Vorgängerbeispiele

C#
```
int i = 1;
do
{
  listBox1.Items.Add(i.ToString() + " Viele Wege führen nach Rom!");
  i ++;
}
while (i <= 10);
```

Das komplette Programm finden Sie im Praxisbeispiel

► 2.8.3 Schleifenanweisungen verstehen

Wozu braucht man die *do*-Schleife überhaupt, wenn man doch mit der *while*-Schleife zum gleichen Ergebnis kommt?

Die Antwort: Manchmal brauchen Sie vielleicht eine Schleife, bei der die Anweisungen im Schleifenkörper mindestens einmal ausgeführt werden. In diesem Fall dürfte der Unterschied deutlich sein.

BEISPIEL 2.86: Ersetzen Sie die erste Anweisung in beiden Vorgängerbeispielen durch

C#
```
int i = 11;
```

Während die *do*-Schleife eine einzige Zeile ausgibt, bleibt die *ListBox* bei der *while*-Schleife leer.

Vorzeitiges Verlassen einer Schleife

Genauso wie Sie mit *break* eine Verzweigung verlassen können, ist damit auch ein vorzeitiger Abbruch einer Schleifenanweisung möglich.

BEISPIEL 2.87: Vorzeitiges Verlassen einer Schleife

C#

Auch dieser Code gibt zehnmal untereinander einen Text in einer *ListBox* aus. Statt aber die Abbruchbedingung im Schleifenkopf festzulegen, wird sie in den Schleifenkörper verlagert.

```
for (int i = 1; ; i++)
{
   listBox1.Items.Add(i.ToString() + " Viele Wege führen nach Rom!");
   if (i == 10) break;            // Abbruchbedingung testen
}
```

Im Gegensatz zu *break* bewirkt *continue*, dass die Abbruchbedingung sofort (also nicht erst beim nächsten Schleifeneintritt) ausgewertet wird.

HINWEIS: Da die *continue*-Anweisung zu schwer auffindbaren Programmierfehlern führen kann, sollten Sie sie nur in Ausnahmefällen verwenden.

2.6 Benutzerdefinierte Datentypen

Sie sind als Programmierer nicht nur auf die einfachen Datentypen *int, float, ...* angewiesen, sondern können auch eigene Datentypen kreieren. Interessant sind vor allem

- Enumerationen und
- strukturierte Datentypen.

Beide bauen auf den einfachen Datentypen auf und sind – ebenfalls wie diese – so genannte *Wertetypen*, die im Stack des Computers gespeichert werden.

2.6.1 Enumerationen

Mit dem *enum*-Schlüsselwort erstellen Sie einen Enumtyp, dessen mögliche Werte auf eine bestimmte Menge symbolischer Namen beschränkt ist. Häufig werden Sammlungen von miteinander verwandten Konstanten in Enumerationen zusammengefasst.

Deklaration

Der *enum*-Datentyp wird ähnlich wie ein strukturierter Datentyp deklariert.

SYNTAX:
```
enum Bezeichner : Datentyp
{
  Feld1 = Wert1;
  Feld2 = Wert2;
  ...
}
```

Alle in der Enumeration enthaltenen Konstanten müssen vom gleichen Datentyp sein, zulässig sind nur die acht Integer-Typen *byte*, *sbyte, short*, *ushort, int, uint, long* und *ulong*. Falls die Angabe des Datentyps weggelassen wird, handelt es sich automatisch um *int*-Konstanten.

BEISPIEL 2.88: Eine Enumeration für die Monate eines Jahres

C#
```
enum Monate : byte
{Januar = 1, Februar, März, April, Mai, Juni,
 Juli, August, September, Oktober, November, Dezember}
```

Jedem Element der Enumeration ist ein bestimmter Zahlenwert zugeordnet, der standardmäßig mit 0 beginnt. Falls Sie das ändern wollen, so müssen Sie den Wert explizit zuordnen. Wenn Sie das nur für den ersten Wert tun, so erhalten die Nachfolger vom Compiler automatisch den nächsthöheren Wert (siehe obiges Beispiel).

Zugriff

Wenn ein Enumtyp deklariert ist, können Sie ihn wie jeden anderen Datentyp verwenden. Das heißt, Sie deklarieren damit auf gewohnte Weise Variablen oder Konstanten, denen Sie dann Werte zuweisen.

BEISPIEL 2.89: Deklarieren der Konstanten *ersterMonat* vom Enumtyp *Monate*

```
C# const Monate ersterMonat = Monate.Januar;
```

BEISPIEL 2.90: Deklarieren von zwei Variablen des Enumtyps *Monate*

```
C# Monate monat1, monat2;
   monat1 = Monate.Januar;
   monat2 = monat1 + 1;
   MessageBox.Show(monat2.ToString());    // zeigt Februar
```

Um nicht den Namen, sondern die Integer-Zahl des Elements zu ermitteln, müssen Sie eine Typkonvertierung vornehmen.

BEISPIEL 2.91: Der folgende Code wird an das Vorgängerbeispiel angehängt

```
C# byte m = (byte) monat2;
   MessageBox.Show(m.ToString());        // zeigt 2
```

2.6.2 Strukturen

Mit dem *struct*-Schlüsselwort definieren Sie komplexe Datentypen, die ein oder auch mehrere Element(e) enthalten dürfen. In diesem Abschnitt wollen wir aber Strukturen nicht tiefgründiger behandeln, sondern nur einen ersten Einblick gewähren.

Deklaration

Vom Prinzip her entspricht die Definition einer *Struktur* der einer *Klasse*, die allerdings zu den so genannten *Verweis*- oder *Referenztypen* gehört (siehe Kapitel 3).

Eine einfache Struktur hat folgenden Aufbau:

```
SYNTAX: struct Bezeichner
        {
           Modifizierer Datentyp feld1;
           Modifizierer Datentyp feld2;
           ...
        }
```

Beachten Sie:

- Die *struct*-Anweisung ist innerhalb von Methoden unzulässig und nur auf globaler Ebene anwendbar.
- Damit die Elemente einer Struktur von außerhalb sichtbar sind, muss der *public*-Modifizierer vorangestellt werden[1].

[1] Diese Darstellung ist etwas vereinfacht, die ausgereifte Programmierung verlangt auch hier, genauso wie bei Klassen, das Prinzip der Kapselung.

BEISPIEL 2.92: Eine Struktur

C# Um in einer Variablen zur Personenbeschreibung neben dem Namen auch noch das Alter zu erfassen, definieren Sie einen neuen Datentyp:

```
struct Person
{
  public string vorName, nachName;
  public byte alter;
}
```

HINWEIS: Denken Sie immer daran, dass allein mit der Definition eines Datentyps noch keine Variable dieses Typs existiert! Diese muss – wie jede andere Variable auch – erst noch deklariert werden.

BEISPIEL 2.93: Sie erzeugen zwei Variablen des oben definierten Datentyps *Person*

```
C# Person person1, person2;
```

Datenzugriff

Um auf den Wert einer Strukturvariablen zuzugreifen, müssen Name und Element durch einen Punkt (so genannter *Qualifizierer*) voneinander getrennt sein[1].

BEISPIEL 2.94: Fortsetzung des Vorgängerbeispiels

```
C# int a; string name;
person1.vorName = "Max";        // Initialisieren  (Schreibzugriff)
person1.nachName = "Müller";    // dto.
person1.alter = 50;             // dto.
a = person1.alter;               // Lesezugriff
name = person1.vorName + "  " + person1.nachName;          // "Max Müller"
```

HINWEIS: Beim Lesezugriff auf nicht initialisierte Felder erhalten Sie einen Compilerfehler.

BEISPIEL 2.95: Das Feld *alter* von *person2* wurde nicht initialisiert (siehe Bemerkungen)

```
C# a = person2.alter;    // Fehler!
```

Man kann natürlich nicht nur, wie eben beschrieben, auf die einzelnen Felder einer Strukturvariablen, sondern auch auf die Variable insgesamt zugreifen.

BEISPIEL 2.96: Die Variable *person1* wird "geklont"

```
C# person2 = person1;
```

[1] Dies entspricht der generellen Schreibweise beim Zugriff auf Objekteigenschaften.

Struktur versus Klasse

Wie bereits bekannt, haben Strukturen viele Ähnlichkeiten mit Klassen. Genauso wie diese können sie beispielsweise über einen oder mehrere (überladene) Konstruktoren verfügen, mit denen die Felder initialisiert werden können. Aber es gibt da mehrere wesentliche Unterschiede, unter anderem können Sie für eine Struktur selbst keinen Standard-Konstruktor (das ist einer ohne Parameter, d.h. mit leeren Klammern) erstellen, denn dies wird immer vom Compiler erledigt, der die Felder dann (je nach Datentyp) mit den Werten *0*, *null* oder *false* initialisiert.

BEISPIEL 2.97: Die Strukturvariable *person1* wird mit ihrem Standard-Konstruktor erzeugt und initialisiert

C#
```
Person person1 = new Person();
int a = person1.alter;                // a erhält automatisch den Wert 0
```

Und noch ein wesentlicher Unterschied zu Klassen und anderen Verweistypen:

HINWEIS: Bei Strukturvariablen spielt sich nach wie vor alles auf dem Stack ab (auch das Instanziieren mit *new*), sie sind Wertetypen und für diese ist der Heap tabu!

2.7 Nutzerdefinierte Methoden

Methoden kapseln wiederverwendbaren Programmcode und erleichtern somit nicht nur die Arbeit des Programmierers, sondern tragen auch im erheblichen Maß zur Übersichtlichkeit des Programmcodes bei.

Jede Methode hat einen Namen und einen Körper. Letzterer enthält die beim Methodenaufruf auszuführenden Anweisungen.

Grundsätzlich unterscheiden wir zwischen

- Methoden **mit** Rückgabewert
- Methoden **ohne** Rückgabewert

Methoden **mit** Rückgabewert sind Ihnen sicherlich aus anderen Programmiersprachen[1] als *Funktionen* ein Begriff, während eine Methode **ohne** Rückgabewert als *Prozedur, Sub* oder einfach nur als *Unterprogramm* bezeichnet wird.

In diesem Abschnitt wollen wir Methoden noch nicht als öffentliche (*public)* Mitglieder einer Klasse verstehen, wie sie in der OOP das Verhalten von Objekten definieren, denn dazu kommen wir erst im Kapitel 3. Vorerst behandeln wir sie lediglich als private Klassenmitglieder.

HINWEIS: Für das Verständnis der Parameterübergabe an Methoden ist es wichtig, dass Sie den Unterschied zwischen Werte- und Referenztypen kennen (siehe 2.2.2).

[1] z.B. Visual Basic oder Delphi

2.7.1 Methoden mit Rückgabewert

Deklaration

Die Deklaration ist ähnlich wie bei einer normalen Variablen, nur dass an den Namen der Methode in Klammern die Parameterliste angefügt wird (kann auch leer sein).

SYNTAX:
```
Datentyp Name (Parameterliste)
{
   // Code definieren
   // .....
   // ...
   return (Rückgabewert)
}
```

Die *Parameterliste* ist eine durch Kommata separierte Liste mit der Aufzählung der einzelnen Parameter:

SYNTAX: `([ref|out] Datentyp Parameter, [ref|out] Datentyp Parameter, ...)`

HINWEIS: Zur Bedeutung der optionalen Übergabe mit *ref* bzw. *out* siehe die Abschnitte 2.7.3 und 2.7.4.

Der Methodenkörper ist durch eine öffnende und eine schließende geschweifte Klammer eingegrenzt.

Als Rückgabewert kommt natürlich nur der im Methodenkopf deklarierte Datentyp in Frage. Den Rückgabewert, der auch ein Ausdruck sein kann, schreiben Sie in Klammern hinter die *return*-Anweisung.

BEISPIEL 2.98: Funktion zur Berechnung des Kugelgewichts.

C#

Übergabeparameter sind der Radius *ra* und das spezifische Gewicht *sg*.

```
private double Kugel(double ra, double sg)
{
   double vol = 4 / 3F * Math.PI * Math.Pow(ra, 3);
   return(sg * vol);
}
```

HINWEIS: Schreiben Sie die *return*-Anweisung immer an das Ende des Methodenkörpers, denn alle dahinter stehenden Anweisungen werden nicht mehr ausgeführt.

Aufruf

Beim Aufruf einer Methode muss die *Signatur* der Parameterliste sorgfältig beachtet werden, d.h., die Reihenfolge der Argumente und ihr Datentyp müssen zur deklarierten Parameterliste passen.

Im aufrufenden Code muss der Rückgabewert einer Variablen zugewiesen werden, die den gleichen Datentyp wie die Methode hat.

BEISPIEL 2.99: Obige Funktion wird mit einer eisernen Kugel von 20 cm Durchmesser getestet.

C#
```
double r = 10, s = 7.87;           // Radius ist 10cm, spez. Gewicht = 7,87 gr/cm3
double gew = Kugel(r, s);          // Aufruf
MessageBox.Show(gew.ToString("#,#0.000 Gramm"));   // Gewicht der Kugel ist 32.965,779 Gramm
```

HINWEIS: Bei **jedem** Methodenaufruf müssen Sie die runden Klammern **()** angeben, auch wenn die Methode keine Parameter besitzt!

2.7.2 Methoden ohne Rückgabewert

Deklaration

Soll die Methode keinen Wert zurückgeben, so setzen Sie anstelle des Datentyps das *void*-Schlüsselwort ein.

SYNTAX:
```
void Name (Parameterliste)
{
   // Code definieren
   // .....
}
```

Eine *void*-Methode arbeitet oft mit *Feldern* zusammen. Unter Feldern verstehen wir in C# globale Variablen, die auf Klassenebene deklariert wurden (im Unterschied zu den lokalen Variablen im Methodenkörper).

BEISPIEL 2.100: Anwenden einer *void*-Methode zum Berechnen des Kugelgewichts.

C#

Deklarationen auf Klassenebene:

```
private double gew;      // dieser globalen Variablen wird das Ergebnis zugewiesen

private void Kugel(double ra, double sg)
{
  double vol = 4 / 3F * Math.PI * Math.Pow(ra, 3);
  gew = sg * vol;        // Feld wird mit dem Ergebnis gefüttert
}
```

Aufruf

Im Unterschied zum Aufruf einer Methode mit Rückgabewert handelt es sich diesmal um keine direkte Zuweisung des Rückgabewerts, da dieser der globalen Variablen *gew* entnommen wird.

BEISPIEL 2.101: Unsere Methode wird nun ebenfalls mit der legendären Eisenkugel getestet.

C#

```
double r = 10, sg = 7.87;          // Radius ist 10cm, spez. Gewicht = 7,87 gr/cm3
Kugel(r, sg);                      // Aufruf
MessageBox.Show(gew.ToString("#,#0.000 Gramm"));   //  zeigt "32.965,779 Gramm"
```

HINWEIS: Obwohl Sie in einer *void*-Methode die *return*-Anweisung nicht brauchen, können Sie diese trotzdem zum vorzeitigen Verlassen der Methode verwenden. Dazu schreiben Sie einfach *return* gefolgt von einem Semikolon.

2.7.3 Parameterübergabe mit ref

Bei jedem Methodenaufruf werden – je nach Länge der Parameterliste – kein, ein oder auch mehrere Argument(e) an die Parameter der Methode übergeben. Handelt es sich hierbei um einen Wertetyp, so wird der entsprechende Parameter automatisch zu einer Kopie des Arguments.

Wenn Sie aber bei der Methodendeklaration dem Parameter das Schlüsselwort *ref* voranstellen, so wird nicht eine Kopie, sondern der Zeiger auf die Speicherplatzadresse des Arguments übermittelt. Im Körper der aufgerufenen Methode wird es dadurch möglich, am Wert des Arguments "herumzudoktern", denn sämtliche hier vorgenommenen Änderungen am Wert des Parameters wirken sich auf das Argument aus. Auf diese Weise können Methoden ihre Ergebnisse ohne Verwendung globaler Variablen direkt an die aufrufenden Argumente zurückliefern.

HINWEIS: Wenn Sie ein Argument an einen *ref*-Parameter übergeben, müssen Sie beim Aufruf ebenfalls das *ref*-Schlüsselwort voranstellen.

BEISPIEL 2.102: Parameterübergabe mit *ref*

C#

Bei dieser Methode zur Kugelgewichtsbestimmung wurde die Übergabeart des zweiten Parameters in *ref* geändert. Dadurch wird es möglich, über diesen Parameter auch das Berechnungsergebnis zu übertragen.

```
private void Kugel(double ra, ref double g)
{
  double vol = 4 / 3F * Math.PI * Math.Pow(ra, 3);
  g = g * vol;        // g zeigt auf das Ergebnis
}
```

Den Parameter *g* haben wir hier bewusst nicht mit *sg* (für spezifisches Gewicht) bezeichnet, weil er nach dem Aufruf der *Kugel*-Methode auf das Gewicht verweist, also dann z.B. *gew* heißen müsste.

Test

Testen der Methode:

```
double r = 10, g = 7.87;
Kugel(r, ref g);
MessageBox.Show(g.ToString("#,#0.000 Gramm"));     // zeigt "32.965,779 Gramm"
```

Noch einmal – aber bitte langsam

Der *ref*-Parameter hat nur für Wertetypen (dazu gehören z.B. die einfachen Datentypen wie *int*, *double*) eine relevante Bedeutung. Mit der standardmäßigen Parameterübergabe (also ohne *ref*) wird der Wert des übergebenen Parameters in eine vom Compiler erzeugte lokale Variable des Methodenkörpers kopiert. Die aufrufende Variable (das Argument) und die lokale Variable im Inneren der Methode (der Parameter) haben danach nichts mehr miteinander zu tun, eine Veränderung des Wertes der lokalen Variablen bleibt ohne Rückwirkung auf das Argument, sodass ein "Zurückgeben" von Ergebnissen – wie im obigen Beispiel gezeigt – nicht möglich ist.

Wenn Sie das Argument (einen Wertetyp) hingegen mit *ref* übergeben, so verfügt die aufgerufene Methode nur scheinbar über eine eigene lokale Variable. In Wirklichkeit wird mit dem ursprünglichen Argument weitergearbeitet, eine "Entkopplung" hat also nicht stattgefunden.

HINWEIS: Im Interesse der Fehlersicherheit Ihrer Programme sollten Sie *ref*-Parameter möglichst sparsam einsetzen und ansonsten lieber mit Kopien arbeiten. *ref* sollten Sie nur dann verwenden, wenn Sie tatsächlich Werte von Wertetypen "nach außen hin" verändern wollen.

2.7.4 Parameterübergabe mit out

Das Schlüsselwort *out* hat eine sehr ähnliche Bedeutung wie *ref*. Also wird auch hier der Wert des Arguments nicht in den Parameter kopiert, sondern der Parameter wird lediglich zu einem Alias des Arguments.

Einziger Unterschied zwischen *ref* und *out* ist, dass eine Methode einem *ref*-Parameter einen Wert zuweisen **kann**, einem *out*-Parameter hingegen einen Wert zuweisen **muss**.

HINWEIS: Ein Argument, das an einen *ref*-Parameter übergeben wird, muss zuvor initialisiert werden. Dies ist beim *out*-Parameter nicht erforderlich.

BEISPIEL 2.103: Parameterübergabe mit *out*

C#

Dieses Beispiel würde mit *ref* nicht funktionieren, da der beim Aufruf übergebene Parameter *g* nicht initialisiert ist.

```
void Kugel(double ra, double sg, out double gew)
{
  double vol = 4 / 3F * Math.PI * Math.Pow(ra, 3);
  gew = sg * vol;        // gew zeigt auf das Ergebnis
}
```

Test

Der Aufruf:

```
double r = 10, s = 7.87, g;        // g ist nicht initialisiert!
Kugel(r, s, out g);
MessageBox.Show(g.ToString("#,#0.000 Gramm"));    // zeigt "32.965,779 Gramm"
```

2.7.5 Methodenüberladung

Wenn mehrere gleichnamige Methoden im gleichen Gültigkeitsbereich ohne Namenskonflikt friedlich nebeneinander existieren, so spricht man von *überladenen* Methoden. Die Unterscheidung wird vom Compiler anhand der Signatur getroffen. Unter Signatur verstehen wir bekanntlich die Reihenfolge der übergebenen Parameter und deren Datentyp.

HINWEIS: Überladene Methoden müssen immer eine unterschiedliche Signatur haben!

BEISPIEL 2.104: Methodenüberladung

C#

Die *Kugel*-Funktion steht in zwei überladenen Versionen zur Verfügung. Die erste Version verlangt als Parameter den Radius und das spezifische Gewicht.

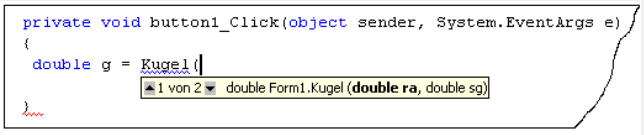
```
double Kugel(double ra, double sg)
{
    double vol = 4 / 3.0 * Math.PI * Math.Pow(ra, 3);
    return(sg * vol);
}
```

Eine zweite Version soll nur das Volumen der Kugel berechnen.

```
double Kugel(double ra)
{
   double vol = 4 / 3.0 * Math.PI * Math.Pow(ra, 3);
   return(vol);
}
```

Test

Wenn Sie jetzt die Methode verwenden wollen, werden Ihnen in der Visual Studio-Entwicklungsumgebung automatisch beide überladenen Versionen angeboten:



Der Aufruf ist entweder so

```
double g = Kugel(10, 7.87);          // liefert  32.965,779
```

oder so möglich:

```
double v = Kugel(10);               // liefert  4.188,790
```

Bemerkungen

- Unterscheiden sich die Deklarationen zweier Methoden nur in der Verwendung von *ref* bzw. *out*, erfolgt ebenfalls eine Überladung.
- Das Überladen des Rückgabewerts einer Methode ist nicht erlaubt.

- Wie Sie bestimmt schon bei der Arbeit im Codefenster von Visual Studio bemerkt haben, werden für die meisten Methoden mehrere Überladungen angeboten. Ein typisches Beispiel hierfür ist die *Show*-Methode der *MessageBox* mit insgesamt 12 (!) Überladungen.

2.7.6 Optionale Parameter

Ein optionaler Parameter gestattet es Ihnen, einem Parameter einen Standardwert zuzuweisen. Beim Aufruf der Methode bleibt es Ihnen überlassen, für diesen Parameter ein Argument zu übergeben oder nicht.

Das folgende Beispiel zeigt, wie Sie in früheren C#-Versionen optionale Parameter mittels Methodenüberladungen simulieren konnten.

BEISPIEL 2.105: Zwei Überladungen einer Methode

C#
```
public static string Hallo()              // erste Überladung
{
    return "Hallo Du!";
}

public static string Hallo(string name)    // zweite Überladung
{
    return "Hallo " + name + "!";
}
```

Bei Verwendung dieses Codes erscheint der *name*-Parameter wie ein optionaler Parameter, denn die folgenden beiden Aufrufe funktionieren:

```
string grüssDich = Hallo();
string grüssFernando = Hallo("Fernando");
```

Wenn Sie Methoden mit mehr Parametern hätten, müssten Sie noch mehr Überladungen bereitstellen. Vielen Entwicklern gefällt dies nicht, da dadurch der Code aufquillt wie Hefeteig.

Das folgende Beispiel zeigt einen optionalen Parameter, welcher die obigen zwei Überladungen ersetzt:

BEISPIEL 2.106: Methode mit einem optionalen Parameter

C#
```
public static string Hallo(string name = "Du")
{
    return "Hallo " + name + "!";
}
```

Wie zu sehen ist, erfordert die Aufrufsyntax für einen optionalen Parameter lediglich das Zuweisen eines Standardwerts (*default value*). Falls beim Aufruf kein Wert übergeben wird, wird der Standardwert genommen.

Auch das folgende Beispiel aus der Office-Automation zeigt, wie man mittels optionaler Parameter den Code vereinfachen kann.

BEISPIEL 2.107: Die *SaveAs*-Methode für ein Excel-Worksheet in klassischer Schreibweise:

C#
```
wkBook.SaveAs("myfile.xls", Missing.Value, Missing.Value,
              Missing.Value, Missing.Value, Missing.Value,
              Excel.XlSaveAsAccessMode.xlShared,
              Missing.Value, Missing.Value, Missing.Value, Missing.Value, Missing.Value);
```

Nur zwei Argumente werden im obigen Beispiel bereitgestellt, die übrigen sind *Missing.Value*.

Jetzt die vereinfachte Version mit optionalen Parametern:

```
wkBook.SaveAs("myfile.xls", Excel.XlSaveAsAccessMode.xlShared);
```

Wenn Sie bislang für Methoden wie *SaveAs* immer kostbare Zeit verschwenden mussten, um die exakte Position eines Parameters herauszufinden, werden Sie die Kürze obigen Codes zu schätzen wissen[1].

2.7.7 Benannte Parameter

Eng verwandt mit den optionalen Parametern sind die benannten Parameter. Einer ihrer Hauptvorzüge ist die Auflösung von Mehrdeutigkeiten. Benannte Parameter erkennt man nicht anhand der Methodendeklaration, sondern lediglich an der Syntax des Methodenaufrufs.

HINWEIS: Benannte Parameter tragen beim Aufruf denselben Namen wie die entsprechenden Parameter in der Methodendeklaration, wobei ein Doppelpunkt angehängt wird.

BEISPIEL 2.108: Eine Methode mit mit drei Parametern

C#
```
public static void addBuch(string titel, string autor = "", DateTime? termin = null)
{
    // noch nicht implementiert
}
```

Der Parameter *titel* ist nicht optional, *autor* hingegen ist optional, beide sind vom Datentyp *string*. Der letzte Parameter zeigt, dass man als Standardwert einem Parameter auch *null* zuweisen kann.

Der Aufruf der Methode mit benannten Parametern:

```
addBuch(autor: "Doberenz und Gewinnus",  termin: DateTime.Now, titel: "Hallo Fernando!");
```

Bewusst haben wir hier die Reihenfolge der übergebenen Parameter verändert, da diese bei benannten Parametern keine Rolle spielt.

Benannte Parameter sind besonders dann nützlich, wenn Sie mehrere optionale Parameter gleichen zu übergebenden Typs haben.

[1] Im Fall des zweiten Arguments käme auch ein benannter Parameter (siehe folgender Abschnitt) infrage. Da aber die Parameterliste nur ein Argument enthält, ist eine Benennung nicht erforderlich.

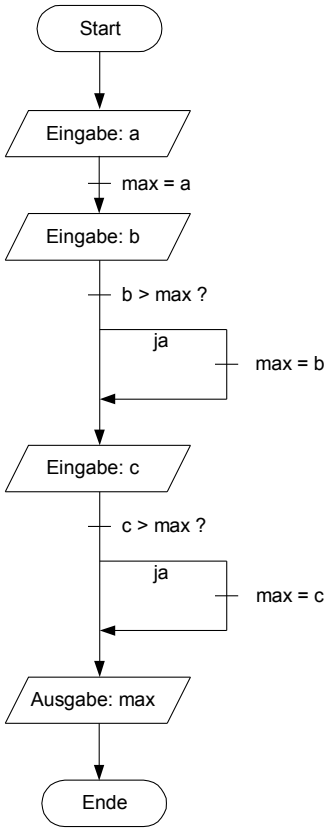
2.8 Praxisbeispiele

2.8.1 Vom PAP zur Konsolenanwendung

Dieses ausgesprochene Einsteiger-Beispiel erläutert die Umsetzung eines Programmablaufplans (PAP) in eine klassische Konsolenanwendung. Es sind nacheinander drei positive ganze Zahlen einzugeben. Das Programm soll die größte der drei Zahlen ermitteln und das Ergebnis anzeigen!

Programmablaufplan

Der nachfolgend abgebildete PAP zeigt die Berechnungsvorschrift (Algorithmus). Sie erkennen hier die typische EVA-Grundstruktur, bei der die Anweisungen in der Reihenfolge **E**ingabe, **V**erarbeitung, **A**usgabe ausgeführt werden.



Programmierung

Beim Schreiben der Befehle (z.B. mit dem Editor aus dem Windows-Zubehör) gibt obiger PAP eine nützliche Orientierung:

```
using System;
class Maximum3
{
   static void Main()
   {
      Console.WriteLine("Maximum von drei Zahlen");          // Überschrift
      Console.WriteLine();                                   // Leerzeile
      int a, b, c, max;                                      //  Variablendeklaration

      Console.WriteLine("Geben Sie die erste Zahl ein!");
      a = Convert.ToInt32(Console.ReadLine());               // Eingabe a
      max = a;                                               // Initialisieren von max
      Console.WriteLine("Geben Sie die zweite Zahl ein!");

      b = Convert.ToInt32(Console.ReadLine());               // Eingabe von b
      if (b > max) max = b;                                  // Bedingung
      Console.WriteLine("Geben Sie die dritte Zahl ein!");

      c = Convert.ToInt32(Console.ReadLine());               // Eingabe c
      if (c > max) max = c;                                  // Bedingung
      Console.WriteLine("Das Maximum ist " + max.ToString());      // Ergebnisausgabe
      Console.ReadLine();                        // Programm wartet auf <Enter>, um zu beenden
   }
 }
```

Speichern Sie die Textdatei als *Maximum3.cs* ab.

Kompilieren

Die Vorgehensweise entspricht exakt dem Abschnitt 1.2 im Einführungskapitel, Sie müssen also zunächst diverse Vorbereitungen treffen (Umgebungsvariable für den C#-Compiler hinzufügen, Datei *cmd.exe* in das Anwendungsverzeichnis kopieren), um bequem kompilieren zu können.

Geben Sie dann den folgenden Text an der Kommandozeile ein:

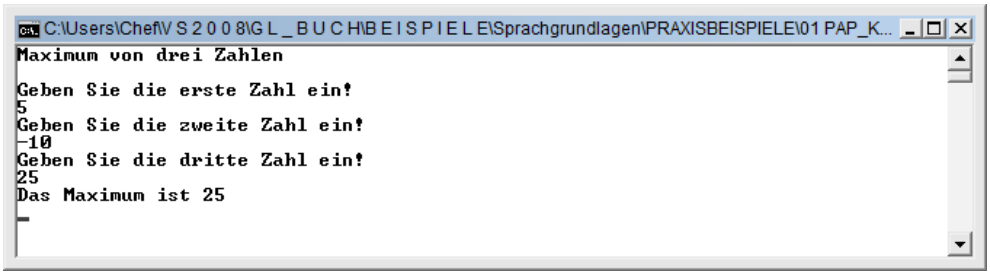
```
csc /t:exe Maximum3.cs
```

Haben Sie beim Eintippen des Quellcodes keine Fehler gemacht, so dürfte das Kompilieren anstandslos verlaufen.

Im Projektverzeichnis finden Sie nun die Datei *Maximum3.exe* vor.

Test

Starten Sie *Maximum3.exe* durch Doppelklick!



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

2.8.2 Ein Konsolen- in ein Windows-Programm verwandeln

Eine Windows-Anwendung ist natürlich wesentlich attraktiver als eine Konsolen-Applikation und schließlich wollen Sie ja zukünftig mit dem Komfort von Visual Studio statt mit einem simplen Texteditor arbeiten!

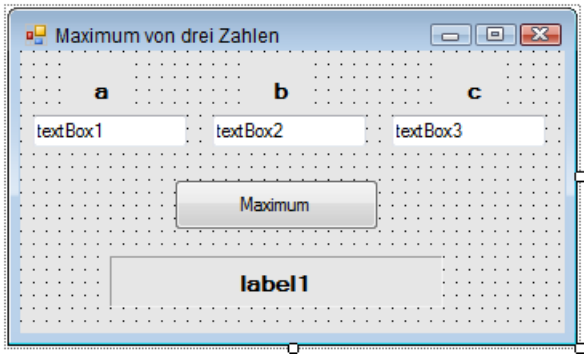
Ziel dieses Beispiels soll es sein, das im Vorgängerbeispiel erstellte Konsolen-Programm in eine "richtige" Windows Forms-Applikation zu verwandeln.

Oberfläche

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

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

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



Quelltext

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

HINWEIS: Sie sollten den Rahmencode der Eventhandler nur in Ausnahmefällen selbst eintippen. Lassen Sie sich stattdessen den Rahmencode durch die Visual Studio-Entwicklungsumgebung generieren (falls es sich um das Standardereignis handelt durch Doppelklick auf das Steuerelement, ansonsten über die "Ereignisse"-Seite des Eigenschaftenfensters)!

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

namespace Maximum3
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            // Hier müssen Ihre C#-Anweisungen eingefügt werden!
        }
    }
}
```

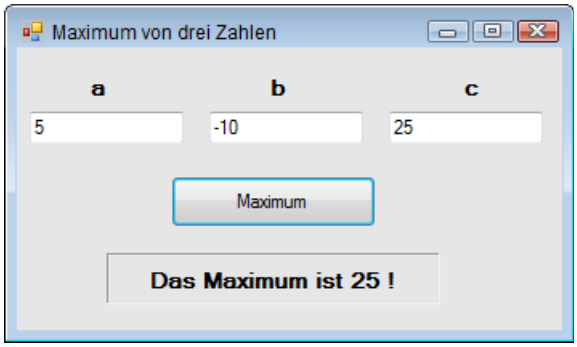
Füllen Sie den zunächst leeren Körper des Eventhandlers mit den erforderlichen Anweisungen aus, sodass der komplette Eventhandler schließlich folgendermaßen aussieht:

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

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

Test

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



Bemerkungen

Neben dem attraktiveren Outfit einer Windows-Anwendung schlagen auch noch weitere Vorteile gegenüber der tristen Konsolenanwendung deutlich zu Buche:

- So ist z.B. die Reihenfolge der Zahleneingaben ohne Bedeutung und
- Sie können bequem mittels *Tab*-Taste zwischen den Steuerelementen wechseln.

2.8.3 Schleifenanweisungen verstehen

C# offeriert Ihnen ein reichhaltiges Angebot an Schleifenanweisungen. Da der Umgang mit ihnen zum Einmaleins des Programmierens gehört, demonstriert Ihnen das vorliegende Beispiel die prinzipielle Anwendung der wichtigsten Schleifentypen (außer *foreach*-Schleife).

Ziel soll es sein, zehnmal untereinander einen Text in einer *ListBox* auszugeben, wobei fünf verschiedene Schleifenkonstruktionen gegenübergestellt werden.

Oberfläche

Alles, was Sie zum Testen brauchen, ist das mit einer *ListBox* und einigen *Button*s bestückte Startformular *Form1* (siehe Laufzeitabbildung am Schluss).

Quellcode

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

Wir beginnen mit der altbekannten *for*-Schleife:

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

```
        for (int i = 1; i<=10; i++)
            listBox1.Items.Add(i.ToString() + " Viele Wege führen nach Rom!");
    }
```

Eine der möglichen Modifikationen, wo Sie sich selbst um die Verwaltung der Zählvariablen kümmern müssen:

```
    private void button2_Click(object sender, EventArgs e)
    {
        int i = 1;                    // Ersatz für Initialisierung der Zählvariablen
        for (; i<=10; )
        {
            listBox1.Items.Add(i.ToString() + " Viele Wege führen nach Rom!");
            i++;                        // Ersatz für Aktualisierung der Zählvariablen
        }
    }
```

Die *while*-Schleife ist kopfgesteuert:

```
    private void button3_Click(object sender, EventArgs e)
    {
        int i = 1;
        while (i <= 10)
        {
            listBox1.Items.Add(i.ToString() + " Viele Wege führen nach Rom!");
            i++;
        }
    }
```

Fußgesteuert hingegen gibt sich die *do-while*-Schleife:

```
    private void button4_Click(object sender, EventArgs e)
    {
        int i = 1;
        do
        {
            listBox1.Items.Add(i.ToString() + " Viele Wege führen nach Rom!");
            i ++;
        }
        while (i <= 10);
    }
```

Das vorzeitige Verlassen der Schleife mittels *break*:

```
    private void button5_Click(object sender, EventArgs e)
    {
        for (int i = 1; ; i++)
        {
            listBox1.Items.Add(i.ToString() + " Viele Wege führen nach Rom!");
            if (i == 10) break;                                  // Abbruchbedingung
        }
    }
```

Eher nebensächlich ist das Löschen der *ListBox*:

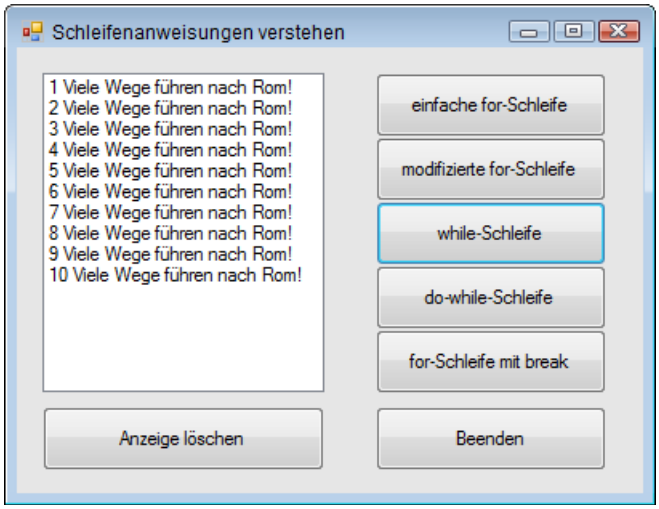
```
    private void button6_Click(object sender, EventArgs e)
    {
        listBox1.Items.Clear();
    }
```

Das Beenden des Programms:

```
    private void button7_Click(object sender, EventArgs e)
    {
        this.Close();
    }
}
```

Test

Alle fünf Schleifenvarianten produzieren ein absolut identisches Ergebnis:



2.8.4 Benutzerdefinierte Methoden überladen

In C# spricht man nicht mehr im klassischen Sinn von Funktionen und Prozeduren, sondern nur noch von Methoden mit Rückgabewert und von Methoden ohne Rückgabewert (*void*). Beschränkt man sich auf private Methoden (Aufruf innerhalb einer Klasse), so spielen diese quasi die Rolle von "Unterprogrammen", sind also strenggenommen keine Methoden mehr im Sinne der OOP. Trotzdem können auch solche Methoden überladen werden, d.h., in mehreren gleichnamigen Versionen nebeneinander existieren, die sich nur durch ihre Signatur unterscheiden.

Das vorliegende, durchaus auch praxistaugliche, Programm demonstriert drei verschiedene Überladungen der bekannten Formel zur Berechnung des Gewichts einer Kugel:

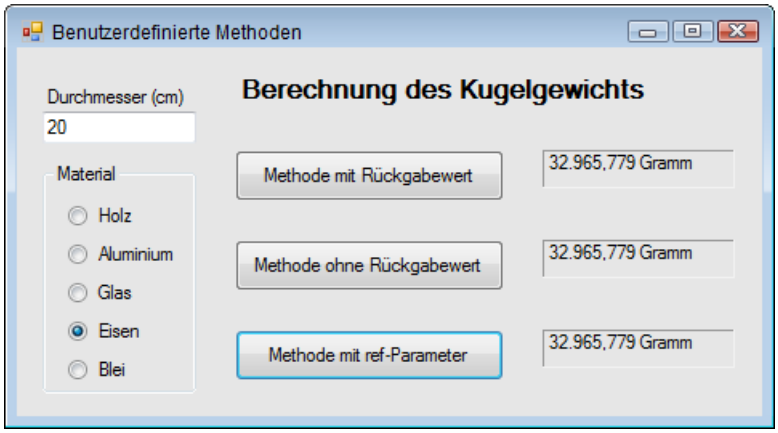
$$G = \frac{4}{3} * \pi * r^3 * \gamma$$

G = Gewicht (Gramm)
r = Radius (Zentimeter)
π = Pi = 3,14159...
γ = spezifisches Gewicht (Gramm/Kubikzentimeter)

En passant wird weiteres Einsteigerwissen wie Strukturen (*struct*), *if*-Statements, *RadioButton*-Auswahl etc. vermittelt.

Oberfläche

Auf das Startformular *Form1* setzen Sie eine *GroupBox*, die fünf verschiedene *RadioButton*s enthält, mit denen ein bestimmtes Material (Holz, Aluminium, Glas, Eisen, Blei) ausgewählt wird Weiterhin werden noch eine *TextBox*, drei *Button*s und verschiedene *Label*s benötigt (siehe folgende Laufzeitabbildung).



Quellcode

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

Globale Deklarationen:

```
        private struct Kugel       // kapselt Radius und spez. Gewicht einer Kugel
        {
            public double radius, sg;
        }

        private double gew;       // Rückgabewert für Variante 2

        private Kugel getKugel()  // Hilfsfunktion, liest Kugelwerte aus Eingabemaske
        {
            Kugel kug;
            kug.radius = Convert.ToDouble(textBox1.Text)/2;     // Kugelradius zuweisen
```

```
        // spezifisches Gewicht zuweisen:
        if (radioButton1.Checked) kug.sg = 1.4D;          // Holz
        else if (radioButton2.Checked) kug.sg = 2.7D;     // Alu
        else if (radioButton3.Checked) kug.sg = 3.0D;     // Glas
        else if (radioButton4.Checked) kug.sg = 7.87D;    // Eisen
        else kug.sg = 11.3D;                              // Blei
        return kug;
    }
```

Drei verschiedene Methodenüberladungen zur Berechnung des Kugelgewichts:

Variante 1:

```
    private double KugelGewicht(double ra, double sg)
    {
        double vol = 4 / 3D * Math.PI * Math.Pow(ra, 3);
        return (sg * vol);     // Rückgabe des Gewichts als Funktionswert
    }
```

Variante 2:

```
    private void KugelGewicht(Kugel kug)
    {
        double vol = 4 / 3D * Math.PI * Math.Pow(kug.radius, 3);
        gew = kug.sg * vol;    // das Gewicht wwird der globalen Variablen gew zugewiesen
    }
```

Variante 3:

```
    private void KugelGewicht(ref Kugel kug)
    {
        double vol = 4 / 3D * Math.PI * Math.Pow(kug.radius, 3);
        kug.sg = kug.sg * vol;  // der Übergabeparameter kug zeigt auf das Gewicht!
    }
```

Bei den folgenden Codierungen der Methode *Kugelgewicht* wird Ihnen die IntelliSense von Visual Studio jeweils eine Auswahl zwischen den drei Überladungen anbieten:

Test Variante 1:

```
    private void button1_Click(object sender, EventArgs e)
    {
        Kugel kug = getKugel();
        double r = kug.radius, sg = kug.sg;             // Radius und spez. Gewicht
        double gew = KugelGewicht(r, sg);               // Aufruf
        label1.Text = gew.ToString("#,#0.000 Gramm");   // lokale Variable gew
    }
```

Test Variante 2:

```
    private void button2_Click(object sender, EventArgs e)
    {
        Kugel kug = getKugel();
```

```
            KugelGewicht(kug);                              // Aufruf
            label2.Text = gew.ToString("#,#0.000 Gramm");   // globale Variable gew
        }
```

Test Variante 3:

```
        private void button3_Click(object sender, EventArgs e)
        {
            Kugel kug = getKugel();
            KugelGewicht(ref kug);                           // Aufruf (kug trägt Ergebnis)
            label3.Text = kug.sg.ToString("#,#0.000 Gramm");
        }
    }
```

Test

Geben Sie das Material und den Durchmesser der Kugel ein und lassen Sie sich das Gewicht anzeigen. Alle drei Methodenüberladungen liefern identische Ergebnisse (siehe obige Abbildung).

Kapitel 3

OOP-Konzepte

C# erlaubt es Ihnen, bereits ohne fundierte OOP-Kenntnisse objektorientiert zu programmieren! Davon haben Sie bereits vor der Lektüre dieses Kapitels, mehr oder weniger unbewusst, Gebrauch gemacht: Sie haben Ereignisbehandlungsroutinen (Eventhandler) geschrieben und den Objekten der visuellen Benutzerschnittstelle (Form, Steuerelemente) Eigenschaften zugewiesen bzw. deren Methoden aufgerufen.

Die Entwicklungsumgebung von Visual Studio erlaubt objektorientiertes Programmieren bereits mit einem Minimum an Vorkenntnissen. Das vorliegende Kapitel will etwas tiefer in die OOP-Problematik eindringen und präsentiert Ihnen neben einigen grundlegenden Ausführungen die für den Einstieg wichtigsten objektspezifischen Features von C# im Überblick.

HINWEIS: In C# haben Sie grundsätzlich die Möglichkeit, zwischen Klassen (sind Verweistypen) und Strukturen (sind Wertetypen) zu wählen. Strukturen (*struct*) wurden bereits im Sprachkapitel (Abschnitt 2.6.2) einführend behandelt, bieten allerdings noch weitaus mehr Möglichkeiten, die fast an die von Klassen heranreichen. Wir aber wollen uns im vorliegenden Kapitel ausschließlich mit Klassen und Objekten, den Grundelementen der OOP, beschäftigen.

3.1 Kleine Einführung in die OOP

In .NET ist alles ein Objekt! Viele Entwickler – insbesondere wenn sie mit "altem" Code zu kämpfen haben – tun sich immer noch ziemlich schwer mit OOP, weil ihnen die Komplexität einer vollständigen Anwendung zu hoch erscheint.

3.1.1 Historische Entwicklung

Im Unterschied zur objektorientierten ist die klassische strukturierte Programmierung ziemlich sprachunabhängig und hatte Zeit genug, um auch in den letzten Winkel der Programmierwelt vorzudringen.

Demgegenüber stand es um die Akzeptanz der objektorientierten Programmierung bis Anbruch des .NET-Zeitalters zu Beginn dieses Jahrtausends noch nicht zum Besten, das aber hat sich seitdem dramatisch geändert.

Strukturierte Programmierung

Gern bezeichnet man die strukturierte Programmierung auch als Vorläufer der objektorientierten Programmierung, obwohl dieser Vergleich hinkt. Richtig ist, dass sowohl strukturierte als auch objektorientierte Programmierung fundamentale Denkmuster[1] sind, die gleichberechtigt nebeneinander existieren.

Die Grundkonzepte der strukturierten Programmierung wurden beginnend mit dem Ende der Sechzigerjahre entwickelt und lassen sich mit folgenden Stichwörtern charakterisieren: hierarchische Programmorganisation, logische Programmeinheiten, zentrale Programmsteuerung, beschränkte Datenverfügbarkeit.

Ziel der strukturierten Programmierung ist es, Algorithmen so darzustellen, dass ihr Ablauf einfach zu erfassen und zu verändern ist.

Gegenstand der strukturierten Programmierung ist also die bestmögliche Anordnung von Code, um dessen Transparenz, Testbarkeit und Wiederverwendbarkeit zu maximieren[2].

Dass C# eine konsequent objektorientierte Sprache ist, bedeutet noch lange nicht, dass man damit nicht auch strukturiert programmieren könnte, im Gegenteil. Im Kapitel 2, wo sich alles um die grundlegenden sprachlichen Elemente von C# dreht, haben wir uns fast ausschließlich auf dem Boden der traditionellen strukturierten Programmierung bewegt und versucht, die OOP noch weitestgehend auszuklammern.

So haben wir es größtenteils ignoriert, dass selbst die einfachen Datentypen Objekte sind, und haben z.B. statt mit Methoden mit Funktionen und Prozeduren und statt mit Klassen mit strukturierten Datentypen (*struct*) gearbeitet. Tatsächlich können Sie aber mit OOP alles machen, was auch die strukturierte Programmierung erlaubt. Statt beispielsweise globale Variablen in einem Modul zu deklarieren, können Sie statische Klasseneigenschaften verwenden.

Um fit für die aktuellen Herausforderungen zu sein, sollten Sie deshalb – wo immer es vertretbar ist – nach objektorientierten Lösungen streben.

Objektorientiertes Programmieren

Die objektorientierte Programmierung entfaltete auf breiter Basis erst seit Ende der 80er-Jahre mit dem Beginn des Windows-Zeitalters ihre Wirkung. Sehr bekannte Vertreter objektorientierter Sprachen sind C++, Java, Smalltalk und Borland Delphi – aber auch das alte Visual Basic war bereits in vielen wesentlichen Zügen objektorientiert aufgebaut[3].

---

[1] Im Fachjargon heißt das "Paradigma".

[2] Bahnbrechendes auf diesem Gebiet leistete Prof. Niklaus Wirth mit seinen Sprachen Pascal und Modula.

[3] Mit VB.NET wurde auch Visual Basic endlich vollwertiges Mitglied der OOP-Welt.

Objektorientierte Programmierung ist ein Denkmuster, bei dem Programme als Menge von über Nachrichten kooperierenden Objekten organisiert werden und jedes Objekt Instanz einer Klasse ist.

Im Unterschied zur strukturierten Programmierung bedeutet "objektorientiert" also, dass Daten und Algorithmen nicht mehr nebeneinander existieren, sondern in Objekten zusammengefasst sind.

Während Module in der strukturierten Programmierung zwar auch Daten und Code zusammenfassen, stellen Klassen jetzt Vorlagen dar, von denen immer neue Kopien (Instanzen) angefertigt werden können. Diese Instanzen, d.h. die Objekte, kapseln den Zugriff auf die enthaltenen Daten hinter Schnittstellen (Interfaces).

Der große Vorteil der OOP ist ihre Ähnlichkeit mit den menschlichen Denkstrukturen. Dadurch wird vor allem dem Einsteiger, der bisher über keine bzw. wenig Programmiererfahrung verfügt, das Verständnis der OOP erleichtert.

HINWEIS: Die OOP verlangt eine Anpassung des Software-Entwicklungsprozesses und der eingesetzten Methoden an den Denkstil des Programmierers – nicht umgekehrt!

Die OOP ist eine der wenigen Fälle, in denen der Einsteiger gegenüber dem Profi zumindest einen kleinen Vorteil besitzt: Er ist noch nicht in der Denkweise klassischer Programmiersprachen gefangen, die dazu erziehen, in Abläufen zu denken, bei denen die in der realen Welt zu beobachtenden Abläufe Schritt für Schritt in Algorithmen umgesetzt werden, etwa um betriebliche Prozesse per Programm zu automatisieren.

Die OOP entspricht hingegen der üblichen menschlichen Denkweise, indem sie z.B. reale Objekte aus der abzubildenden Umwelt identifiziert und in ihrer Art beschreibt.

3.1.2 Grundbegriffe der OOP

Bereits im Kapitel 1hatten Sie gesehen, dass Objekte durch Eigenschaften, Methoden und Ereignisse beschrieben werden. Auf diese und andere Weise überwindet das Konzept der objektorientierten Programmierung (OOP) den prozeduralen Ansatz der klassischen strukturellen Programmierung zugunsten einer realitätsnahen Modellierung.

Bevor wir uns den Details zuwenden, sollen die wichtigsten Begriffe der objektorientierten Programmierung (OOP) zunächst allgemein, d.h. ohne Bezug auf eine konkrete Programmiersprache, erörtert werden.

Objekt

Der Programmierer versteht unter einem *Objekt* die Zusammenfassung (Kapselung) von Daten und zugehörigen Funktionalitäten. Ein solches Softwareobjekt wird auch oft benutzt, um Dinge des täglichen Lebens für Zwecke der Datenverarbeitung abzubilden. Aber das ist nur ein Aspekt, denn Objekte sind ganz allgemein Dinge, die Sie in Ihrem Code beschreiben wollen, es sind Gruppen von Eigenschaften, Methoden und Ereignissen, die logisch zusammengehören. Als Programmierer arbeiten Sie mit einem Objekt, indem Sie dessen Eigenschaften und Methoden manipulieren und auf seine Ereignisse reagieren.

Klasse

Eine *Klasse* ist nicht mehr und nicht weniger als ein "Bauplan", auf dessen Grundlage die entsprechenden Objekte zur Programmlaufzeit erzeugt werden. Gewissermaßen als Vorlage (Prägestempel) für das Objekt legt die Klasse fest, wie das Objekt auszusehen hat und wie es sich verhalten soll. Es handelt sich bei einer Klasse also um eine reine Softwarekonstruktion, die Eigenschaften, Methoden und Ereignisse eines Objekts definiert, ohne das Objekt zu erzeugen.

HINWEIS: Oft wird anstatt des Begriffs "Klasse" mit völlig gleichwertiger Bedeutung auch "Objekttyp" verwendet.

Instanz

Man erhält erst dann ein konkretes Objekt, wenn man eine *Instanz* einer Klasse bildet. Es lassen sich viele Objekte mit einer einzigen Klassendefinition erzeugen.

BEISPIEL 3.1: Instanz

Auf dem Montageband werden zahlreiche Auto-Objekte nach ein und denselben Konstruktionsvorschriften für die Klasse *Auto* gebaut. Diesen Vorgang könnte man auch als Bildung von Instanzen der Klasse *Auto* bezeichnen. Während die Klasse lediglich die Eigenschaft *Farbe* definiert, wird der konkrete Wert (rot, blau, grün ...) erst beim Erzeugen des Objekts (der Instanz) zugewiesen.

Kapselung

Klassen realisieren das Prinzip der *Kapselung* von Objekten, das es ermöglicht, die Implementierung der Klasse (der Code im Inneren) von deren Schnittstelle bzw. Interface (die öffentlichen Eigenschaften, Methoden und Ereignisse) sauber zu trennen. Durch das Verbergen der inneren Struktur werden die internen Daten und einige verborgene Methoden geschützt, sind also von außen nicht zugänglich. Die Manipulation des Objekts kann lediglich über streng definierte, über die Schnittstelle zur Verfügung gestellte öffentliche Methoden erfolgen.

Wiederverwendbarkeit

Klassen ermöglichen die *Wiederverwendbarkeit* von Code. Nachdem eine Klasse geschrieben wurde, können Sie diese an verschiedenen Stellen innerhalb einer Applikation verwenden. Klassen reduzieren somit den redundanten Code einer Anwendung, sie erleichtern außerdem die Wartung des Codes.

Vererbung

Echte Vererbung (*Implementierungsvererbung*) ermöglicht es Klassen zu definieren, die von anderen Klassen abgeleitet werden, wobei nicht nur die Schnittstelle, sondern auch der dahinter liegende Code (die Implementierung) vom Nachkommen übernommen wird.

Da es nun möglich ist, die Implementierung einer Klasse für weitere Klassen als Grundlage zu verwenden, kann man Unterklassen bilden, die alle Eigenschaften und Methoden ihrer Oberklasse erben. Diese Unterklassen können zu den geerbten Eigenschaften neue hinzufügen oder Eigenschaften der Oberklasse verstecken, indem sie diese überschreiben.

Wird von einer solchen Unterklasse ein Objekt erzeugt (also eine Instanz der Unterklasse gebildet), dann dient für dieses Objekt sowohl die Ober- als auch die Unterklasse als "Bauplan".

C# unterstützt das Überschreiben (*Overriding*) von Methoden[1] der Oberklasse mit alternativen Methoden der Unterklasse (siehe Abschnitt 3.6.2).

Polymorphie

OOP macht es möglich, ein und dieselbe Methode für ganz verschiedene Objekte zu verwenden, man nennt dies dann *Polymorphie* (Vielgestaltigkeit). Jedes dieser Objekte kann die Ausführung unterschiedlich realisieren. Für das aufrufende Objekt bleibt der Vorgang trotzdem derselbe.

BEISPIEL 3.2: Polymorphie

Die Methode *Beschleunigen* ist in einer *Fahrzeug*-Klasse definiert, welche an die Unterklassen *Auto* und *Fahrrad* vererbt. Es ist klar, dass diese Methoden in beiden Unterklassen überschrieben, d.h. völlig unterschiedlich implementiert werden müssen.

Als Polymorphie, die aufs Engste mit der Vererbung verknüpft ist, kann man also die Fähigkeit von Unterklassen bezeichnen, Eigenschaften und Methoden mit dem gleichen Namen, aber mit unterschiedlichen Implementierungen aufzurufen (siehe Abschnitt 3.6.6).

3.1.3 Sichtbarkeit von Klassen und ihren Mitgliedern

Um die Klasse bzw. ihre Mitglieder (Member, Elemente) gezielt zu verbergen oder offen zu legen, sollten Sie von den Zugriffsmodifizierern Gebrauch machen, die den Gültigkeitsbereich (bzw. die *Sichtbarkeit*) einschränken.

Klassen

Die folgende Tabelle zeigt die möglichen Einschränkungen bei der Sichtbarkeit von Klassen:

Modifizierer	Sichtbarkeit
public	Unbeschränkt. Auch von anderen Assemblierungen aus können Objekte der Klasse erstellt werden.
internal	Nur innerhalb des aktuellen Projekts. Außerhalb des Projekts ist kein Objekt dieser Klasse erstellbar. Gilt als Standard, falls kein Modifizierer vorangestellt wird.
private	Nur innerhalb einer anderen Klasse.

[1] Nicht zu verwechseln mit dem Überladen (Overloading) von Methoden (siehe 3.3.2).

Klassenmitglieder

Die folgende Tabelle zeigt die Zugriffsmöglichkeiten auf die Klassenmitglieder (Member).

Modifizierer	Sichtbarkeit
public	Unbeschränkt.
protected	Innerhalb der Klasse und der daraus abgeleiteten Klassen.
internal	Innerhalb des aktuellen Projekts.
internal protected	Innerhalb des aktuellen Projekts oder der abgeleiteten Klassen.
private	Nur innerhalb der Klasse.

Die Schlüsselwörter *private* und *public* definieren immer die beiden Extreme des Zugriffs. Betrachtet man die jeweiligen Klassen als allein stehend, so reichen diese beiden Zugriffsarten völlig aus. Mit solchen, quasi isolierten, Klassen lassen sich allerdings keine komplexeren Probleme lösen.

Um einzelne Klassen miteinander zu verbinden, verwenden Sie den mächtigen Mechanismus der Vererbung. In diesem Zusammenhang gewinnt die *protected*-Deklaration wie folgt an Bedeutung:

- Da eine abgeleitete Klasse auf die *protected*-Member zugreifen kann, sind diese Member für die abgeleitete Klasse quasi *public*.
- Ist eine Klasse nicht von einer anderen abgeleitet, kann sie nicht auf deren *protected*-Member zugreifen, da diese dann quasi *private* sind.

Mehr zu diesem Thema finden Sie im Abschnitt 3.6 (Vererbung).

3.1.4 Allgemeiner Aufbau einer Klasse

Bevor der Einsteiger seine erste Klasse schreibt, sollte er sich zunächst im einführenden Sprachkapitel 2 mit den Strukturen (*struct*, siehe Abschnitt 2.6.2) anfreunden, die in Aufbau und Anwendung starke Ähnlichkeiten zu Klassen aufweisen[1]. Auch im Aufbau von Methoden sollte er sich auskennen (Abschnitt 2.7).

Im Unterschied zu einer Struktur (Schlüsselwort *struct*) wird eine Klasse mit dem Schlüsselwort *class* deklariert. Die (stark vereinfachte) Syntax:

SYNTAX:
```
Modifizierer class Bezeichner
                                {
                                  // ... Felder
                                  // ... Konstruktoren
                                  // ... Eigenschaften
                                  // ... Methoden
                                  // ... Ereignisse
                                }
```

[1] Der wesentliche Unterschied ist der, dass Strukturen Wertetypen, Klassen hingegen Referenztypen sind.

Zur Bedeutung der (Zugriffs-)*Modifizierer* wird auf obige Tabellen verwiesen.

Im Klassenkörper haben es wir es mit "Klassenmitgliedern" (Member) wie Feldern, Konstruktoren, Eigenschaften, Methoden und Ereignissen zu tun, auf die wir noch detailliert zu sprechen kommen werden.

Die Definition der Klassenmitglieder bezeichnet man auch als *Implementation* der Klasse.

BEISPIEL 3.3: Eine einfache Klasse *CKunde* wird deklariert und implementiert.

C#
```
public class CKunde
{
   private string _anrede;          //     Feld
   private string _name;            //     dto.

 public CKunde(string anr, string nam)        // Konstruktor
 {
   _anrede = anr;
   _name = nam;
 }

 public string name             // Eigenschaft
   {
         get {return(_name); }
         set {_name = value; }
   }

   public string adresse()               // Methode
   {
         string s = _anrede + " " + _name;
         return(s);
   }
 }
```

Unsere Klasse verfügt damit über zwei Felder, eine Eigenschaft und eine Methode. Da die beiden Felder mit dem *private*-Modifizierer deklariert wurden, sind sie von außen nicht sichtbar.

3.1.5 Das Erzeugen eines Objekts

Existiert eine Klasse, so steht dem Erzeugen von Objektvariablen nichts mehr im Weg. Eine Objektvariable ist ein Verweistyp, sie enthält also nicht das Objekt selbst, sondern stellt lediglich einen Zeiger (Adresse) auf den Speicherbereich des Objekts bereit. Es können sich also durchaus mehrere Objektvariablen auf ein und dasselbe Objekt beziehen. Wenn eine Objektvariable den Wert *null* enthält, bedeutet das, dass sie momentan "ins Leere" zeigt, also kein Objekt referenziert.

Unter der Voraussetzung, dass eine gültige Klasse existiert, verläuft der Lebenszyklus eines Objekts in Ihrem Programm in folgenden Etappen:

- Referenzierung (eine Objektvariable wird deklariert, sie verweist momentan noch auf *null*)
- Instanziierung (die Objektvariable zeigt jetzt auf einen konkreten Speicherplatzbereich)
- Initialisierung (die Datenfelder der Objektvariablen werden mit Anfangswerten gefüllt)
- Arbeiten mit dem Objekt (es wird auf Eigenschaften und Methoden des Objekts zugegriffen, Ereignisse werden ausgelöst)
- Zerstören des Objekts (das Objekt wird dereferenziert, der belegte Speicherplatz wird wieder freigegeben)

Werfen wir nun einen genaueren Blick auf die einzelnen Etappen.

Referenzieren und Instanziieren

Es stehen zwei Varianten zur Verfügung.

In *Variante 1* wird pro Schritt eine Anweisung verwendet:

SYNTAX:
```
Modifizierer Klasse Object;
Object = new Klasse(Parameter);
```

BEISPIEL 3.4: Ein Objekt *kunde1* wird referenziert und erzeugt.

C#
```
private CKunde kunde1;                    // Referenzieren
kunde1 = new CKunde1();                   // Erzeugen
```

In *Variante2*, der Kurzform, sind beide Schritte in einer Anweisung zusammengefasst, d.h., das Objekt wird zusammen mit seiner Deklaration erzeugt.

SYNTAX:
```
Klasse Object = new Klasse();
```

BEISPIEL 3.5: Das Äquivalent zum Vorgängerbeispiel.

C#
```
private CKunde kunde1 = new CKunde();
```

Dem Klassenbezeichner (*Klasse*) müsste genauer genommen noch der Name der Klassenbibliothek (bzw. Name des Projekts) vorangestellt werden, doch dies wird unter Visual Studio nicht erforderlich sein, da der entsprechende Namensraum (*Namespace*) bereits automatisch eingebunden wurde (*using*-Anweisung).

Obwohl die Kurzform sehr eindrucksvoll ist, können Sie hier keine Fehlerbehandlung (*try...catch*-Block) durchführen. Diese Einschränkung macht diese Art von Deklaration weniger nützlich.

Empfehlenswert ist also fast immer das getrennte Deklarieren und Erzeugen[1].

[1] Aus Platzgründen halten sich die Autoren leider nicht immer an diese Empfehlung.

BEISPIEL 3.6: Eine mögliche Fehlerbehandlung

```
C#
private CKunde kunde1;
try
{
     kunde1 = new CKunde();
}
catch(Exception ex)
{
 MessageBox.Show(ex.Message);
}
```

Klassische Initialisierung

Statt die Anfangswerte einzeln zuzuweisen, können Sie diese zusammen mit einem Konstruktor übergeben.

BEISPIEL 3.7: Das Objekt *kunde1* wird erzeugt (Standardkonstruktor), zwei Eigenschaften werden einzeln zugewiesen.

```
C#
CKunde kunde1 = new CKunde();
kunde1.anrede = "Frau"; kunde1.name = "Müller";
```

BEISPIEL 3.8: Das Objekt *kunde1* wird erzeugt und mit einem Konstruktor initialisiert.

```
C#
CKunde kunde1 = new CKunde("Frau", "Müller");
```

Weitere Einzelheiten entnehmen Sie dem Abschnitt 3.5.1.

Objekt-Initialisierer

Ab C# 3.0 wurden – vor allem in Hinblick auf die in der neuen LINQ-Technologie erforderlichen anonymen Typen (siehe Abschnitt 2.2) – so genannte *Objektinitialisierer* eingeführt. Damit können nun öffentliche Eigenschaften und Felder von Objekten ohne das explizite Vorhandensein des jeweiligen Konstruktors in beliebiger Reihenfolge initialisiert werden. Das Initialisieren geschieht über geschweifte Klammern, in denen die einzelnen Felder/Eigenschaften des Objekts mit Werten belegt werden.

BEISPIEL 3.9: Gegeben ist eine Klasse *CPerson*:

```
C#
public class CPerson
{
      public string Name;
      public string Strasse;
      public int PLZ;
      public string Ort;
}
```

BEISPIEL 3.9: Gegeben ist eine Klasse *CPerson*:

Das Erzeugen und Initialisieren einer Instanz von *CPerson* bedarf keines Konstruktors:

```
CPerson person1 = new CPerson { Name = "Müller", Strasse = "Am Waldesrand 7", PLZ = 12345,
                                Ort = "Musterhausen" };
```

Arbeiten mit dem Objekt

Wie Sie bereits wissen, erfolgt der Zugriff auf Eigenschaften und Methoden eines Objekts, indem der Name des Objekts mit einem Punkt (.) vom Namen der Eigenschaft/Methode getrennt wird.

SYNTAX: *Objekt.Eigenschaft|Methode()*

BEISPIEL 3.10: Die Eigenschaft *guthaben* des Objekts *kunde1* wird zugewiesen und die Methode *adresse* aufgerufen

C#
```
kunde1.guthaben = 10;
label1.Text = kunde1.adresse();
```

Zerstören des Objekts

Wenn Sie das Objekt nicht mehr brauchen, können Sie die Objektvariable auf *null* setzen.

BEISPIEL 3.11: Der *kunde1* wird in die ewigen Jagdgründe befördert

C#
```
kunde1 = null;
```

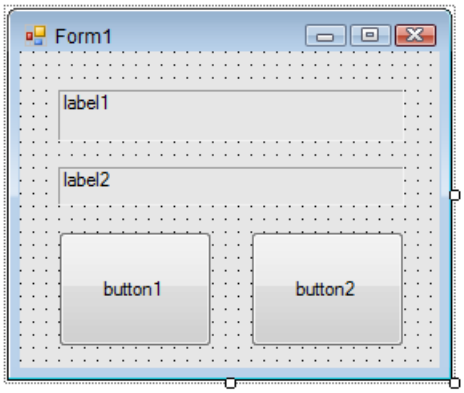
Das Objekt wird allerdings erst dann zerstört, wenn der Garbage Collector festgestellt hat, dass es nicht länger benötigt wird (siehe Abschnitt 3.5.2).

3.1.6 Einführungsbeispiel

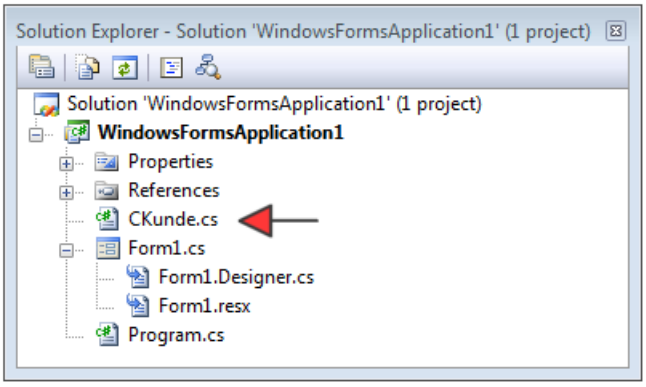
Raus aus dem muffigen Hörsaal, lasst uns endlich einmal selbst eine einfache Klasse erstellen und beschnuppern!

Vorbereitungen

- Öffnen Sie ein neues Projekt (z.B. mit dem Namen "Kunden") als Windows Forms-Anwendung.
- Auf das Startformular (*Form1*) platzieren Sie zwei *Label*s und zwei *Button*s.



- Nachdem Sie den Menüpunkt *Projekt|Klasse hinzufügen...* gewählt haben, geben Sie im Dialogfenster den Namen *CKunde.cs* ein und klicken "Hinzufügen". Der Projektmappen-Explorer zeigt jetzt die neue Klasse:



Sie müssen eine Klasse nicht unbedingt in einem eigenen Klassenmodul definieren, Sie könnten die Klasse z.B. auch zum bereits vorhandenen Code des Formulars (*Form1.cs*) hinzufügen. Das Verwenden eigener Klassenmodule (idealerweise eins pro Klasse) steigert aber die Übersichtlichkeit des Programmcodes und erleichtert dessen Wiederverwendbarkeit.

Klasse definieren

Im Code-Fenster *CKunde.cs* ist bereits der Rahmencode für unsere Klasse vorbereitet:

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

```
namespace WindowsFormsApplication1
{
    class CKunde
    {
    }
}
```

Tragen Sie dann in den Klassenkörper die Implementierung der Klasse ein, sodass der komplette Code der Klasse schließlich folgendermaßen ausschaut:

```
public class CKunde
{
    private const char LF = (char) 10;  //  private Konstante (Zeilenumbruch)

    public string anrede;          //    öffentliches Feld
    public string name;            //      dto.
    public int plz;                //      dto.
    public string ort;             //      dto.
    public bool stammkunde;        //      dto.
    public decimal guthaben;       //      dto.

    public string adresse()                 //    öffentliche Methode
    {
      string s = anrede + " " + name + LF + plz.ToString() + " " + ort;
      return(s);
    }

    public void addGuthaben(decimal betrag)   // öffentliche Methode
    {
        if (stammkunde) guthaben += betrag;
    }
}
```

Bemerkungen

- Die Klasse verfügt über sechs "einfache" Eigenschaften, und zwar sind das alle als *public* deklarierten Variablen, die man auch als "öffentliche Felder" bezeichnet. Die Betonung liegt hier auf "einfach", da wir später noch lernen werden, wie man "richtige" Eigenschaften programmiert.
- Weiterhin verfügt die Klasse über zwei *Methoden*. Die *string*-Methode *adresse()* liefert einen Rückgabewert, nämlich die komplette Anschrift.
- Die *void*-Methode *addGuthaben* hingegen liefert keinen Wert zurück, sie erhöht den Wert des *guthaben*-Felds bei jedem Aufruf um 50 €.
- Die private Konstante *LF* wird von der Methode *adresse()* für das Einfügen des Zeilenumbruchs benötigt.

Objekt erzeugen und initialisieren

Wechseln Sie nun in das Code-Fenster von *Form1*.

Auf Klassenebene deklarieren Sie eine Objektvariable *kunde1*:

```
private CKunde kunde1;                // Objekt referenzieren
```

Dem linken Button geben Sie die Beschriftung "Objekt erzeugen und initialisieren" und belegen das *Click*-Ereignis wie folgt:

```
private void button1_Click(object sender, EventArgs e)
{
    kunde1 = new CKunde();          // Objekt erzeugen

    // Objektfelder initialisieren:
    kunde1.anrede = "Herr";
    kunde1.name = "Müller";
    kunde1.plz = 12345;
    kunde1.ort = "Berlin";
    kunde1.stammkunde = true;
}
```

Objekt verwenden

Hinterlegen Sie nun den rechten Button mit der Beschriftung "Methoden und Eigenschaften verwenden" wie folgt:

```
private void button2_Click(object sender, System.EventArgs e)
{
    label1.Text = kunde1.adresse();  // erste Methode aufrufen
    kunde1.addGuthaben(50M);         // zweite Methode aufrufen
    label2.Text = "Guthaben ist " + kunde1.guthaben.ToString("C");     // Eigenschaft lesen
}
```

Unterstützung durch die IntelliSense

Sie haben beim Eintippen des Quelltextes (insbesondere im Code-Fenster von *Form1*) bereits gemerkt, dass Sie durch die IntelliSense von Visual Studio eifrigst unterstützt werden.

Die IntelliSense weist Sie z.B. auf die verfügbaren Klassenmitglieder (Eigenschaften und Methoden) hin und ergänzt den Quellcode automatisch, wenn Sie auf den gewünschten Eintrag doppelklicken.

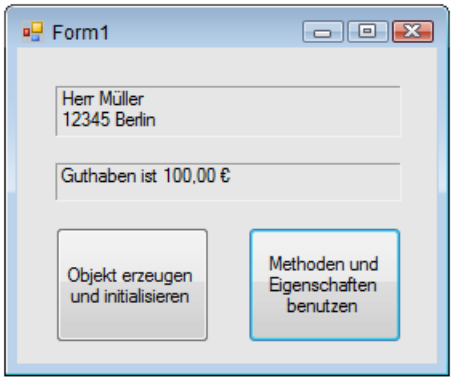
```
// Objekt erzeugen und initialisieren:
private void button1_Click(object sender, System.EventArgs e)
{
    kunde1 = new CKunde();
    kunde1.
    kund                                    er";
    kund
    kund                                    n";
    kund                                     true;
}
// Method                                  aften benutzen:
private                                     ick(object sender, System.EventArgs e)
{
    labe                                    1.Adresse();     // erste Methode aufrufen
    kund                                    50M);          // zweite Methode aufrufen
```

addGuthaben
Adresse
anrede
Equals
GetHashCode
GetType
guthaben
name
ort
plz

Falls das gewünschte Klassenmitglied nicht erscheint, müssen Sie sofort stutzig werden und es keinesfalls mit dem gewaltsamen Eintippen des Namens versuchen, denn dann gibt es wahrscheinlich einen Fehler beim Kompilieren. Überprüfen Sie stattdessen lieber nochmals die Klassendeklaration, z.B. ob vielleicht nicht doch der *public*-Modifizierer vergessen wurde.

Objekt testen

Nun ist es endlich so weit, dass Sie Ihr erstes eigenes C#-Objekt vom Stapel lassen können. Unmittelbar nach Programmstart betätigen Sie den linken Button und danach den rechten. Durch mehrmaliges Klicken auf den zweiten Button wird sich das Guthaben des Kunden Müller in 50-€-Schritten erhöhen.



Falls Sie zu voreilig gewesen sind und unmittelbar nach Programmstart den zweiten statt den ersten Button gedrückt haben, stürzt Ihnen das Programm mit der Laufzeit-Fehlermeldung "Der Objektverweis wurde nicht auf eine Objektinstanz festgelegt." ab.

Bemerkungen

Unsere Klasse funktioniert nach außen hin zwar ohne erkennbare Mängel, ist hinsichtlich ihrer inneren Konstruktion aber keinesfalls als optimal zu bezeichnen. Wir haben deshalb keinerlei Grund, uns zufrieden zurückzulehnen, denn das uns unter C# zur Verfügung stehende OOP-Instrumentarium wurde von uns bei weitem noch nicht ausgeschöpft.

- Beispielsweise haben wir nur "einfache" Eigenschaften, nämlich *public*-Felder verwendet, was eigentlich eine schwere Sünde in den Augen der OOP-Puristen ist (siehe Abschnitt 3.2.1).
- Weiterhin war das Initialisieren der Eigenschaften über mehrere Codezeilen ziemlich mühselig (von einem hilfreichen Konstruktor haben wir noch keinerlei Gebrauch gemacht, siehe Abschnitt 3.5.1).
- Außerdem wird eine Klasse erst dann so richtig effektiv, wenn wir davon nicht nur eine, sondern mehrere Instanzen (sprich Objekte) ableiten. Diese wiederum kann man ziemlich elegant in so genannten Auflistungen (Collections) verwalten (siehe Kapitel 5).

Doch zur Beseitigung dieser und anderer Unzulänglichkeiten kommen wir erst später. Ein weiteres Problem, was uns unter den Nägeln brennt, duldet keinen weiteren Aufschub und wir wollen es deshalb gleich im folgenden Abschnitt behandeln.

3.2 Eigenschaften

Eigenschaften bestimmen die statischen Attribute eines Objekts, sie leiten sich von dessen *Zustand* ab, wie er in den Zustandsvariablen (Objektfeldern) gespeichert ist. Im Unterschied zu den Methoden, die von allen Instanzen der Klasse gemeinsam genutzt werden, sind die den Eigenschaften zugewiesenen Werte für alle Objekte einer Klasse meist unterschiedlich.

3.2.1 Eigenschaften mit Zugriffsmethoden kapseln

Von den im Objekt enthaltenen Feldern sind die *public*-Felder als "einfache" Eigenschaften zu betrachten.

In unserem Beispiel hatten wir für die Klasse *CKunde* solche "einfachen" Eigenschaften als *public*-Variable deklariert. Das allerdings ist nicht die "feine Art" der objektorientierten Programmierung, denn das Veröffentlichen von Feldern widerspricht dem hochgelobten Prinzip der Kapselung und erlaubt keinerlei Zugriffskontrolle wie z.B. Wertebereichsüberprüfung oder die Vergabe von Lese- und Schreibrechten.

Idealerweise sind deshalb in einem Objekt nur private Felder enthalten, und der Zugriff auf diese wird durch Accessoren (Zugriffsmethoden) gesteuert.

In diesem Sinn ist eine *Eigenschaft* gewissermaßen ein Mittelding zwischen Feld und Methode. Sie verwenden die Eigenschaft wie ein öffentliches Feld. Vom Compiler aber wird der Feldzugriff in den Aufruf von Accessoren – das sind spezielle Zugriffsmethoden auf private Felder – übersetzt. Doch schauen wir uns das Ganze lieber in der Praxis an.

Deklarieren von Eigenschaften

Eigenschaften werden ähnlich wie öffentliche Methoden deklariert. Innerhalb der Deklaration implementieren Sie für den Lesezugriff eine *set*- und für den Schreibzugriff eine g*et*-Zugriffsmethode. Während die *get*-Methode ihren Rückgabewert über *return* liefert, erhält die *set*-Methode den zu schreibenden Wert über *value*.

SYNTAX:
```
Modifizierer Datentyp Eigenschaftsname
{
    get
    {
        // hier Lesezugriff auf priv. Felder implementieren
        return(privatesFeld);
    }
    set
    {
        // hier Schreibzugriff auf priv. Felder implementieren
        privatesFeld = value;
    }
}
```

Wir wollen nun unser Beispiel mit "echten" Eigenschaften ausstatten. Dazu werden zunächst die public-Felder in *private* verwandelt und durch Voranstellen von "_" umbenannt, um Namenskonflikte mit den gleichnamigen Eigenschafts-Deklarationen zu vermeiden.

Der Schreibzugriff auf die Eigenschaft *anrede* wird so kontrolliert, dass nur die Werte "Herr" oder "Frau" zulässig sind.

```
public class CKunde
{
    private string _anrede;          //    privates Feld
    private string _name;            //      dto.
    ...

    public string anrede
    {
        get {return(_anrede); }
        set
        {
            if (value == "Herr" || value == "Frau") _anrede = value;
            else MessageBox.Show("Die Anrede '" + value + "' ist nicht zulässig!");
        }
    }

    public string name
    {
        get {return(_name); }
        set {_name = value; }
    }
```

```
    ...
  }
}
```

Zugriff

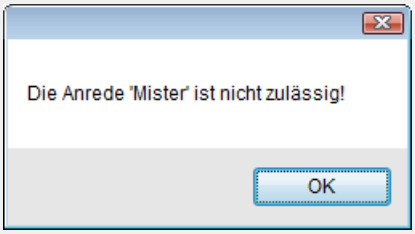
Wenn Sie ein Objekt verwenden, merken Sie auf Anhieb natürlich nicht, ob es noch über "einfache" oder schon über "richtige" Eigenschaften verfügt, es sei denn, die in die *get*- bzw. *set*-Methoden eingebauten Zugriffsbeschränkungen werden verletzt und Sie erhalten entsprechende Fehlermeldungen.

BEISPIEL 3.12: Sie wollen die Anrede "Mister" zuweisen, was zu einem Laufzeitfehler führt.

C#

```
kunde1 = new CKunde();
kunde1.anrede = "Mister";      // Fehler!
```

Ergebnis



Bemerkung

- Beim Schreiben des Quellcodes in der Entwicklungsumgebung Visual Studio merken Sie den "feinen" Unterschied zwischen "einfachen" und "richtigen" Eigenschaften, denn die IntelliSense zeigt dafür unterschiedliche Symbole[1].
- In unserem Beispiel verhält sich nur die Eigenschaft *anrede* "intelligent", d.h., sie unterliegt einer Zugriffskontrolle. Bei den übrigen Eigenschaften erfolgt lediglich eine 1:1-Zuordnung zu den privaten Feldern. Hier sollte man nicht "päpstlicher als der Papst" sein und es bei den ursprünglichen *public*-Feldern belassen. Wir aber haben diesen (eigentlich sinnlosen) Aufwand nur wegen des Lerneffekts betrieben.

3.2.2 Berechnete Eigenschaften

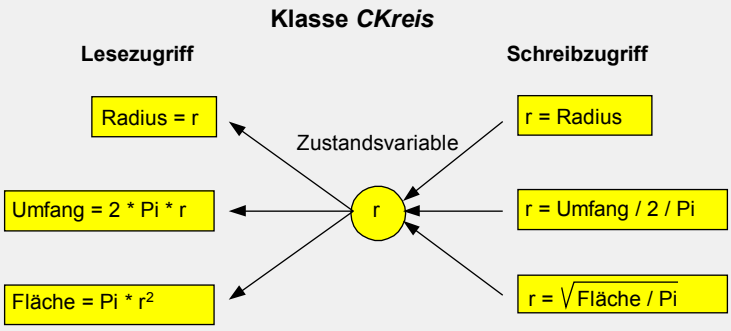
Mit Zugriffsmethoden lässt sich weit mehr anstellen, als nur den Zugriff auf private Felder der Klasse zu kontrollieren. So können z.B. innerhalb der Methode komplexe Berechnungen mit den Feldern (die man auch *Zustandsvariablen* nennt) und den übergebenen Parametern ausgeführt werden.

[1] Probieren Sie das bitte selbst aus!

BEISPIEL 3.13: Berechnete Eigenschaften

C#

Eine Klasse *CKreis* hat die Eigenschaften *radius*, *umfang* und *fläche*. In der einzigen Zustandsvariablen *r* braucht aber nur der Radius abgespeichert zu werden, da sich die übrigen Eigenschaften aus *r* berechnen lassen (*get* = Lesezugriff) bzw. umgekehrt (*set* = Schreibzugriff).




```
public class CKreis
{
    private double r;                    // das einzige Feld (Zustandsvariable)
```

Die Eigenschaft *radius*:

```
    public string radius
    {
       get {return (r.ToString("#,#0.00")); }
       set
       {
           if (value != "")  r = Convert.ToDouble(value);
           else r = 0;
       }
    }
```

Die Eigenschaft *umfang*:

```
    public string umfang
    {
        get {return (2 * Math.PI * r).ToString("#,#0.00"); }
        set
        {
            if (value != "")  r = Convert.ToDouble(value) / 2 / Math.PI;
            else r = 0;
        }
    }
```

Die Eigenschaft *fläche*:

```
    public string fläche
    {
```

BEISPIEL 3.13: Berechnete Eigenschaften

```
        get {return (Math.PI * Math.Pow(r, 2)).ToString("#,#0.00");}
        set
        {
            if (value != "") r = Math.Sqrt(Convert.ToDouble(value) / Math.PI);
            else r = 0;
        }
    }
}
```

Das komplette Programm finden Sie im Praxisbeispiel

► 3.9.1 Eigenschaften sinnvoll kapseln

3.2.3 Lese-/Schreibschutz

Es kommt häufig vor, dass bestimmte Felder bzw. Eigenschaften nur gelesen oder nur geschrieben werden dürfen.

Für Felder kann man einen Schreibschutz einfach durch Voranstellen des *readonly*-Modifizierers realisieren.

BEISPIEL 3.14: Ein schreibgeschütztes öffentliches Feld wird deklariert und initialisiert.

C#
```
public readonly double mwst = 0.19;
```

HINWEIS: Außer beim Deklarieren kann man ein ReadOnly-Feld auch in einem Konstruktor initialisieren! Genau dadurch unterscheidet sich das *readonly*- vom *const*-Schlüsselwort, denn *readonly*-Felder können – abhängig vom verwendeten Konstruktor – über unterschiedliche Werte verfügen.

Verwendet man statt öffentlicher Felder "richtige" Eigenschaften, so ist für eine Zugriffsbeschränkung keinerlei zusätzlicher Aufwand erforderlich – im Gegenteil:

HINWEIS: Um eine Eigenschaft allein für den Lese- bzw. Schreibzugriff zu deklarieren, lässt man einfach die *get*- bzw. die *set*-Zugriffsmethode weg.

BEISPIEL 3.15: In der *CKunde*-Klasse soll das Guthaben für den Schreibzugriff gesperrt werden

C#
Das klingt sicher logisch, da zur Erhöhung des Guthabens bereits die Methode *addGuthaben* existiert.

```
public decimal guthaben
{
    get {return(_guthaben); }
}
```

BEISPIEL 3.15: In der *CKunde*-Klasse soll das Guthaben für den Schreibzugriff gesperrt werden

Ergebnis

In der Entwicklungsumgebung von Visual Studio wird nun der Versuch abgewiesen, dieser Eigenschaft einen Wert zuzuweisen:

```
kunde1.stammkunde = true;
kunde1.guthaben = 10M;
```

Einer Eigenschaft oder einem Indexer 'Kunden.CKunde.guthaben' kann nicht zugewiesen werden -- sie sind schreibgeschützt

3.2.4 Property-Accessoren

Es möglich, den Zugriff auf *get*- oder *set*- Accessoren von Eigenschaften zu beschränken. Meist ist dies nur für den *set*-Accessor sinnvoll, während der *get*-Accessor in der Regel öffentlich bleibt.

BEISPIEL 3.16: Property-Accessoren

C#

Eine Eigenschaft mit *get*- und *set*-Accessoren. Der *get*-Accessor besitzt die gleiche Sichtbarkeit wie die *KontoNummer*-Eigenschaft, während der *set*-Accessor nur einen *protected*-Zugriff erlaubt.

```
public string KontoNummer
{
    get
    {
        return _knr;
    }
    protected set
    {
        _knr = value;
    }
}
```

3.2.5 Statische Felder/Eigenschaften

Mitunter gibt es Felder bzw. Eigenschaften, deren Werte für alle aus der Klasse instanziierten Objekte identisch sind und die deshalb nur einmal in der Klasse gespeichert zu werden brauchen.

HINWEIS: Statische Felder/Eigenschaften (*Klasseneigenschaften*) werden mit dem Schlüsselwort *static* deklariert.

Statische Eigenschaften bzw. öffentliche Felder können benutzt werden, ohne dass dazu eine Objektvariable deklariert und ein Objekt instanziiert werden müsste! Es genügt das Voranstellen des Klassenbezeichners.

BEISPIEL 3.17: Die Klasse *CKunde* soll zusätzlich ein öffentliches Feld (bzw. eine "einfache" Eigenschaft) *rabatt* bekommen, die für jedes Kundenobjekt immer den gleichen Wert hat.

C#
```
public class CKunde
{
    ...
    public static double rabatt;
    ...
}
```

Der Zugriff ist sofort über den Klassenbezeichner möglich, ohne dass dazu eine Objektvariable erzeugt werden müsste.

BEISPIEL 3.18: Allen Kunden wird ein Rabatt von 15% zugewiesen.

C#
```
CKunde.rabatt = 0.15;
```

Vielen Umsteigern, die aus der strukturierten Programmierung kommen, bereitet es Schwierigkeiten, auf ihre geliebten globalen Variablen zu verzichten, mit denen sie bequem Werte zwischen verschiedenen Programmmodulen austauschen konnten. Genau hier bieten sich statische Eigenschaften bzw. öffentliche statische Felder an, die z.B. in einer extra für derlei Zwecke angelegten Klasse *CAllerlei* abgelegt werden könnten.

BEISPIEL 3.19: Die Klassen *Form1* und *Form2* greifen für allgemeine Berechnungen auf eine statische Eigenschaft *MWSt* der Klasse *CAllerlei* zu.

C#
```
public class CAllerlei
{
    private static double _mwst;

    public static double MWSt
    {
        get { return (_mwst);}
        set { _mwst = value; }
    }
    ...
```

Hier könnten z.B. auch nichtstatische Klassenmitglieder eingefügt werden, die natürlich auch auf das statische Feld *_mwst* zugreifen dürfen.

```
}
```

Sowohl von *Form1* als auch von *Form2* aus kann direkt auf die statische Eigenschaft *MWst* zugegriffen werden, eine Instanz von *CAllerlei* braucht dazu nicht erzeugt zu werden:

Zuweisen der Mehrwertsteuer in *Form1*:

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

BEISPIEL 3.19: Die Klassen *Form1* und *Form2* greifen für allgemeine Berechnungen auf eine statische Eigenschaft *MWSt* der Klasse *CAllerlei* zu.

```
        CAllerlei.MWSt = 0.19;
    ...
}
```

Anzeige der Mehrwertsteuer in *Form2*:

```
public partial class Form2 : Form
{
    ...
    private void button1_Click(object sender, EventArgs e)
    {
        textBox1.Text = CAllerlei.MWSt.ToString();
    }
}
```

Es versteht sich, dass innerhalb der *get*- bzw. *set*-Accessoren statischer Eigenschaften nur auf statische Klassenmitglieder (Felder) zugegriffen werden kann. Gleiches gilt übrigens auch für statische Methoden (siehe 3.3.3).

Konstante Felder

Obwohl ein Feld nicht als *static const* deklariert werden kann, verhält sich ein *const*-Feld im Wesentlichen statisch. Deshalb kann auf *const*-Felder mit der gleichen Notation wie bei *static*-Feldern zugegriffen werden (eine mit *new* erzeugte Objektinstanz ist nicht erforderlich).

BEISPIEL 3.20: Deklaration einer öffentlichen Konstanten und Zugriff

C#

```
public class CAllerlei
{
    public const int anzahl = 50;
    ...
}
```

Der Zugriff ist direkt möglich:

```
public partial class Form1 : Form
{
    ...
    n = CAllerlei.anzahl;
    ...
}
```

Worin unterscheiden sich denn dann statische und konstante Felder? Der wesentliche Unterschied ist der, dass man den Wert statischer Felder beim Erzeugen eines Objekts (durch Aufruf unterschiedlicher Konstruktoren) oder zur Laufzeit (durch Methodenaufrufe) ändern kann, bei konstanten Feldern geht das natürlich nicht.

3.2.6 Einfache Eigenschaften automatisch implementieren

Wie bereits im Abschnitt 3.2.1 erwähnt, gehört es zum schlechten Programmierstil, wenn *public*-Variablen quasi die Rolle von einfachen Eigenschaften übernehmen. Getreu der Devise "Hauptsache es funktioniert" ist es aber für den schreibfaulen Programmierer oft der bequemere Weg, erspart er sich damit doch viele Zeilen stupiden Codes. Automatisch implementierte Eigenschaften befreien den Programmierer aus diesem Zwiespalt, sie benötigen in der Regel auch nur eine einzige Codezeile, trotzdem erfolgt im Hintergrund eine exakte Implementierung mit *get*- und *set*-Zugriffsmethoden.

BEISPIEL 3.21: Drei Varianten für eine einfache Eigenschaft *Nachname.*

C#

1. Der kurze, aber schlechte Programmierstil:

```
public class CKunde
{
    public string Nachname;
    ...
}
```

2. Die exakte, aber umständliche Schreibweise:

```
public class CKunde
{
    private string _nachname;
    public string Nachname
    {
        get {return (_nachname);}
        set {_nachname = value;}
    }
    ...
}
```

3. Die Eigenschaft wird automatisch und sauber implementiert:

```
public class CKunde
{
    public string Nachname { get; set; }
    ...
}
```

Wie Sie sehen, ist nur eine einzige Codezeile erforderlich: Der Compiler erstellt hier ein privates, anonymes dahinter liegendes Feld (analog zu *_nachname* bei der ersten Variante), das nur durch die *get*- und *set*-Accessoren aufgerufen werden kann.

HINWEIS: Soll die Eigenschaft schreibgeschützt sein, so legen Sie einfach einen privaten *set*-Accessor fest.

BEISPIEL 3.22: Die automatisch implementierte Eigenschaft *Nachname* ist schreibgeschützt.

C#
```
public string Nachname { get; private set; }
```

HINWEIS: Es sei hier nochmals betont, dass sich nur einfache Eigenschaften automatisch implementieren lassen, keine berechneten Eigenschaften!

3.3 Methoden

Methoden bestimmen die dynamischen Attribute eines Objekts, also sein Verhalten. Eine Methode ist eine Funktion, die im Körper der Klasse implementiert ist.

3.3.1 Öffentliche und private Methoden

Bereits im Kapitel 2 haben wir gelernt, wie man Methoden programmiert. Jetzt wollen wir noch etwas nachhaken und den Fokus auf die Methoden richten, die in unseren selbst programmierten Klassen zum Einsatz kommen.

Genau wie das bei "richtigen" Eigenschaften der Fall ist, arbeiten in einer sauber programmierten Klasse alle Methoden ausschließlich mit privaten Feldern (Zustandsvariablen) zusammen.

HINWEIS: Wenn Sie eine Methode als *private* deklarieren, ist sie nur innerhalb der Klasse sichtbar, und es handelt sich um keine Methode im eigentlichen Sinn der OOP, sondern eher um eine Funktion/Prozedur im herkömmlichen Sinn.

BEISPIEL 3.23: Die Methoden *adresse* und *addGuthaben* arbeiten mit sechs privaten Feldern zusammen

C#
```
public class CKunde
{
    private string _anrede;          //    privates Feld
    private string _name;            //      dto.
    private int _plz;                //      dto.
    private string _ort;             //      dto.
    private bool _stammkunde;        //      dto.
    private decimal _guthaben;       //      dto.

    public string adresse()          //    öffentliche Methode
    {
        string s = _anrede + " " + _name  + _plz.ToString() + " " + _ort;
        return(s);
    }

    public void addGuthaben(decimal betrag)   // öffentliche Methode
    {
```

BEISPIEL 3.23: Die Methoden *adresse* und *addGuthaben* arbeiten mit sechs privaten Feldern zusammen

```
        if (stammkunde) _guthaben += betrag;
    }
}
```

Deklaration und Aufruf:

```
CKunde kunde1 = new CKunde();
...
label1.Text = kunde1.adresse();   // erste Methode aufrufen
kunde1.addGuthaben(50M);          // zweite Methode aufrufen
```

3.3.2 Überladene Methoden

Obwohl wir bereits in Abschnitt 2.7.5 ganz allgemein auf dieses Thema eingegangen sind, soll es hier nochmals im OOP-Kontext diskutiert werden.

Innerhalb des Klassenkörpers dürfen zwei und mehr gleichnamige Methoden konfliktfrei nebeneinander existieren, wenn sie eine unterschiedliche Signatur (Reihenfolge und Datentyp der Übergabeparameter) besitzen.

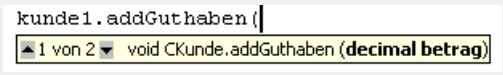
BEISPIEL 3.24: Überladene Methoden

C#

Zwei überladene Versionen einer Methode in der Klasse *CKunde*, die erste hat nur den Nettobetrag als Parameter die zweite den Bruttobetrag und die Mehrwertsteuer.

```
public void addGuthaben(decimal betrag)              // erste Überladung
{
  if (stammkunde) _guthaben += betrag;
}

public void addGuthaben(decimal brutto, decimal mwst)   // zweite Überladung
{
  _guthaben += Convert.ToDecimal(brutto/(1 + mwst));
}
```

Wenn Sie diese Methoden verwenden wollen, so fällt die Auswahl im Code-Fenster leicht:



3.3.3 Statische Methoden

Genauso wie die unter 3.2.5 erläuterten *statischen Eigenschaften* können *statische Methoden* (auch als *Klassenmethoden* bezeichnet) ohne Verwendung eines Objekts aufgerufen werden. Statische Methoden eignen sich z.B. gut für diverse Formelsammlungen (ähnlich *Math*-Klassenbibliothek). Auch können Sie damit auf private statische Klassenmitglieder zugreifen.

HINWEIS: Der Einsatz statischer Methoden für relativ einfache Aufgaben ist bequemer und ressourcenschonender als das Arbeiten mit Objekten, die Sie jedes Mal extra instanziieren müssten.

BEISPIEL 3.25: Wir bauen eine Klasse, in der wir wahllos einige von uns häufig benötigte Berechnungsformeln verpacken.

C#
```
public class CMeineFormeln
{
    public static double kreisUmfang(double radius)
    {
       return (2 * Math.PI * radius);
    }

    public static double kugelVolumen(double radius)
    {
       return ( 4 / 3.0 * Math.PI * Math.Pow(radius, 3));
    }

    public static decimal netto(decimal brutto, double mwst)
    {
        return(brutto/ Convert.ToDecimal(1 + mwst));
    }

   // .... weitere Methoden
}
```

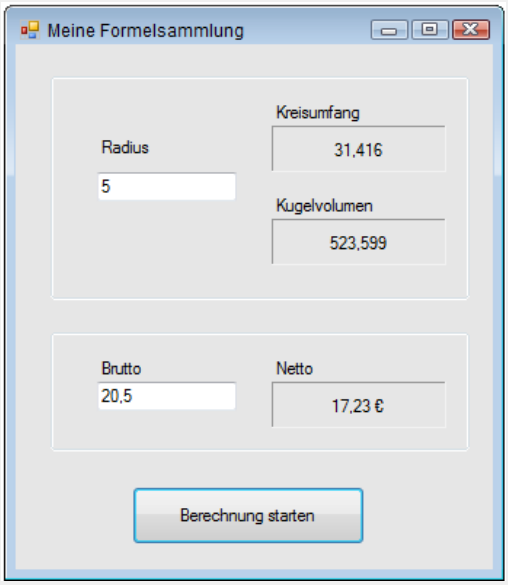
Der Zugriff von außerhalb ist absolut problemlos, weil man sich nicht mehr um das lästige Instanziieren einer Objektvariablen kümmern muss.

BEISPIEL 3.26: Die statischen Methoden der Klasse *meineFormeln* werden in einer Eingabemaske aufgerufen.

C#
```
private void button1_Click(object sender, EventArgs e)
{
     double r = Convert.ToDouble(textBox1.Text);              // Kreisradius konvertieren

     label1.Text = CMeineFormeln.kreisUmfang(r).ToString("0.000");
     label2.Text = CMeineFormeln.kugelVolumen(r).ToString("0.000");

     decimal b = Convert.ToDecimal(textBox2.Text);            // Brutto konvertieren
     label3.Text = CMeineFormeln.netto(b, 0.19).ToString("C");
}
```

BEISPIEL 3.26: Die statischen Methoden der Klasse *meineFormeln* werden in einer Eingabemaske aufgerufen.

Ergebnis



Bemerkungen

- Sie können mit *static* auch ein Feld deklarieren, das von allen Instanzen der Klasse gemeinsam genutzt werden kann und nicht für jedes Objekt extra zugewiesen werden muss. Zum Initialisieren kann ein so genannter *statischer Konstruktor* Verwendung finden (siehe 3.5.1).
- Eine Klasse mit ausschließlich statischen Mitgliedern kann mit dem Schlüsselwort *static* deklariert werden, siehe 3.7.4 oder Praxisbeispiel

  ▶ 3.9.2 Eine statische Klasse anwenden.

3.4 Ereignisse

Nachdem wir uns den Eigenschaften und Methoden von Objekten ausführlich gewidmet haben, wollen wir die Dritten im Bunde, die Ereignisse, nicht vergessen. Wie Sie bereits wissen, werden Ereignisse unter bestimmten Bedingungen vom Objekt ausgelöst und können dann in einer Ereignisbehandlungsroutine abgefangen und ausgewertet werden.

Allerdings bieten bei weitem nicht alle Klassen Ereignisse an, denn diese werden nur benötigt, wenn auf bestimmte Änderungen eines Objekts reagiert werden soll.

Nachdem wir mit dem Deklarieren von Eigenschaften und Methoden überhaupt keine Probleme hatten, hört aber bei Ereignissen der Spaß auf.

HINWEIS: Eine ausführliche Einführung in das Ereignismodell erhalten Sie im Kapitel 27 (Microsoft Event Pattern).

Im Folgenden werden deshalb nur die wichtigsten Grundlagen der Ereignismodellierung erläutert.

3.4.1 Ereignis hinzufügen

Um einer Klasse ein Ereignis hinzuzufügen, sind drei Schritte erforderlich:

1. Die Deklaration des Ereignistyps (*delegate*-Schlüsselwort)
2. Die Instanziierung des Ereignisses (*event*-Schlüsselwort)
3. Das Auslösen des Ereignisses (innerhalb einer Methode oder Eigenschaft)

Um einer heillosen Verwirrung vorzubeugen, machen wir es diesmal umgekehrt und beginnen gleich mit einem Beispiel, ehe wir später die Syntax und weitere Einzelheiten erklären.

BEISPIEL 3.27: Ereignis-Delegate hinzufügen

C#

In unserer *CKunde*-Klasse wird ein Ereignis-Delegate mit dem Namen *GuthabenLeer* deklariert, davon wird ein Ereignis mit dem Namen *guthabenLeer1* instanziiert. Dieses Ereignis "feuert" innerhalb der Methode *addGuthaben* genau dann, wenn das Guthaben den Wert von 10 € unterschreitet.

```
public class CKunde
{
```

Um den Code für den Benutzer der Klasse etwas zu vereinfachen, werden den privaten Feldern Standardwerte zugewiesen[1]:

```
    private bool _stammkunde = true;        //      initialisiertes Feld
    private decimal _guthaben = 100M;       //      dto.
```

1.Schritt: den Ereignistyp definieren:

```
    public delegate void GuthabenLeer(object sender, string e);
```

2. Schritt: eine Ereignisinstanz *guthabenLeer1* deklarieren:

```
    public event GuthabenLeer guthabenLeer1;
```

Die Methode, in welcher das Ereignis ausgelöst wird:

```
    public void addGuthaben(decimal betrag)
    {
        if (stammkunde) _guthaben += betrag;
```

3. Schritt: Ereignis auslösen:

```
        if (_guthaben <= 10)
        {
```

[1] Später werden wir diese Aufgabe dem Konstruktor übertragen.

BEISPIEL 3.27: Ereignis-Delegate hinzufügen

```
            // das Ereignis feuert nur, wenn ...
            if (guthabenLeer1 != null)  // ... mindestens ein Eventhandler angemeldet ist
            {
                string msg = "Das Guthaben beträgt nur noch " +
                             _guthaben.ToString("C") + "!";
                guthabenLeer1(this, msg);
            }
        }
    }
    // ... weitere Implementierungen
}
```

Nun kommen wir zu den sicherlich dringend notwendigen Erklärungen:

Ereignis deklarieren

Wie bereits kurz erwähnt, werden Ereignisse von so genannten Delegaten abgeleitet. Ein Delegate ist ein Ereignistyp, er sieht – bis auf das *delegate*-Schlüsselwort – wie eine Methode aus und verhält sich auch ähnlich.

HINWEIS: Delegaten ermöglichen es, ein Framework von Rückruf- und Benachrichtigungsmethoden für miteinander kooperierende Klassen zu implementieren.

SYNTAX: *Modifizierer* **delegate** *Datentyp delegateName* (*Datentyp Parameter*);

Falls der Delegate keinen Rückgabewert liefert, wird dieser – wie bei einer Methode – als *void* angegeben.

BEISPIEL 3.28: Die Deklaration des Delegaten aus dem Vorgängerbeispiel

C#
```
public delegate void GuthabenLeer(object sender, string e);
```

Ereignis instanziieren

Nachdem Sie mittels Delegaten den Ereignistyp deklariert haben, steht dem Erzeugen einer Delegatinstanz, also eines spezifischen Ereignisses, nichts mehr im Wege. Sie verwenden dazu das *event*-Schlüsselwort.

SYNTAX: *Modifizierer* **event** *delegateName ereignisName*;

Ähnlich wie bei Objekten (diese sind bekanntlich Instanzen einer Klasse) handelt es sich bei einem Ereignis um eine Instanz des Delegaten. Da der Aufbau bereits feststeht, genügen die Angabe des Namens der Ereignisdeklaration (*delegateName*) und der spezielle Name des Ereignisses (*ereignisName*).

BEISPIEL 3.29: Von im Vorgängerbeispiel deklarierten Delegaten wird ein Ereignis mit dem Namen *guthabenLeer1* instanziiert.

C#
```
public event GuthabenLeer guthabenLeer1;
```

Es ist durchaus möglich und üblich, auch mehrere Ereignisse vom gleichen Delegaten abzuleiten.

Ereignis auslösen

Ein Ereignis wird immer innerhalb der Klasse ausgelöst, in der es deklariert und erzeugt wurde. Das kann an verschiedenen Stellen innerhalb von Methoden oder Eigenschaften geschehen.

Das Auslösen erfolgt wie ein normaler Methodenaufruf:

SYNTAX: *ereignisName*(*Parameter*);

Die Signatur der Parameter (Reihenfolge und Datentyp) muss der im entsprechenden Delegate festgelegten Parameterliste entsprechen.

HINWEIS: Ereignisse sind Verweistypen und können demzufolge auch auf *null* abgefragt werden.

BEISPIEL 3.30: Ereignis auslösen

C#

Das im Vorgängerbeispiel deklarierte Ereignis wird innerhalb der Methode *addGuthaben* ausgelöst[1]. Vor dem Aufruf wird getestet, ob zumindest ein Eventhandler für dieses Ereignis angemeldet ist (was genau unter "Anmelden" zu verstehen ist, erfahren Sie im nächsten Abschnitt).

```
public void addGuthaben(decimal betrag)
{
    if (stammkunde) _guthaben += betrag;
    if (_guthaben <= 10)
    {
        if (guthabenLeer1 != null)    // mindestens ein Eventhandler angemeldet?
        {
            string msg = "Das Guthaben beträgt nur noch " +
                         _guthaben.ToString("C") + "!";
            guthabenLeer1(this, msg);
         }
    }
}
```

[1] Im Programmiererjargon sagt man auch "Das Ereignis feuert"!

3.4.2 Ereignis verwenden

In der Klasse, in welcher wir mit dem Ereignis arbeiten wollen, sind – zusätzlich zur Erzeugung der Objektvariablen – zwei Schritte durchzuführen:

1. Ereignisbehandlung (Eventhandler) schreiben
2. Eventhandler anmelden

Lassen Sie uns auch hier mit einem Beispiel beginnen.

BEISPIEL 3.31: Ereignisse verwenden

C#

Wir nutzen die im Vorgängerabschnitt definierte Klasse *CKunde*, welche von uns gerade mit dem Ereignis *guthabenLeer1* nachgerüstet wurde.

Auf Klassenebene referenzieren wir zunächst die übliche Objektvariable:

```
private CKunde kunde1;            // Objekt referenzieren
```

1. Schritt: Wir schreiben nun eine Ereignisbehandlung (Eventhandler) für das Ereignis:

```
private void guthabenKontrolle(object o, string s)
{
    CKunde k = (CKunde)o;
    label2.Text = k.nachName + ": " + s;       // Ausgabe einer Warnung
}
```

2. Schritt: Um das Ereignis mit dem Eventhandler zu verbinden, ist eine Anmeldung erforderlich, die wir in den Konstruktorcode von *Form1* einfügen können:

```
public Form1()
{
       InitializeComponent();
       kunde1 = new CKunde();      // Objekt instanziieren

       // Eventhandler anmelden:
       kunde1.guthabenLeer1 += new CKunde.GuthabenLeer(guthabenKontrolle);
}
```

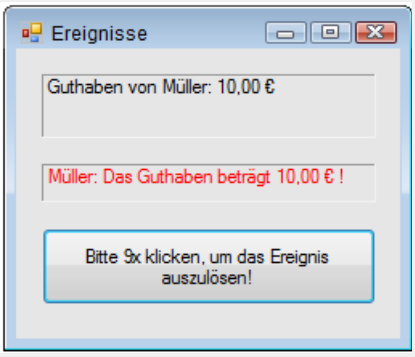
Das Ereignis ist jetzt eingebunden, und einem Funktionstest steht nichts mehr im Weg. Dazu rufen wir wiederholt die Methode *addGuthaben* auf, die das Guthaben jedes Mal um 10 € verringert:

```
private void button1_Click(object sender, EventArgs e)
{
    label1.Text = kunde1.adresse();
    kunde1.addGuthaben(-10M);         // Guthaben verringern
    label2.Text = "Guthaben ist " + kunde1.guthaben.ToString("C");
}
```

BEISPIEL 3.31: Ereignisse verwenden

Ergebnis

Nachdem Sie den *Button* neunmal geklickt haben, wird das Ereignis ausgelöst:



Den kompletten Quellcode entnehmen Sie bitte den Buchbeispielen.

Nun zu den Details.

Ereignisbehandlung schreiben

Die Frage "Was soll passieren, wenn das Ereignis ausgelöst wurde?" wird in einer Ereignisbehandlungsmethode (Eventhandler) beantwortet.

SYNTAX: *Modifizierer Datentyp methodenName (Datentyp Parameter)*

Den Namen der Methode können Sie frei wählen. Den konkreten Namen des Ereignisses finden Sie hier nicht, d.h., eine Ereignisbehandlung lässt sich auch von mehreren Ereignissen gemeinsam verwenden. Lediglich die Methodensignatur (*Datentyp Parameter*) muss der des Delegates entsprechen, nach dessen Muster das Ereignis erzeugt wurde.

BEISPIEL 3.32: Ereignisbehandlung schreiben

C#

Beim Auftreten des Ereignisses wird nicht nur der Parameter *s* angezeigt, in der Titelleiste erscheint zusätzlich noch der Name des Kunden, den wir durch explizite Typkonvertierung aus dem ebenfalls übergebenen *object*-Parameter "herausziehen".

```
private void guthabenKontrolle(object o, string s)
{
    CKunde k = (CKunde) o;                  // Typecasting
    label2.Text = k.nachName + ": " + s;    // Warnung ausgeben
}
```

Ereignisbehandlung anmelden

Um dem Compiler mitzuteilen, welcher Eventhandler bei Auftreten des Ereignisses denn nun aufzurufen ist, müssen Sie die gewünschte Ereignisbehandlung beim Objekt (der Klasseninstanz) anmelden.

SYNTAX: *Objekt.ereignisName* += new *delegateName*(*eventHandlerName*);

BEISPIEL 3.33: Dem Objekt *kunde1* wird mitgeteilt, dass bei Auftreten des Ereignisses *guthabenLeer1* der Eventhandler *guthabenKontrolle* aufzurufen ist

C#
```
kunde1.guthabenLeer1 += new CKunde.GuthabenLeer(guthabenKontrolle);
```

Wichtig ist dabei die Verwendung des Operators += (siehe Sprachkapitel, Abschnitt 2.4), denn pro Ereignis sind durchaus mehrere Eventhandler möglich. In diesem Fall erfolgt deren Abarbeitung in der Reihenfolge der Anmeldung.

BEISPIEL 3.34: Zum Ereignis *guthabenLeer1* wird ein zweiter Eventhandler hinzugefügt

C# Beim Eintreten des Ereignisses erscheint zunächst das vom ersten Eventhandler produzierte Meldungsfenster (siehe oben) und anschließend der Name des Kunden in einem Label.

```
// zweiten Eventhandler implementieren
private void kundenAnschrift(object o, string s)
{
    CKunde k = (CKunde) o;
    label3.Text = k.name;
}
 ....
// zweiten Eventhandler hinzufügen:
kunde1.guthabenLeer1 += new CKunde.GuthabenLeer(kundenAnschrift);
```

Falls ein Eventhandler nicht mehr benötigt wird, sollten Sie ihn wieder abmelden.

BEISPIEL 3.35: Abmelden eines Eventhandlers *kundenAnschrift*

C#
```
kunde1.guthabenLeer1 -= new CKunde.GuthabenLeer(kundenAnschrift);
```

Bemerkungen

Wenn Sie im Eigenschaften-Fenster der Visual Studio-Entwicklungsumgebung auf bekannte Weise Eventhandler für die Objekte der Bedienoberfläche erzeugen, so hat die IDE nicht nur den Rahmencode des Eventhandlers für Sie generiert, sondern – quasi im Verborgenen – auch die benutzten Ereignisse angemeldet. Üblicherweise übergeben diese Ereignisse zwei Parameter an die aufrufende Instanz: eine Referenz auf das Objekt, welches das Ereignis ausgelöst hat, und ein Objekt der *EvenArgs*- oder einer davon abgeleiteten Klasse.

BEISPIEL 3.36: Rahmencode des automatisch generierten Eventhandlers für das *Click*-Ereignis eines *Button*

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

}
```

Klappen Sie die Region *Vom Windows Form-Designer generierter Code* auf, so finden Sie die entsprechende Befehlszeile für die Anmeldung des Eventhandlers:

```
this.button1.Click += new EventHandler(this.button1_Click);
```

Die folgende Abbildung veranschaulicht nochmals die Syntax dieser Zeile.

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

Objekt Ereignis Delegat Event-Handler

Und zum Schluss noch ein Hinweis auf einen ziemlich häufigen Unterlassungsfehler:

HINWEIS: Wenn Sie die Codezeilen eines Eventhandlers komplett per Hand löschen (man sollte das eigentlich nicht tun), so müssen Sie auch die entsprechende Anmeldungszeile (siehe oben) löschen, ansonsten gibt es einen Compilerfehler!

3.5 Arbeiten mit Konstruktor und Destruktor

Eine "richtige" objektorientierte Sprache wie C# realisiert das Erzeugen und Entfernen von Objekten mit Hilfe von Konstruktoren und Destruktoren.

"Bis jetzt sind wir doch glänzend ohne Konstruktor ausgekommen!", werden Sie jetzt vielleicht einwenden. Ganz stimmt das nicht, denn wenn Sie sich um keinen eigenen Konstruktor kümmern, wird der von *System.Object* geerbte parameterlose *new*-Standardkonstruktor verwendet.

3.5.1 Konstruktor und Objektinitialisierer

Der Konstruktor ist gewissermaßen die Standardmethode der Klasse und kann in mehreren Überladungen vorhanden sein.

HINWEIS: Der Name des Konstruktors ist immer identisch mit dem Namen der Klasse.

Der Konstruktor wird automatisch bei der Instanziierung eines Objekts (*new*) aufgerufen und dient vor allem dazu, den Feldern des neu erzeugten Objekts Anfangswerte zuzuweisen.

Deklaration

Einen Konstruktor fügen Sie dem Klassenkörper ähnlich wie eine *public void*-Methode hinzu, nur dass Sie der Methode den Namen der Klasse geben und auf das *void*-Schlüsselwort verzichten. Als Parameter übergeben Sie die Werte für die Felder, die initialisiert werden sollen.

SYNTAX:
```
public KlassenName(Datentyp Parameter)
{
   // Initialisierung der Klasse
}
```

Wie bei jeder anderen Methode können Sie auch hier mehrere überladene Konstruktoren implementieren.

BEISPIEL 3.37: Unserer Klasse *CKunde* werden zwei überladene Konstruktoren hinzugefügt

C#
```
public class CKunde
{
```

Die Felder:

```
    private string _anrede;
    private string _name;
    private int _plz;
    private string _ort;
    private bool _stammKunde;
    private decimal _guthaben;
```

Der erste Konstruktor initialisiert nur zwei Felder:

```
   public CKunde(string anr, string nam)
   {
       _anrede = anr; _name = nam;
   }
```

Der zweite Konstruktor initialisiert **alle** Felder der Klasse:

```
   public CKunde(string a, string n, int p, string o, bool s, decimal g)
   {
       _anrede = a;   _name = n;     _plz = p;
       _ort = o;  _stammKunde = s;   _guthaben = g;
   }
   ...
}
```

Aufruf

Nachdem Sie einer Klasse einen oder mehrere Konstruktoren hinzugefügt haben, sind Sie auch zur Verwendung von mindestens einem davon verpflichtet. Die bisher gewohnte einfache Instanziierung von Objekten ist nicht mehr möglich!

BEISPIEL 3.38: Zwei Objekte der Klasse *CKunde* werden erzeugt und mit Anfangswerten initialisiert.

C#

Objekte referenzieren:

```
CKunde kunde1, kunde2, kunde3;
```

Sicherheitshalber bauen wir das Erzeugen der Objekte in einen Exception-Handler ein:

```
try
{
```

Für jedes Objekt wird ein anderer überladener Konstruktor verwendet:

```
    kunde1 = new CKunde("Herr", "Müller");
    kunde2 = new CKunde("Frau", "Hummel", 12345, "Berlin", true, 100);
    //  kunde3 = new CKunde();                        // erzeugt Compilerfehler!!!
    MessageBox.Show("Objekte erfolgreich erzeugt!");
}
catch(Exception ex)
{
    MessageBox.Show(ex.Message + " Sch... Konstruktor!");
}
```

Wenn Sie den Code mit dem Anfangsbeispiel im Abschnitt 3.1.6 vergleichen, so sehen Sie, dass das Initialisieren der Objekte viel übersichtlicher geworden ist. Statt umständlich eine Eigenschaft nach der anderen zuzuweisen, geht das jetzt in einer einzigen Befehlszeile.

Objektinitialisierer

Objektinitialisierer erlauben das Erzeugen und Initialisieren von Instanzen auf Basis von (öffentlichen) Objekteigenschaften auch ohne das explizite Vorhandensein eines Konstruktors[1].

BEISPIEL 3.39: Erzeugen eines Objekts mittels Objektinitialisierer (das ist kein Konstruktoraufruf!)

C#

```
kunde1 = new CKunde {Anrede = "Herr", Name = "Müller};
```

Statischer Konstruktor

Neben einem oder mehreren "normalen" Konstruktoren kann eine Klasse auch über einen *statischen* Konstruktor verfügen. Ein solcher Konstruktor wird verwendet, um *static*-Felder (siehe 3.2.5) zu initialisieren oder um einmaligen Initialisierungscode auszuführen. Der Aufruf erfolgt automatisch, bevor die erste Instanz erstellt oder auf statische Klassenmitglieder verwiesen wird.

[1] Notwendig wurde diese Spracherweiterung vor allem wegen der neu eingeführten LINQ-Technologie, deren anonyme Typen beispielsweise nach einem solchen Feature verlangen (siehe Kapitel 6).

BEISPIEL 3.40: Eine Klasse besitzt die statische Eigenschaft *MWSt*, welche zu Beginn mit dem Wert 0,19 initialisiert werden soll.

C#
```
public class CAllerlei
{
    private static double _mwst;

    public static double MWSt
    {
        get { return (_mwst); }
        set { _mwst = value; }
    }

    static CAllerlei()                              // statischer Konstruktor
    {
        _mwst = 0.19;
    }
    ...
}
```

Der Zugriff von einem Formular *Form1* aus:

```
public partial class Form1 : Form
{
    ...
    textBox1.Text = CAllerlei.MWSt.ToString();       // zeigt 0,19
    ...
}
```

HINWEIS: Ein statischer Konstruktor akzeptiert weder Zugriffsmodifizierer, noch besitzt er Parameter, er kann auch nicht direkt aufgerufen werden.

3.5.2 Destruktor und Garbage Collector

Das Pendant zum Konstruktor ist aus objektorientierter Sicht der Destruktor. Da der Lebenszyklus eines Objektes bekanntlich mit dessen Zerstörung und der Freigabe der belegten Speicherplatzressourcen endet, ist der Destruktor für das Erledigen von "Aufräumarbeiten" zuständig, kurz bevor das Objekt sein Leben aushaucht.

In .NET haben wir allerdings keine echten Destruktoren, da hier die endgültige Zerstörung eines Objekts nicht per Code, sondern automatisch vom Garbage Collector vorgenommen wird. Dieser durchstöbert willkürlich und in unregelmäßigen Zeitabständen den Heap nach Objekten, um diejenigen zu suchen, die nicht mehr referenziert werden.

An die Stelle eines echten Destruktors tritt ein Quasi-Destruktor. Das ist eine Finalisierungsmethode, die zu einem unbestimmbaren Zeitpunkt vom Garbage Collector aufgerufen wird, kurz bevor dieser das Objekt vernichtet.

Ähnlich wie beim Konstruktor wird auch hier der Name der Klasse als Methodenbezeichner verwendet, allerdings mit einer Tilde (~) als Präfix. Der *public*-Zugriffsmodifizierer entfällt, da Sie selbst den Destruktor nicht aufrufen dürfen, auch Parameter dürfen nicht übergeben werden.

SYNTAX:
```
~KlassenName()
{
    // hier Code für Aufräumarbeiten implementieren
}
```

BEISPIEL 3.41: Destruktor und Garbage Collector

C#

Unsere Klasse *CKunde* erhält ein öffentliches statisches Feld, welches durch den Konstruktor inkrementiert und durch den Quasi-Destruktor dekrementiert werden soll. Wir beabsichtigen damit, die Anzahl der momentan instanziierten Klassen (sprich Anzahl der Kunden) abzufragen.

Der auf das Wesentliche reduzierte Code von *CKunde*:

```
public class CKunde
{
     public static int anzahl = 0;

     // Konstruktor:
     public CKunde()
     {
         anzahl++;
     }

     // Destruktor:
      ~CKunde()
     {
          anzahl--;
     }
}
```

Wir verwenden zum Testen der Klasse ein Windows-Formular mit zwei *Button*s, einer *Timer*-Komponente (*Interval* = 1000, *Enabled* = *True*) und einem *Label*.

Zum Code der Klasse *Form1* fügen Sie hinzu:

```
CKunde kunde1;  // Objekt referenzieren

// Objekt hinzufügen:
private void button1_Click(object sender, EventArgs e)
{
    kunde1 = new CKunde();
}

// Objekt entfernen:
private void button2_Click(object sender, EventArgs e)
{
```

BEISPIEL 3.41: Destruktor und Garbage Collector

```
        kunde1 = null;     // dereferenzieren
    }


    // Anzeige der im Speicher befindlichen Instanzen im Sekundentakt:
    private void timer1_Tick(object sender, EventArgs e)
    {
        label1.Text = CKunde.anzahl.ToString();
    }
```

Ergebnis

Beim Programmtest müssen Sie etwas Geduld aufbringen.

Nach dem Programmstart fügen Sie durch Klicken auf den linken Button ein Objekt *kunde1* hinzu, wonach sich die Anzeige von 0 auf 1 ändert. Anschließend klicken Sie auf den rechten Button, um das Objekt wieder zu entfernen.

Es kann einige Zeit dauern, bis die Anzeige wieder auf 0 zurück geht, nämlich dann, wenn dem Garbage Collector gerade einmal wieder die Lust zum Aufräumen überkommt und er den Quasi-Destruktor aufruft[1].

Übrigens können Sie auch den linken Button mehrmals hintereinander klicken. Die Anzeige zählt zwar hoch, das aber täuscht, denn es bleibt bei nur einer Objektvariablen (*kunde1*). Allerdings wird Ressourcenverschwendung betrieben, denn dem Objekt wird immer wieder ein neuer Speicherbereich zugewiesen. Der vorher belegte Speicher liegt brach und wartet auf die Freigabe durch den Garbage Collector.

HINWEIS: Obiges Beispiel sollten Sie aufgrund seiner Unberechenbarkeit keinesfalls als Vorbild für ähnliche Zählaufgaben verwenden!

[1] Der Garbage Collector läuft in einem eigenen Thread, er wird nur dann aufgerufen, wenn sich die anderen Threads in einem sicheren Zustand befinden.

Da wegen der Unberechenbarkeit der Objektvernichtung der Umgang mit dem Quasi-Destruktor ziemlich problematisch ist, sollten Sie für das definierte Freigeben von Objekten besser eine separate Methode (*Close*- bzw. *Dispose*) oder *using* verwenden (siehe folgender Abschnitt).

3.5.3 Mit using den Lebenszyklus des Objekts kapseln

Mit dem Schlüsselwort *using* kann man nicht nur Namespaces einbinden (siehe Kapitel 5), sondern in völlig anderer Bedeutung auch für das sichere Erzeugen und Vernichten von Objekten sorgen. Hinter den Kulissen wird ein *try-finally*-Block um das entsprechende Objekt generiert und beim Beendigen für das Objekt *Dispose()* aufgerufen. Das folgende (eigentlich lächerliche) Beispiel soll lediglich das Prinzip verdeutlichen.

BEISPIEL 3.42: Zum Prinzip von *using*

C#
```
using (CKunde kunde1 = new CKunde("Herr", "Müller"))      // Erzeugen des Objekts
{
    kunde1.Wohnort = "Berlin";                             // Arbeiten mit dem Objekt
    ...
}                                                          // Freigabe des Objekts
```

Weitaus sinnvollere Beispiele für *using* finden Sie im Praxisbeispiel

► 8.8.3 Ein Memory Mapped File (MMF) verwenden

und im Abschnitt 23.3.5 des ADO.NET-Kapitels (Datenbankzugriff).

3.5.4 Verzögerte Initialisierung

Erzeugen Sie wie bisher ein Objekt mit *new*, so wird der Speicher gleich bei der Initialisierung belegt. Die verzögerte Initialisierung (*Lazy Initialization*) von Objekten hat hingegen den Vorteil, dass die Objekte erst dann Speicherplatz belegen, wenn sie tatsächlich verwendet werden. Das lohnt sich besonders für Anwendungen mit sehr vielen oder sehr umfangreichen Klasseninstanzen.

Realisiert wird die verzögerte Initialisierung mit der ab .NET 4.0 eingeführten generischen Klasse *System.Lazy<>*, welche die tatsächliche Klasse kapselt.

Um das Prinzip zu verdeutlichen, gehen wir von einer allgemeinen Klasse aus:

```
public class Klasse1
{
    public Klasse1()
    {
        // Konstruktor
    }

    public string Eigenschaft1
    { get; set; }
```

```
    public void Methode1()
    {
        // ...
    }
}
```

Die verzögerte Initialisierung wird vorbereitet:

```
Lazy<Klasse1> objInit;
objInit = new Lazy<Klasse1>();
```

Bis jetzt wurde das Objekt noch nicht erstellt (der Wert der *objInit.IsValueCreated*-Eigenschaft ist *false*).

Erst beim Zugriff auf eines seiner Mitglieder wird das Objekt erzeugt:

```
objInit.Value.Eigenschaft1 = "Das ist der erste Wert!";
```

Damit ist das Objekt initialisiert (die *objInit.IsValueCreated*-Eigenschaft ist *true*).

HINWEIS: Standardwerte von Feldern werden erst bei der erstmaligen Verwendung zugewiesen, dasselbe gilt für die Ausführung eines evtl. vorhandenen Konstruktors.

3.6 Vererbung und Polymorphie

Ein zentrales OOP-Thema ist die *Vererbung*, die es ermöglicht, Klassen zu definieren, die von anderen Klassen abhängen. Eng mit der Vererbung verknüpft ist die *Polymorphie* (Vielgestaltigkeit). Man versteht darunter die Fähigkeit von Subklassen, die Methoden der Basisklasse mit unterschiedlichen Implementierungen zu verwenden. C# unterstützt sowohl Vererbung als auch polymorphes Verhalten, da das Überschreiben (*Overriding*) der Basisklassenmethoden mit alternativen Implementierungen erlaubt ist.

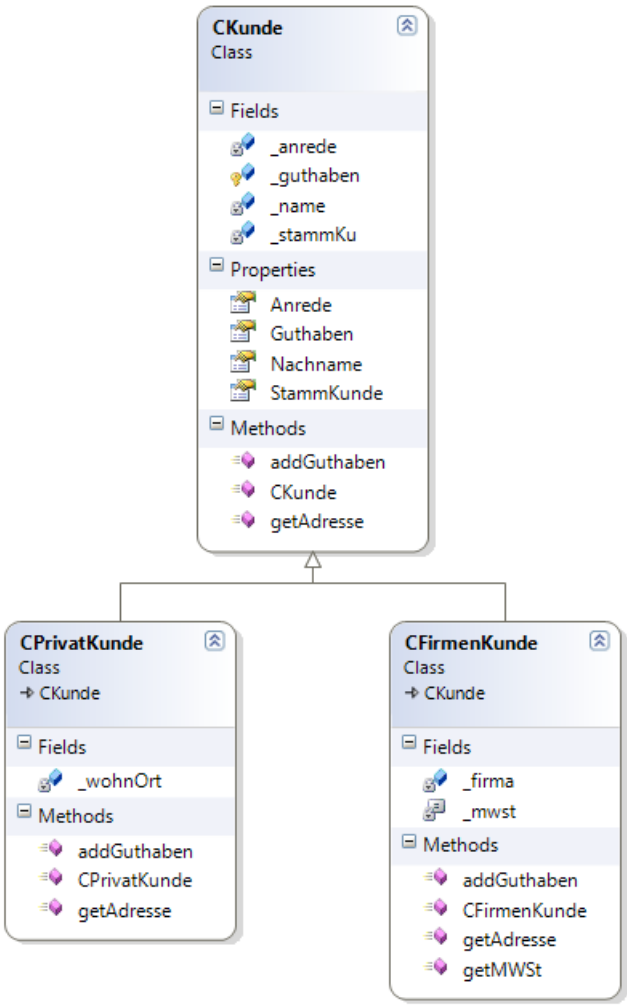
Durch Vererbung können Sie sich die Programmierarbeit wesentlich erleichtern, indem Sie spezialisierte Subklassen verwenden, die den Code zum großen Teil von einer allgemeinen Basisklasse erben. Die Subklassen heißen auch *abgeleitete Klassen, Kind-* oder *Unterklassen*, die Basisklasse wird auch als *Super-* oder *Elternklasse* bezeichnet. In den Subklassen können Sie bestimmte Funktionalitäten überschreiben, um spezielle Prozesse auszuführen.

Lassen Sie uns anhand eines kurzen und dennoch ausführlichen Beispiels die wichtigsten Vererbungstechniken demonstrieren! Wir beginnen mit dem Klassendiagramm.

3.6.1 Klassendiagramm

Mittels *Unified Modeling Language* (UML) lassen sich Vererbungsbeziehungen zwischen verschiedenen Klassen grafisch darstellen.

Das folgende, mit Visual Studio erzeugte, Klassendiagramm[1] zeigt eine Basisklasse *CKunde*, von der die Klassen *CPrivatKunde* und *CFirmenKunde* "erben". Die Basisklasse hat die Eigenschaften *Anrede*, *Nachname*, *StammKunde* (ja/nein) und *Guthaben* und die Methoden *getAdresse()* und *addGuthaben()* (das Guthaben ist hier als Bonus zu verstehen, der den Kunden in prozentualer Abhängigkeit von den getätigten Einkäufen gewährt wird). Die Methode *CKunde* ist nichts weiter als der Konstruktor.



[1] Der Umgang mit dem Klassendesigner wird ausführlich im Kapitel 26 beschrieben.

3.6.2 Method-Overriding

Die Subklassen *CPrivatKunde* und *CFirmenKunde* können auf sämtliche Eigenschaften und Methoden der Basisklasse zugreifen und fügen selbst eigene Methoden (auch Eigenschaften wären natürlich möglich) hinzu.

Die "geerbten" Methoden *getAdresse* und *addGuthaben* sind in unserem Beispiel allerdings so genannte *überschriebene Methoden* (*Method-Overriding*), d.h., Adresse und Guthaben sollen für Privatkunden auf andere Weise als für Firmenkunden ermittelt werden. Genaueres dazu erfahren Sie im nächsten Abschnitt.

3.6.3 Klassen implementieren

Vorbild für die drei zu implementierenden Klassen ist obiges Klassendiagramm.

Basisklasse CKunde

Die Deklaration entspricht (fast) der einer normalen Klasse. Dass es sich um eine Basisklasse handelt, erkennt man in unserem konkreten Fall eigentlich nur an dem *protected*-Feld und an den *virtual*-Methodendeklarationen[1].

```
public class CKunde                // Basisklasse
{
```

Die privaten Felder:

```
    private string _anrede;
    private string _name;
    private bool _stammKu;
```

Durch den *protected*-Modifizierer für das *guthaben*-Feld wird es möglich, dass auch die beiden Subklassen auf dieses Feld zugreifen können:

```
    protected decimal _guthaben = 0;      // Feld ist in Subklassen sichtbar!
```

Ein eigener Konstruktor ersetzt den Standardkonstruktor:

```
    public CKunde(string anr, string nName)     // Konstruktor
    {
        _anrede = anrede; _name = nName;
    }
```

Die Eigenschaften:

```
    public string Anrede
    {
         get { return (_anrede); }
         set { _anrede = value; }
    }
```

[1] Eigentlich hätten wir unsere Klasse auch noch als *abstract* deklarieren müssen (siehe dazu Abschnitt 3.7.1).

```
    public string Nachname
    {
         get { return (_name); }
         set { _name = value; }
    }
    public bool StammKunde
    {
        get {return(_stammKu); }
        set {_stammKu = value; }
    }

    public decimal Guthaben        // ReadOnly
    {
        get {return(_guthaben); }
    }
```

Nun zu den beiden Methoden, die durch die Subklassen überschrieben werden können.

Der Rückgabewert der virtuellen Methode *getAdresse()* setzt sich aus Anrede und Namen des Kunden zusammen:

```
    public virtual string getAdresse()                    // virtuelle Methode
    {
        string s = _anrede + " " + _name;
        return(s);
    }
```

Die virtuelle Methode *addGuthaben()* erhöht das Guthaben des Kunden um den im Argument übergebenen Bonusbetrag. Allerdings kommen nur Stammkunden in diesen Genuss:

```
    public virtual void addGuthaben(decimal betrag)      // virtuelle Methode
    {
        if (_stammKu) _guthaben += betrag;
    }
}
```

Subklasse CPrivatKunde

Diese Klasse erbt alle Eigenschaften und Methoden der Basisklasse, wird also sozusagen um deren Code "erweitert". Das *override*-Schlüsselwort der beiden Methoden bedeutet, dass hier die in der Basisklasse als *virtual* definierten Funktionen überschrieben werden. Das erlaubt der Subklasse, eine eigene Implementierung der Funktionen zu realisieren.

```
public class CPrivatKunde : CKunde             // erbt von der Basisklasse CKunde!
{
   private string _wohnOrt;
```

Der Konstruktor ist unbedingt notwendig, weil auch die Basisklasse einen eigenen Konstruktor verwendet. Es wird das *base*-Schlüsselwort benutzt, um den Konstruktor der Basisklasse aufzurufen.

```
    public CPrivatKunde(string anrede, string name, string ort): base(anrede, name)
    {
        _wohnOrt = ort;              // klassenspezifische Ergänzung
    }
```

Die Methode *getAdresse()* wird so überschrieben, dass zusätzlich zu Anrede und Name (von der Basisklasse geerbt) noch der Wohnort des Privatkunden angezeigt wird.

```
    public override string getAdresse()
    {
        const char LF = (char) 10;       // Zeilenvorschub
        return(base.adresse() + LF + _wohnOrt);
    }
```

Die Methode *addGuthaben()* wird komplett neu überschrieben. Ohne Rücksicht auf die Zugehörigkeit zur Stammkundschaft werden jedem Privatkunden 5% vom Rechnungsbetrag als Bonusguthaben angerechnet:

```
    public override void addGuthaben(decimal geld)
    {
        // Zugriff auf protected-Variable in Basisklasse:
        _guthaben += 0.05M * geld;
    }
}
```

Subklasse CFirmenKunde

Der Code für die Subklasse *CFirmenKunde* unterscheidet sich in folgenden Details von der Klasse *CPrivatKunde*:

- Die Methode *getAdresse()* liefert statt des Wohnorts den Namen der Firma des Kunden.
- Die *addGuthaben()*-Methode berechnet zunächst den Nettobetrag und addiert davon 1% zum Bonusguthaben. Damit nur Stammkunden in den Genuss dieser Vergünstigung kommen, wird dazu die gleichnamige Methode der Basisklasse aufgerufen.
- Die neu hinzugekommene "stinknormale" Methode *getMWSt()* erlaubt einen Lesezugriff auf die Mehrwertsteuer-Konstante.

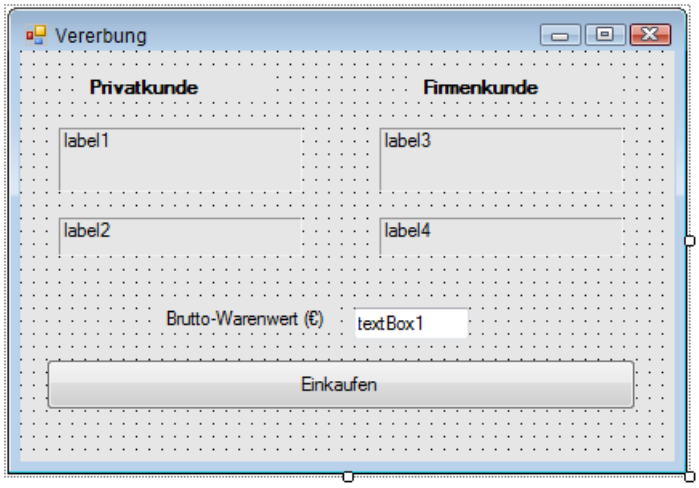
```
public class CFirmenKunde : CKunde
{
    private string _firma;
    private  const float _mwst = 0.19F;     // Mehrwertsteuer

    public CFirmenKunde(string anrede, string name, string frm): base(anrede, name)
    {
        _firma = frm;
    }

    public override string adresse()
    {
        const char LF = (char) 10;      // Zeilenvorschub
```

```
        return(base.adresse() + LF + _firma);
    }

    public override void addGuthaben(decimal brutto)
    {
        decimal netto = brutto / Convert.ToDecimal(1 + _mwst);
        base.addGuthaben(netto * 0.01M);   // Aufruf der Methode der Basisklasse
    }

    public double getMWSt()    // eine ganz normale Methode
    {
        return(_mwst);
    }
}
```

Die Implementierung unserer drei Klassen ist geschafft!

Testoberfläche

Um die Funktionsfähigkeit der drei Klassen zu testen, gestalten Sie die folgende Benutzerschnittstelle:



3.6.4 Implementieren der Objekte

Für einen kleinen Test genügt es, wenn wir mit nur zwei Objekten (ein Privat- und ein Firmenkunde) arbeiten.

```
CPrivatKunde kunde1;
CFirmenKunde kunde2;
```

Im Konstruktor des Formulars werden die beiden Objekte erzeugt. Die Ja-/Nein-Eigenschaft *StammKunde* muss allerdings extra zugewiesen werden, da es dazu keinen passenden Konstruktor gibt.

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

    public Form1()
    {
        InitializeComponent();

        kunde1 = new CPrivatKunde("Herr", "Krause", "Leipzig");
        kunde1.StammKunde = false;

        kunde2 = new CFirmenKunde("Frau", "Müller", "Master Soft GmbH");
        kunde2.StammKunde = true;
        textBox1.Text = "100";
    }
```

Bei Klick auf den "Einkaufen"-Button werden für jedes Objekt diverse Eigenschaften abgefragt und Methoden aufgerufen:

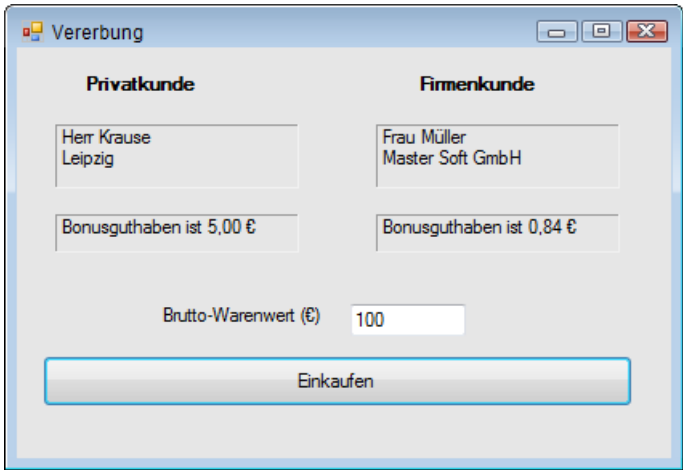
```
    private void button1_Click(object sender, EventArgs e)
    {
        decimal brutto = Convert.ToDecimal(textBox1.Text);
        label1.Text = kunde1.getAdresse();
        kunde1.addGuthaben(brutto);
        label2.Text = "Bonusguthaben ist " + kunde1.Guthaben.ToString("C");

        label3.Text = kunde2.getAdresse();
        kunde2.addGuthaben(brutto);
        label4.Text = "Bonusguthaben ist " + kunde2.guthaben.ToString("C");
    }
}
```

Praxistest

Überzeugen Sie sich nun davon, dass die drei Klassen wie gewünscht zusammenarbeiten und dass Vererbung tatsächlich funktioniert.

Die Werte in der folgenden Laufzeitabbildung sind wie folgt zu interpretieren:

- Dem Privatkunden Krause wurde ein Guthaben von 5 € (5% aus 100 €) zugebilligt (Stammkundschaft spielt bei Privatkunden keine Rolle, da die Methode *addGuthaben()* komplett überschrieben ist).
- Frau Müller ist eine Firmenkundin und erhält – nur weil sie Stammkundin ist – ein mickriges Guthaben von 0,84 € (1% auf den Nettowert).
- Durch wiederholtes Klicken auf "Einkaufen" kumulieren die Bonusguthaben.



3.6.5 Ausblenden von Mitgliedern durch Vererbung

Durch in eine abgeleitete Klasse oder Struktur eingeführte Mitglieder (Konstanten, Felder, Eigenschaften, Methoden, Ereignisse oder Typen) werden alle gleichnamigen Basisklassenelemente verdeckt bzw. ausgeblendet.

BEISPIEL 3.43: Zur Klasse *CKunde* fügen wir eine Methode *test* hinzu

C#

```
class CKunde                        // Basisklasse
{
   ...
   public void test()
   {
        MessageBox.Show("Hallo Kunde!");
   }
}
```

Eine Methode gleichen Namens fügen wir auch zur Klasse *CPrivatKunde* hinzu:

```
class CPrivatKunde : CKunde     // abgeleitete Klasse
{
   ...
   public void test()
   {
     MessageBox.Show("Hallo PrivatKunde!");
   }
}
```

Im Quellcode-Editor erscheint der Name *test()* grün unterschlängelt. Der entsprechende Warnhinweis lautet: *"WindowsFormsApplication1.CPrivatKunde.test()" blendet den versteckten Member "WindowsFormsApplication1.CKunde.test()" aus. Verwenden Sie das new-Schlüsselwort, wenn das Ausblenden vorgesehen war.*

BEISPIEL 3.43: Zur Klasse *CKunde* fügen wir eine Methode *test* hinzu

Der Test erfolgt in *Form1*:

```
public partial class Form1 : Form
{
    private CPrivatKunde kunde1 = new CPrivatKunde();
    ...
    private void button1_Click(object sender, EventArgs e)
    {
       ...
       kunde1.test();
    }
 }
```

Ergebnis

Wie Sie sehen, wurde die Methode *test()* der Klasse *CKunde* durch die Methode *test()* der Klasse *CPrivatKunde* ausgeblendet:

Um Missverständnissen vorzubeugen (und um obigen Warnhinweis im Quellcode zu vermeiden), sollte man die gleichnamige Methode in der abgeleiteten Klasse mit dem *new*-Schlüsselwort markieren.

BEISPIEL 3.44: Die folgende Änderung macht das Vorgängerbeispiel transparenter (Ergebnis bleibt dasselbe)

C#

```
...
class CPrivatKunde : CKunde     // abgeleitete Klasse
{
   ...
   new public void test()
   {
     MessageBox.Show("Hallo PrivatKunde!");
   }
}
```

HINWEIS: Die Deklaration eines neuen Mitglieds verdeckt ein geerbtes Mitglied nur innerhalb des Gültigkeitsbereichs des neuen Mitglieds!

3.6.6 Allgemeine Hinweise und Regeln zur Vererbung

Nachdem wir nun am praktischen Beispiel die Programmierung von Vererbungsbeziehungen kennengelernt haben, werden wir auch die folgenden Regeln und Hinweise verstehen:

- Alle öffentlichen Eigenschaften und Methoden der Basisklasse sind auch über die abgeleiteten Subklassen verfügbar.
- Methoden der Basisklasse, die von den abgeleiteten Subklassen überschrieben werden dürfen (so genannte *virtuelle Methoden*), müssen mit dem Schlüsselwort *virtual* deklariert werden.
- Fehlt das Schlüsselwort *virtual* bei der Methodendeklaration, so bedeutet das, dass dies die einzige Implementierung der Methode ist.
- Methoden der Subklassen, welche die gleichnamige Methode der Basisklasse überschreiben, müssen mit dem Schlüsselwort *override* deklariert werden.
- Wenn Sie das *override*-Schlüsselwort in der Subklasse vergessen, wird angenommen, dass es sich um eine "Schattenfunktion" der originalen Funktion handelt. Eine solche Funktion hat denselben Namen wie das Original, überschreibt dieses aber nicht.
- Private Felder der Basisklasse, auf die die Subklassen zugreifen dürfen, müssen mit *protected* deklariert werden.
- Die Basisklasse wird der Subklasse durch einen der Klassendeklaration nachgestellten Doppelpunkt bekannt gemacht:

  SYNTAX:
  ```
  class SubKlasse : BasisKlasse
  {
      // ... Implementierungscode
  }
  ```

- Eine Subklasse kann immer nur von einer einzigen Basisklasse abgeleitet werden (keine multiple Vererbung möglich).
- Mit dem *base*-Objekt kann von den Subklassen auf die Basisklasse zugegriffen werden, mit dem *this*-Objekt auf die eigene Klasse.
- Wenn die Basisklasse einen eigenen Konstruktor verwendet, so müssen in den Subklassen ebenfalls eigene Konstruktoren definiert werden (Konstruktoren können nicht vererbt werden!).
- Der Konstruktor einer Subklasse muss den Konstruktor seiner Basisklasse aufrufen (*base*-Schlüsselwort).
- Falls aber die Basisklasse über keinen eigenen Konstruktor verfügt, wird der Standardkonstruktor automatisch aufgerufen, wenn ein Objekt aus einer Subklasse erzeugt wird.

Wenn Sie mit Vererbung arbeiten, sollten Sie Folgendes beachten:

- Es gibt keinerlei Beschränkung bezüglich der Stufenanzahl der Vererbungshierarchie. Sie können die Hierarchie so tief wie nötig staffeln, die Eigenschaften/Methoden werden trotzdem durch alle Vererbungsstufen hindurchgereicht. Allgemein gilt, je weiter unten sich eine Klasse in der Hierarchie befindet, umso spezialisierter ist ihr Verhalten. Zum Beispiel eine *CHoch-*

schulKunden-Klasse, die von einer *CSchulKunden* erbt und diese wiederum von der *CKunden*-Klasse.

- Um die Komplexität zu minimieren und die Wartbarkeit des Codes zu vereinfachen, sollten Sie die Vererbungshierarchie nicht tiefer als ca. vier Stufen staffeln.
- Jede Subklasse kann nur von einer Basisklasse erben! So kann z.B. eine *CHochSchulKunden*-Klasse nicht sowohl von der *CKunden*-Klasse und einer *CSchulKunden*-Klasse erben. Das ist in Ordnung so, denn eine solche multiple Vererbung könnte sehr schnell zu einem komplexen, unübersichtlichen und nicht mehr beherrschbaren Ungetüm entarten.

Es gibt zwei primäre Anwendungsfälle für Vererbung in Anwendungen:

- Sie verwenden Objekte unterschiedlichen Typs mit ähnlicher Funktionalität. So erben z.B. *CSchulKunden*-Klasse und *CStaatsKunden*-Klasse von der *CKunden*-Klasse.
- Sie haben gleiche Prozesse mit einer Menge von Objekten auszuführen. So erbt z.B. jeder Typ eines Geschäftsobjekts von einer Business Object(BO)-Klasse.

Sie sollten in folgenden Fällen auf Vererbung verzichten:

- Sie brauchen nur eine einzige Funktion von der Basisklasse. In diesem Fall sollten Sie die Funktion in die eigene Klasse delegieren, statt von einer anderen zu erben.
- Sie möchten alle Funktionen überschreiben. In einem solchen Fall sollten Sie eine Schnittstelle (*Interface*, siehe Abschnitt) statt Vererbung verwenden.

3.6.7 Polymorphes Verhalten

Untrennbar mit der Vererbung verbunden ist die so genannte Polymorphie (Vielgestaltigkeit). Polymorphes Verhalten bedeutet, dass erst zur Laufzeit einer Anwendung entschieden wird, welche der möglichen Methodenimplementierungen aufgerufen wird, da dies zum Zeitpunkt des Compilierens noch unbekannt ist.

Im obigen Beispiel hatten wir von den Vorzügen der Polymorphie allerdings noch keinen Gebrauch gemacht, denn Privat- und Firmenkunde wurden in einzelnen Objektvariablen gespeichert und bereits per Programmcode fest mit ihren Methoden *getAdresse()* und *addGuthaben()* verbunden.

Um Polymorphie sichtbar zu machen, müssen wir das bei der Implementierung der Objekte zielgerichtet ausnutzen. Wie wir gleich sehen werden, treten die Vorzüge von Polymorphie besonders augenscheinlich zutage, wenn Objekte unterschiedlicher Klassenzugehörigkeit nacheinander in Arrays oder Auflistungen abgespeichert werden.

BEISPIEL 3.45: Polymorphes Verhalten

C#

Wir nehmen die drei Klassen des Vorgängerbeispiels (*CKunde*, *CPrivatKunde*, *CFirmenKunde*) als Grundlage. An deren Implementierungen brauchen wir keinerlei Veränderungen vorzunehmen, denn polymorphes Verhalten ergibt sich als logische Konsequenz aus der Vererbung von Klassen. Änderungen müssen wir lediglich beim Abspeichern der Objektvariablen vornehmen.

BEISPIEL 3.45: Polymorphes Verhalten

Aus den Subklassen *CPrivatKunde* und *CFirmenKunde* wollen wir insgesamt drei Objekte (*kunde1*, *kunde2*, *kunde3*) instanziieren (ein Privatkunde, zwei Firmenkunden).

Innerhalb des Klassencodes von *Form1* deklarieren Sie:

```
private CPrivatKunde kunde1;
private CFirmenKunde kunde2, kunde3;
private CKunde[] kunden = new CKunde[3];       // Array für 3 Objekte!

private const char LF = (char) 10;             // für Zeilenvorschub
```

Im Konstruktor von *Form1* werden die notwendigen Initialisierungen vorgenommen:

```
    public Form1()
    {
        kunde1 = new CPrivatKunde("Herr", "Krause", "Leipzig");
        kunde1.StammKunde = false;
        kunde2 = new CFirmenKunde("Frau", "Müller", "Master Soft GmbH");
        kunde2.StammKunde = true;
        kunde3 = new CFirmenKunde("Herr", "Maus", "Manfreds Internet AG");
        kunde3.StammKunde = false;
```

Da das Array vom Typ der Basisklasse ist, kann es auch Objekte der Subklassen (Privat- und Firmenkunden) in wahlloser Reihenfolge aufnehmen:

```
        kunden[0] = kunde1;
        kunden[1] = kunde2;
        kunden[2] = kunde3;
        textBox1.Text = "100";
}
```

Das Array wird in einer *for*-Schleife durchlaufen und ausgelesen. Dabei werden die polymorphen Methoden (das sind die mit *virtual* bzw. *override* deklarierten) für alle Objekte aufgerufen:

```
    private void button1_Click(object sender, EventArgs e)
    {
        decimal brutto = Convert.ToDecimal(textBox1.Text);
        label1.Text = String.Empty;
        for (int i = 0; i < kunden.Length; i++)
        {
            kunden[i].addGuthaben(brutto);
            label1.Text = label1.Text + LF + kunden[i].getAdresse() + LF
                          + kunden[i].Guthaben.ToString("C");
        }
    }
```

Obwohl im Array die Objekte bunt durcheinander gewürfelt sein können, "weiß" das Programm zur Laufzeit genau, welche Implementierung der beiden polymorphen Methoden *getAdresse()* und *addGuthaben()* jeweils für Privat- und für Firmenkunden die richtige ist.

HINWEIS: Genau hier liegt der springende Punkt zum Verständnis der Polymorphie!

BEISPIEL 3.46: Polymorphes Verhalten Variante 2

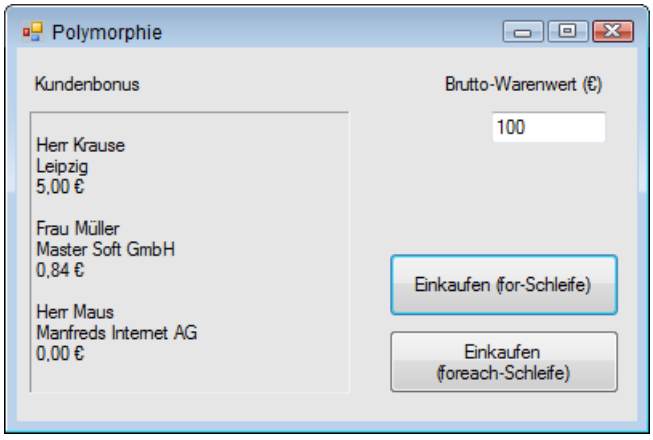
C#

Eine alternative Implementierung mittels *foreach*-Schleife bringt die Polymorphie noch deutlicher ans Tageslicht, da die Methodenaufrufe nicht mit Objekten der Subklassen *CPrivatKunde*/*CFirmenkunde*, sondern mit Objekten der Basisklasse *CKunde* verknüpft sind:

```
private void button2_Click(object sender, EventArgs e)
{
    decimal brutto = Convert.ToDecimal(textBox1.Text);
    label1.Text = String.Empty;
    foreach (CKunde ku in kunden)
    {
        ku.addGuthaben(brutto);
        label1.Text = label1.Text + LF + ku.getAdresse() + LF +
                                              ku.Guthaben.ToString("C");
    }
}
```

Praxistest

Das Ergebnis anhand der abgebildeten Testoberfläche beweist, dass Vererbung und Polymorphie tatsächlich untrennbar miteinander verbunden sind. Egal ob Privat- oder Firmenkunde – es werden immer die jeweils passenden Methodenimplementierungen aufgerufen[1].



Das tiefere Verständnis der Polymorphie ist mit Sicherheit der schwierigste Part der OOP, deshalb wurde unser Beispiel bewusst einfach gehalten, damit Sie zunächst zu einem Grundverständnis gelangen, welches Sie später weiter ausbauen können.

[1] Tja, der arme Herr Maus. Weil er kein Stammkunde ist, bekommt er auch kein Bonusguthaben.

3.6.8 Die Rolle von System.Object

Jedes Objekt in .NET erbt von der Basisklasse *System.Object*. Diese Klasse ist Teil des Microsoft .NET Frameworks und beinhaltet die Basiseigenschaften und -methoden, wie sie für ein .NET-Objekt erforderlich sind.

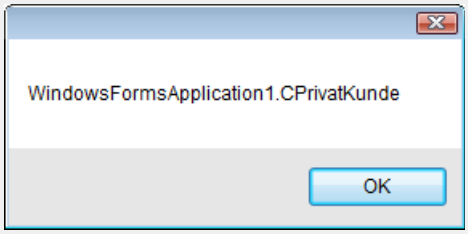
Alle öffentlichen Eigenschaften und Methoden von *System.Object* stehen automatisch auch in jedem Objekt zur Verfügung, welches Sie erzeugt haben. Beispielsweise ist in *System.Object* bereits ein Standardkonstruktor enthalten. Wenn Sie in Ihrem Objekt keinen eigenen Konstruktor definiert haben, wird es mit diesem Konstruktor erzeugt.

Viele der öffentlichen Eigenschaften und Methoden von *System.Object* haben eine Standardimplementation. Das heißt, Sie brauchen selbst keinerlei Code zu schreiben, um sie zu verwenden.

BEISPIEL 3.47: Die *ToString*-Methode liefert den Namen der Anwendungskomponente (die Windows-Anwendung heißt hier *Vererbung1*) und die Klassenzugehörigkeit von *kunde1*.

C#
```
MessageBox.Show(kunde1.ToString());
```

Ergebnis

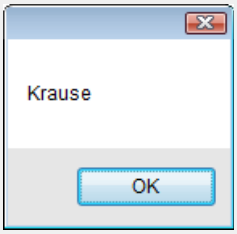


Sie können das standardmäßige Verhalten von *ToString()* mittels *override*-Schlüsselwort verändern. Dies erlaubt Ihnen eine individuelle Implementierung einiger Eigenschaften bzw. Methoden von *System.Object*.

BEISPIEL 3.48: Die gleiche *ToString()*-Methode des Vorgängerbeispiels liefert nun den Namen des Kunden, wenn Sie die folgende Methode zum Klassenkörper von *CKunde* hinzufügen.

C#
```
public override string ToString()
{
   return (_name);
}
```

Ergebnis



3.7 Spezielle Klassen

Es gibt einige Klassen, die spezielle Features aufweisen, bzw. nur für einen beschränkten Einsatzbereich infrage kommen. Gekennzeichnet werden diese Klassen in der Regel durch einen vorangestellten Modifizierer (*abstract*, *sealed*, *partial*, *static*).

3.7.1 Abstrakte Klassen

Klassen, die lediglich ihr "Erbmaterial" an andere Klassen weitergeben und von denen selbst keine Instanzen gebildet werden, bezeichnet man als *abstrakt*. Typische Beispiele für abstrakte Klassen wären *Fahrzeug*, *Tier* oder *Nahrung*[1].

Um zu verhindern, dass von abstrakten Klassen Instanzen gebildet werden, müssen diese mit dem Modifikator *abstract* gekennzeichnet werden.

BEISPIEL 3.49: In unserem Vorgängerbeispiel werden von der Klasse *CKunde* keine Instanzen gebildet, sie kann deshalb als abstrakt deklariert werden.

C#
```
public abstract class CKunde
{ ... }
```

Während die Referenzierung nach wie vor möglich ist

```
CKunde kunde;
```

schlägt der Versuch einer Instanziierung fehl:

```
kunde = new CKunde("Herr", "Krause");              // Fehler
```

Abstrakte Klassen ähneln einem weiteren wichtigen Softwarekonstrukt der OOP, den Schnittstellen (Interfaces). Eine Schnittstelle müssen Sie sich wie eine abstrakte Klasse vorstellen, in welcher nur öffentliche Methoden definiert, aber nicht implementiert werden (die Methodenrümpfe bleiben leer). Die Deklaration einer Schnittstelle ähnelt der einer Klasse, nur dass das Schlüsselwort *class* gegen das Schlüsselwort *interface* ausgetauscht wird (siehe Abschnitt 3.8).

Abstrakte Methoden

In Verbindung mit polymorphem Verhalten finden sich innerhalb abstrakter Klassen oft auch *abstrakte Methoden*, diese enthalten grundsätzlich keinen Code, da sie in den abgeleiteten Klassen komplett mit *override* überschrieben werden. Zur Kennzeichnung abstrakter Methoden verwenden Sie das Schlüsselwort *abstract*. Die Deklaration erfolgt in einer Zeile, also ohne Rumpf.

BEISPIEL 3.50: Die Funktion *adresse()* der abstrakten *CKunde*-Klasse wird in der Subklasse *CPrivatKunde* komplett implementiert.

C#
```
public abstract class CKunde
{
    ...
```

[1] Können vielleicht Sie sich vorstellen, wie eine Instanz von *Nahrung* aussehen soll?

BEISPIEL 3.50: Die Funktion *adresse()* der abstrakten *CKunde*-Klasse wird in der Subklasse *CPrivatKunde* komplett implementiert.

```
    public abstract string adresse();            // abstrakte Methode
    ...
}
```

Da eine abstrakte Methode implizit eine virtuelle Methode darstellt, muss sie in der Subklasse mit *override* deklariert werden.

```
public class CPrivatKunde : CKunde;
{
    ...
    public override string adresse()          // überschreibt abstrakte Methode
    {
         return(_anrede + "  " + _name + "  " & _wohnort)
    }
    ...
}
```

3.7.2 Versiegelte Klassen

Wenn Sie unbedingt verhindern möchten, dass andere Programmierer von einer von Ihnen entwickelten Komponente weitere Subklassen ableiten, so müssen Sie Ihre Klasse mit Hilfe des Modifikators *sealed* schützen.

BEISPIEL 3.51: Die Klasse *CPrivatKunde* wird versiegelt und darf deshalb keine Nachkommen haben.

C#
```
public sealed class CPrivatKunde : CKunde
{    ...
}
```

Beim Versuch, davon eine Subklasse abzuleiten, schlägt Ihnen der Compiler erbarmungslos auf die Pfoten:

```
public class CStudent : CPrivatKunde          // Fehler!!!
{
...
}
```

HINWEIS: Vererbungsmodifikatoren wie *abstract* und *virtual* führen in einer versiegelten Klasse zum Compilerfehler, da sie keinen Sinn ergeben!

Übrigens: Ein bekanntes Beispiel für eine versiegelte Klasse ist der *string*-Datentyp, was jedweden Begehrlichkeiten einen Riegel vorschiebt.

3.7.3 Partielle Klassen

Das Konzept partieller Klassen ermöglicht es, den Quellcode einer Klasse auf mehrere einzelne Dateien aufzusplitten. In Visual Studio wird zum Beispiel auf diese Weise der vom Designer automatisch erzeugte Layout-Code vom Code des Entwicklers getrennt, was zu einer gesteigerten Übersichtlichkeit beiträgt, wovon man sich nach Öffnen eines neuen Windows Forms Projekts selbst überzeugen kann (siehe auch neues Code-Beside-Modell von ASP.NET).

Die Programmierung ist denkbar einfach, denn alle Teile der Klasse sind lediglich mit dem Modifizierer *partial* zu kennzeichnen, dieser muss hinter dem Sichtbarkeitsmodifizierer (*private, public, protected*, ...) stehen.

BEISPIEL 3.52: Eine einfache Klasse *CKunde*

C#

```
public class CKunde
{
        private string _name;
        protected decimal _guthaben = 0;

        public CKunde(string nachName)
        { _name = nachName; }

        public string Name
        { get { return (_name); } }

        public decimal Guthaben
        { get { return (_guthaben); } }

        public void addGuthaben(decimal betrag)
        { _guthaben += betrag; }
}
...
```

Die Klasse *CKunde* könnte (als eine von vielen Möglichkeiten) wie folgt in drei partielle Klassen aufgesplittet werden:

```
public partial class CKunde
{
        private string _name;
        protected decimal _guthaben = 0;

        public CKunde(string nachName)
        { _name = nachName; }
}

public partial class CKunde
{
        public string Name
        { get { return (_name); } }
```

BEISPIEL 3.52: Eine einfache Klasse *CKunde*

```
        public decimal Guthaben
        { get { return (_guthaben); } }
}

public partial class CKunde
{
        public void addGuthaben(decimal betrag)
        { _guthaben += betrag; }
}
```

3.7.4 Statische Klassen

Mit dem *static*-Modifizierer kann man nicht nur statische Klassenmitglieder (Eigenschaften, Methoden, siehe 3.2.5 und 3.3.3), sondern auch komplette *statische* Klassen deklarieren. Eine solche Klasse kann nicht instanziiert werden und hat nur statische Mitglieder. Praktische Anwendungen ergeben sich z.B. für Formelsammlungen. Auch in der .NET-Klassenbibliothek finden sich zahlreiche vordefinierte statische Klassen (z.B. *File*, *Directory*, *Debug*, ...).

BEISPIEL 3.53: Eine statische Klasse zur Berechnung des Kugelvolumens bei gegebenem Durchmesser (und umgekehrt)

C#

```
public static class CKugel
{
    private static double Pi = Math.PI;

    public static double Durchmesser_Volumen(double durchmesser)
    {
        double vol = Math.Pow(durchmesser, 3) * Pi/6.0;
        return vol;
    }

    public static double Volumen_Durchmesser(double volumen)
    {
        double dur = Math.Pow(6/Pi * volumen, 1/3.0);
        return dur;
    }
}
```

Beispielhafte Anwendung der statischen Klasse *CKugel* zur Berechnung des Durchmessers einer Kugel mit 1 Kubikmeter Rauminhalt:

```
double v = 1.0d;
double d = CKugel.Volumen_Durchmesser(v);          // liefert 1,24...
```

Die Verwendung einer ähnlichen Klasse wird im folgenden Praxisbeispiel demonstriert:

▶ 3.9.2 Eine statische Klasse anwenden

3.8 Schnittstellen (Interfaces)

Das .NET-Framework (die CLR) unterstützt keine Mehrfachvererbung, d.h., eine Unterklasse kann immer nur von einer einzigen Oberklasse erben. Dies ist wohl mehr ein Segen als ein Fluch, denn allzu leicht würde sonst der Programmierer im Gestrüpp mehrfacher Vererbungsbeziehungen über mehrere Hierarchie-Ebenen hinweg die Übersicht verlieren, instabiler Code und Chaos wären die Folge.

Ein Ausweg ist die Verwendung von Schnittstellen, diese bieten fast alle Möglichkeiten der Mehrfachvererbung, vermeiden aber deren Nachteile. Schnittstellen dienen dazu, um gemeinsame Merkmale ansonsten unabhängiger Klassen beschreiben zu können.

Eine Schnittstelle können Sie sich zunächst wie eine abstrakte Klasse (siehe 3.7.1) vorstellen, in welcher nur abstrakte Methoden definiert werden[1].

3.8.1 Definition einer Schnittstelle

Eine Schnittstelle können Sie zu Ihrem Projekt genauso hinzufügen wie eine neue Klasse. Statt des Schlüsselworts *class* verwenden Sie aber *interface*.

HINWEIS: Laut Konvention sollte der Namen einer Schnittstelle mit "I" beginnen.

BEISPIEL 3.54: Eine Schnittstelle *IPerson*, die zwei Eigenschaften und eine Methode definiert

C#
```
public interface IPerson
{
    string Nachname
    { get; set; }

    string Vorname
    { get; set; }

    string getName();
}
```

Vielleicht vermissen Sie im obigen Beispiel die Zugriffsmodifizierer (*public string Nachname ...*), diese haben aber in einer Schnittstellendefinition generell nichts zu suchen.

HINWEIS: Die Festlegung der Zugriffsmodifizierer für die Mitglieder der Schnittstelle ist allein Angelegenheit der Klasse, die die Schnittstelle implementiert!

[1] Dieser Vergleich hinkt natürlich wegen der auch bei abstrakten Klassen nicht möglichen Mehrfachvererbung.

3.8.2 Implementieren einer Schnittstelle

Die Syntax entspricht der bei der normalen Implementierurngsvererbung[1], d.h., an die Deklaration der erbenden Klasse wird ein Doppelpunkt angefügt, dem der Namen der Schnittstelle folgt.

HINWEIS: Die implementierende Klasse geht die Verpflichtung ein, ausnahmslos **alle** Mitglieder der Schnittstelle zu implementieren!

BEISPIEL 3.55: Die Klasse *CKunde* implementiert die Schnittstelle *IPerson*

C#

```
class CKunde : IPerson
{
       private string _nachname;
       private string _vorname;
       ...
```

Alle von *IPerson* geerbten abstrakten Klassenmitglieder müssen implementiert werden:

```
       public string Nachname
       {
           get { return _nachname; }
           set { _nachname = value; }
       }

       public string Vorname
       {
           get { return _vorname; }
           set { _vorname = value; }
       }

       public override string  getName()
       {
          return _vorname + " " +_nachname ;
       }
```

Es folgen die normalen Klassenmitglieder:

```
       ...
}
```

Den kompletten Code finden Sie Im Praxisbeispiel

► 3.9.4 Schnittstellenvererbung verstehen

Dort wird auch gezeigt, wie man eine mit abstrakten Methoden ausgestattete abstrakte Klasse ganz leicht in eine Schnittstelle überführen kann.

[1] Der Vergleich trifft zumindest dann zu, wenn nur eine einzige Schnittstelle implementiert wird.

3.8.3 Abfragen, ob Schnittstelle vorhanden ist

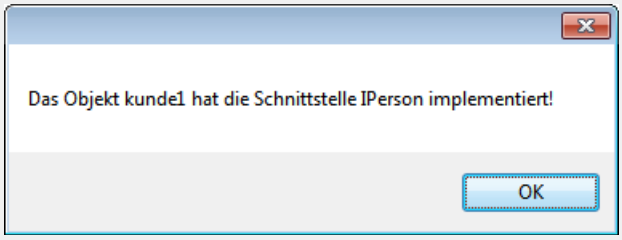
Manchmal möchte man vor der eigentlichen Arbeit mit einem Objekt wissen, ob dieses eine bestimmte Schnittstelle implementiert hat. Eine Lösung bietet eine Abfrage mit dem *is*-Operator.

BEISPIEL 3.56: Abfragen, ob das Objekt *kunde1* die Schnittstelle *IPerson* implementiert hat

C#

```
private void button2_Click(object sender, EventArgs e)
{
  if (kunde1 is IPerson)
      MessageBox.Show("Das Objekt kunde1 hat die Schnittstelle IPerson implementiert!");
}
```

Ergebnis



3.8.4 Mehrere Schnittstellen implementieren

Eine Klasse kann nicht nur eine, sondern auch mehrere Schnittstellen gleichzeitig implementieren, was quasi Mehrfachvererbung bedeutet, wie sie mit der klassischen Implementierungsvererbung unmöglich ist.

BEISPIEL 3.57: Eine Klasse implementiert zwei Schnittstellen

C#

```
public class CPrivatkunde : IPerson, IKunde
{
    ...
}
```

3.8.5 Schnittstellenprogrammierung ist ein weites Feld

... und bis jetzt haben wir nur an der Oberfläche gekratzt. Wichtige Prinzipien hier nochmals in Kürze:

- Statt von einer abstrakten Klasse zu erben, werden die abstrakten Methoden über eine Schnittstelle veröffentlicht. Damit erlangt man gewissermaßen die Funktionalität der Mehrfachvererbung und umgeht deren Nachteile.
- Eine Schnittstelle ist wie ein Vertrag: Sobald eine Klasse eine Schnittstelle implementiert, muss sie auch ausnahmslos alle (!) Mitglieder der Schnittstelle implementieren und veröffentlichen.

- Der Name der implementierten Methode sowie deren Signatur muss mit deren Definition in der Schnittstelle exakt übereinstimmen.
- Mehrere Schnittstellen können zu einer neuen Schnittstelle zusammengefasst werden und selbst wieder Schnittstellen implementieren.

HINWEIS: Mehr zur Schnittstellenprogrammierung finden Sie beispielsweise im Kapitel 5 (*IEnumerable*-Interface) oder im Kapitel 25 (Klassen-Designer).

3.9 Praxisbeispiele

3.9.1 Eigenschaften sinnvoll kapseln

Das Deklarieren von Eigenschaften als öffentliche Variablen der Klasse heißt immer, das Brett an der dünnsten Stelle zu bohren. Der fortgeschrittene Programmierer verwendet stattdessen sogenannte Property-Methoden, die einen kontrollierten Zugriff erlauben. Außerdem ermöglichen die Property-Methoden auch die Implementierung von *berechneten Eigenschaften*, die aus den (privaten) Zustandsvariablen ermittelt werden. Im vorliegenden Rezept handelt es sich um eine Klasse *CKreis* mit den Eigenschaften *radius*, *umfang* und *fläche*. Diese Klasse speichert intern eine einzige Zustandsvariable *r*, aus welcher direkt beim Zugriff alle Eigenschaften berechnet werden.

Oberfläche

Um einen großen Vorteil der OOP zu demonstrieren (ohne viel Mehraufwand lassen sich beliebig viele Instanzen einer Klasse erzeugen), wollen wir mit zwei Objekten (*kreis1* und *kreis2*) arbeiten.

Quellcode (CKreis)

Unsere Klasse wird außerhalb von *Form1* definiert, wir werden sie sogar in ein separates Klassenmodul auslagern. Wählen Sie das Menü *Projekt|Klasse hinzufügen...* und geben Sie dem Klassenmodul den Namen *CKreis.cs*.

Deklarieren und implementieren Sie nun die Klasse wie folgt:

```
public class CKreis
{
  private double r;                    // das einzige Feld (Zustandsvariable)
```

Die Rückgabewerte der Eigenschaften sind hier vom *string*-Datentyp, um die Bedienoberfläche von den lästigen Konvertierungen *string* in *double* und umgekehrt zu entlasten.

Die Eigenschaft *radius*:

```
  public string radius
  {
     get {return (r.ToString("#,#0.00")); }
     set
     {
       if (value != "")  r = Convert.ToDouble(value);
       else r = 0;
     }
  }
```

Die Eigenschaft *umfang*:

```
  public string umfang
  {
     get {return (2 * Math.PI * r).ToString("#,#0.00"); }
     set
     {
       if (value != "")  r = Convert.ToDouble(value) / 2 / Math.PI;
       else r = 0;
     }
  }
```

Die Eigenschaft *fläche*:

```
  public string fläche
  {
     get {return (Math.PI * Math.Pow(r, 2)).ToString("#,#0.00");}
     set
     {
       if (value != "") r = Math.Sqrt(Convert.ToDouble(value) / Math.PI);
       else r = 0;
     }
  }
```

Der eigene Konstruktor:

```
  public CKreis(double rad)
```

```
   { r = rad;  }
}
```

Quellcode (Form1)

Wechseln Sie in den Klassencode von *Form1*.

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

Ein Objekt *kreis1* wird erzeugt und kann (ein Dankeschön an den Konstruktor) gleich mit dem Radius 1.0 (Maßeinheit soll hier keine Rolle spielen) initialisiert werden:

```
CKreis kreis1 = new CKreis(1.0);
```

Die folgenden Eventhandler sind einfach und übersichtlich, da die Objekte ihre inneren Funktionalitäten kapseln:

```
    private void textBox1_KeyUp(object sender, KeyEventArgs e)
    {
        kreis1.radius = textBox1.Text;
        textBox2.Text = kreis1.umfang;
        textBox3.Text = kreis1.fläche;
    }

    private void textBox2_KeyUp(object sender, KeyEventArgs e)
    {
        kreis1.umfang = textBox2.Text;
        textBox1.Text = kreis1.radius;
        textBox3.Text = kreis1.fläche;
    }

    private void textBox3_KeyUp(object sender, KeyEventArgs e)
    {
        kreis1.fläche = textBox3.Text;
        textBox1.Text = kreis1.radius;
        textBox2.Text = kreis1.umfang;
    }
    ...
}
```

Der Code für das zweite Objekt (*kreis2*) ist völlig analog zu *kreis1*, sodass hier auf die Wiedergabe des Quellcodes verzichtet werden kann (siehe Beispieldaten zum Buch).

Test

Sobald Sie eine beliebige Eigenschaft ändern, werden die anderen zwei sofort aktualisiert! Wegen der in der Klasse eingebauten Eingabeprüfung verursacht ein leerer Eingabewert keinen Fehler.

HINWEIS: Geben Sie als Dezimaltrennzeichen immer das Komma (,) ein, als Tausender-Separator dürfen Sie den Punkt (.) verwenden.

Bemerkungen

- Aus Gründen der Übersichtlichkeit wurde aber auf das Abfangen weiterer Eingaben, die sich nicht in einen numerischen Wert konvertieren lassen, verzichtet.
- Ab C# 3.0 kann man ein Objekt auch dann erzeugen und initialisieren, wenn es dazu keinen passenden Konstruktor gibt. Sie könnten also auf den Konstruktor im Code von *CKreis* verzichten und stattdessen im Code von *Form1* die Instanziierung der Klasse durch direktes Zuweisen ihrer Eigenschaften (nicht private Zustandsvariablen!) mittels eines Objektinitialisierers wie folgt vornehmen:

```
CKreis kreis1 = new CKreis { radius = "1.0" };
```

3.9.2 Eine statische Klasse anwenden

Statische Klassen eignen sich ideal für Formelsammlungen (siehe z.B. *Math*-Klasse), da keine Objekte erzeugt werden müssen, denn es kann gleich "losgerechnet" werden. Das vorliegende Rezept demonstriert dies anhand einer statischen Klasse *CKugel* zur Berechnung des Kugelvolumens bei gegebenem Durchmesser (und umgekehrt)[1].

$V = 4/3 * Pi * r^3$

Nimmt man anstatt des Radius den Durchmesser *d* der Kugel, so ergibt sich daraus nach einigen Umstellungen die folgende Berechnungsformel für das Volumen *V*:

$V = d^3 * Pi/6$

Oberfläche

Lediglich ein Formular mit zwei Textfeldern zur Eingabe von Kugeldurchmesser und Kugelvolumen ist erforderlich (siehe Laufzeitansicht).

Quellcode

```
public static class CKugel
// public class CKugel          // das würde auch gehen!
{
        public static double Durchmesser_Volumen(string durchmesser)
        {
            double dur = System.Double.Parse(durchmesser);
            double vol = Math.Pow(dur, 3) * Math.PI/6.0;
            return vol;
        }

        public static double Volumen_Durchmesser(string volumen)
        {
            double vol = System.Double.Parse(volumen);
            double dur = Math.Pow(6/Math.PI * vol, 1/3.0);
```

[1] Alter Merkspruch zum Kugelvolumen: "Bedächtig kommt sie einher geschritten – vier Drittel Pi mal R zur Dritten!"

```
            return dur;
        }
}
```

Die Verwendung der Klasse im Formularcode:

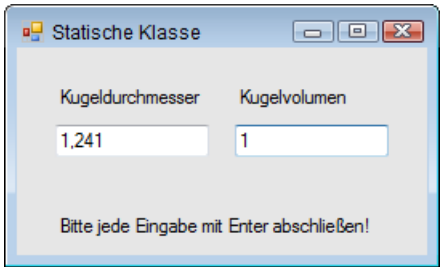
```
public partial class Form1 : Form
{  ...
```

Die Berechnung startet nach Betätigen der Enter-Taste:

```
        private void textBox1_KeyUp(object sender, KeyEventArgs e)
        {
            if ((e.KeyCode == Keys.Enter) && (textBox1.Text != ""))
            {
                textBox2.Text = CKugel.Durchmesser_Volumen(textBox1.Text).ToString("#,##0.000");
            }
        }

        private void textBox2_KeyUp(object sender, KeyEventArgs e)
        {
            if ((e.KeyCode == Keys.Enter) && (textBox2.Text != ""))
            {
                textBox1.Text = CKugel.Volumen_Durchmesser(textBox2.Text).ToString("#,##0.000");
            }
        }
}
```

Test

Es ist egal, ob Sie den Radius oder das Volumen eingeben. Nach Betätigen der *Enter*-Taste wird der Inhalt des jeweils anderen Textfelds sofort aktualisiert.



Die Maßeinheit spielt bei der Programmierung keine Rolle, da sie für beide Eingabefelder identisch ist. Um beispielsweise einen Wasserbehälter mit 1 Kubikzentimeter Inhalt zu realisieren, ist eine Kugel mit dem Durchmesser von 1,241 Zentimetern erforderlich, für 1 Kubikmeter (1000 Liter) wären es 1,241 Meter: