walter DOBERENZ
thomas GEWINNUS

VISUAL BASIC 2012

GRUNDLAGEN UND PROFIWISSEN

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

EXTRA: Kostenloses E-Book inkl. 1000 Seiten Bonuskapitel

HANSER

Doberenz/Gewinnus

Visual Basic 2012
Grundlagen und Profiwissen

Walter Doberenz
Thomas Gewinnus

Visual Basic 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.

http://www.hanser-fachbuch.de

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 978-3-446-43429-5

E-Book-ISBN 978-3-446-43522-3

Inhaltsverzeichnis

Teil II: Technologien

Teil III: WPF-Anwendungen

Teil IV: Windows Store Apps

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

Die Zeit, in der Visual Basic-Programmierer meinten, mit ein paar Klicks auf ein Formular und mit wenigen Zeilen Quellcode eine vollständige Applikation erschaffen zu können, ist zumindest seit Anbruch der .NET-Epoche endgültig vorbei. Vorbei ist aber auch die Zeit, in der mancher C-Programmierer mitleidig auf den VB-Kollegen herabblicken konnte. VB ist seit langem zu einem vollwertigen professionellen Werkzeug zum Programmieren beliebiger Komponenten für das Microsoft .NET Framework geworden, beginnend bei Windows Forms- über WPF-, ASP.NET-Anwendungen bis hin zu systemnahen Applikationen.

Das vorliegende Buch zu Visual Basic 2012 soll ein faires Angebot sowohl für künftige als auch für fortgeschrittene VB-Programmierer sein. Seine Philosophie knüpft an die vielen anderen Titel an, die wir in den vergangenen siebzehn 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 Visual Basic 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.

Das ist auch der Grund, warum das vorliegende Buch keinen ausgesprochenen Lehrbuchcharakter trägt, sondern mehr ein mit sorgfältig gewählten Beispielen durchsetztes Nachschlagewerk der wichtigsten Elemente der .NET-Programmierung unter Visual Basic 2012 ist.

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 Basic 2012 zu liefern oder all die Informatio-

nen 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 Schneise durch den Urwald der .NET-Programmierung mit Visual Basic 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 Basic 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 des Inhaltsverzeichnisses mit dem Vorgängertitel zeigt, dass aus den ursprünglich 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 Basic 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 VB-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 dabei helfen, die nötigen Grundkenntnisse zu gewinnen, um eigene Apps zu entwickeln. Dabei sind neben den fünf WinRT-Kapiteln auch die WPF-Kapitel über die entsprechenden Basistechnologien (XAML/Datenbindung) von Wichtigkeit.

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 Basic 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 C# 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 Basic 2012".

Zu den Codebeispielen

Alle Beispieldaten dieses Buchs können Sie sich unter folgender Adresse herunterladen:

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

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.
- Bei der Fehlermeldung "Der Microsoft.Jet.OLEDB.4.0-Provider ist nicht auf dem lokalen Computer registriert." müssen Sie als Zielplattform für das Projekt x86 wählen, da es sich bei OLEDB um einen 32-Bit-Treiber handelt.
- 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

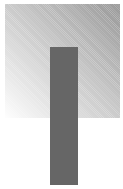
Sie werden – trotz der rund 2200 Seiten – in diesem Buch nicht alles finden, was Visual Basic 2012 bzw. das .NET Framework 4.5 zu bieten haben. Manches ist sicher in einem anderen Spezialtitel noch besser oder ausführlicher beschrieben. Aber Sie halten mit unserem Buch einen optimalen und überschaubaren 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 Visual Basic**
- **Objektorientiertes Programmieren**
- **Arrays, Strings und Funktionen**
- **Weitere wichtige Sprachfeatures**
- **Einführung in LINQ**

Kapitel 1

Einstieg in Visual Studio 2012

Dieses Kapitel bietet dem VB-Programmierer 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 bekanntlich nur durch Beispiele, die man unmittelbar selbst am Rechner ausprobiert!

HINWEIS: Voraussetzung für ein erfolgreiches Studium dieses Buchs ist das Vorhandensein eines PCs 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 Debug-

ging, 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 Visual Basic auch Vorlagen für C#, 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 Visual Basic stehen Ihnen dazu auch noch C# 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 VB-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 Visual Basic-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 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 VB-Kommandozeilencompiler *vbc.exe*.

Compilerpfad eintragen

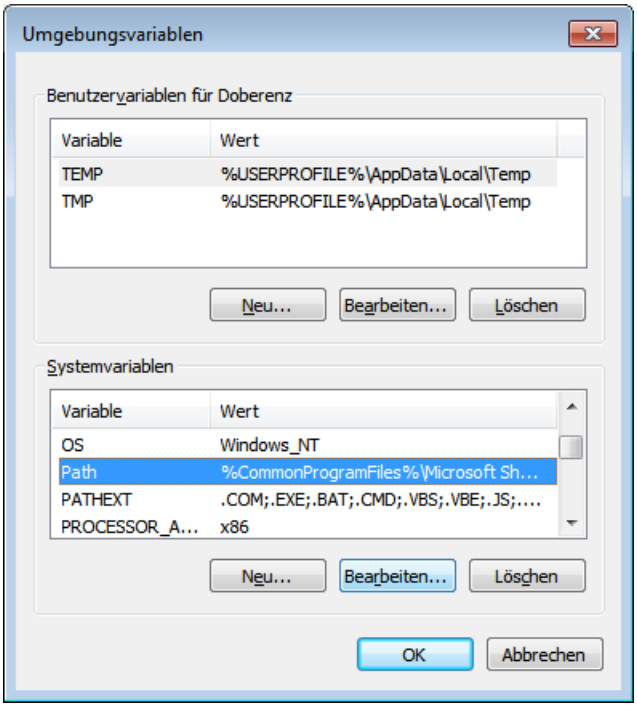
Der VB-Compiler *vbc.exe* befindet sich, ziemlich versteckt, im Verzeichnis

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

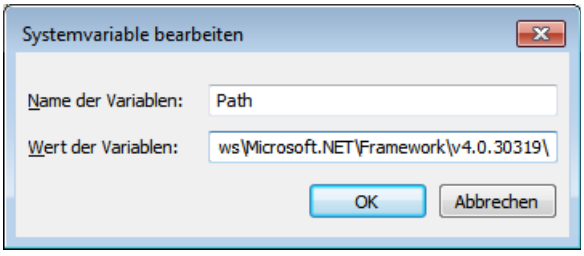
Da das Kompilieren direkt an der Kommandozeile ausgeführt werden soll, werden wir *vbc.exe* in den Windows-Pfad aufnehmen, um so seinen Aufruf von jedem Ordner des Systems aus zu ermöglichen:

- 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 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 *vbc.exe* befindet (*C:\Windows\Microsoft.NET\Framework\v4.0.30319*), durch ein Semikolon (;) getrennt hinten dran:

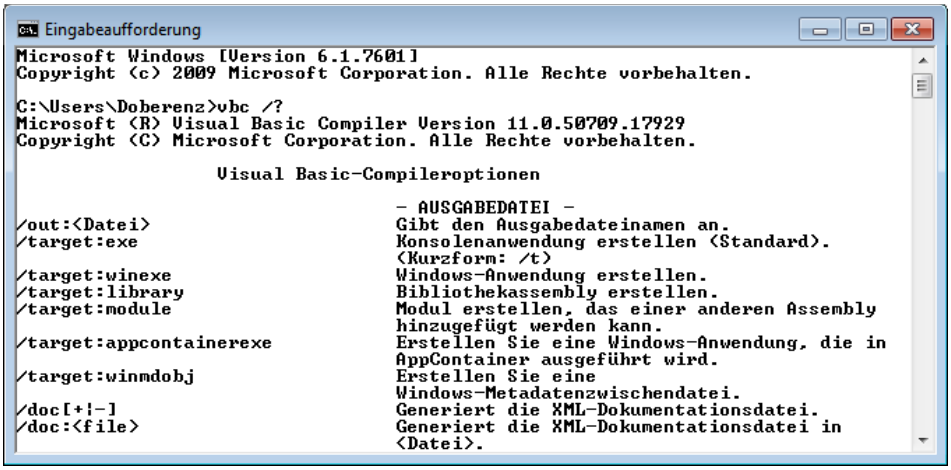


Die erfolgreiche Übernahme der Änderungen an den *Path*-Umgebungsvariablen können Sie in einem kleinen Test überprüfen, bei dem Sie sich als durchaus nützlichen Nebeneffekt gleich die vielfältigen Optionen des Compilers anzeigen lassen.

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:

```
vbc /?
```

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 VB-Source-Datei erstellt haben (siehe folgender Abschnitt).

1.2.2 Programm schreiben

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

```
Imports System
Module KonsolenDemo
    Sub Main()
        Dim kurs, euro, dollar As Single, b As Char
        Console.WriteLine("Umrechnung Euro in Dollar")
        Do
            Console.Write("Kurs 1 : ")
            kurs = CSng(Console.ReadLine())
            Console.Write("Euro: ")
            euro = CSng(Console.ReadLine())
            dollar = euro * kurs
            Console.WriteLine("Sie erhalten " & dollar.ToString("0.00 Dollar"))
            Console.Write("Programm beenden? (j/n)")
            b = CChar(Console.ReadLine())
```

```
        Loop While b <> "j"
    End Sub
End Module
```

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

1.2.3 Programm kompilieren und testen

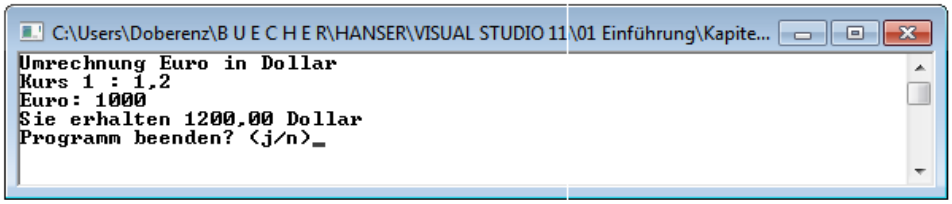
Um bequem an der Kommandozeile arbeiten zu können, kopieren Sie zunächst die Datei *cmd.exe* (Eingabeaufforderung aus ...*\Windows\System32*) in dasselbe Verzeichnis, in welchem sich auch die Datei *EuroDollar.vb* befindet.

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

```
vbc /t:exe EuroDollar.vb
```

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 VB-Programm aus!



HINWEIS: Ist die eingegebene Zahl komplett, so ist mittels Enter-Taste abzuschließen!

1.2.4 Einige Erläuterungen zum Quellcode

Da wir auf die Grundlagen der Sprache Visual Basic erst in den späteren Kapiteln ausführlich zu sprechen kommen, sollen einige Vorabinformationen den größten Wissensdurst stillen.

Befehlszeilen

Das Ende einer VB-Befehlszeile wird in der Regel durch einen Zeilenumbruch markiert. Die auszuführenden Befehlszeilen befinden sich innerhalb der *SubMain*-Prozedur, die mit *End Sub* abgeschlossen wird.

Imports-Anweisung

Mit der ersten Anweisung

```
Imports 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.

Noch kürzer wird es mit

```
Imports System.Console
```

denn dann würde die folgende Anweisung genügen:

```
WriteLine("Umrechnung Euro in Dollar")
```

Module-Anweisung

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

HINWEIS: Ein VB-Programm besteht aus mindestens einer **.vb*-Textdatei mit einem Modul (oder 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 VB-Compiler erzeugten Datei *EuroDollar.exe* handelt es sich **nicht** um eine herkömmliche Exe-Anwendung, 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).

1.2.5 Konsolenanwendungen sind langweilig

Zwar hat seit Visual Basic 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 – außer nostalgischen Erinnerungen an die DOS-Steinzeit – noch positive Emotionen?

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

Mit Visual Studio 2012 werden wir im Praxisteil dieses Kapitels (Abschnitt 1.7.2) das gleiche Problem lösen, diesmal allerdings mit einer attraktiven Windows-Oberfläche. Bevor es aber so richtig losgehen kann, sollten wir uns zunächst ein wenig mit der Windows-Philosophie anfreunden.

1.3 Die Windows-Philosophie

Eine moderne Programmiersprache wie Visual Basic 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 einfache 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 VB mit folgenden Objekten zu tun:

- **Formulare** Das sind die Fenster, in welchen Ihre VB-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* (Event-Handler).

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 Windows-Programmierung unter 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 notwendig) in ihrer Größe verändert.

Bereits während der ersten Etappe hat man – mehr oder weniger unbewusst – Eigenschaften verändert, wie z.B.Position und Abmessungen von Formularen und Steuerelementen. 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* (Event-Handlern) 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) für alle zum jeweiligen Objekt passenden Ereignisse zur Verfügung. Der Programmierer füllt diesen Rahmen mit VB-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 VB-Compiler in einen Zwischencode übersetzt und läuft damit auf jedem Rechner, auf dem das .NET-Framework installiert ist.

---

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

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.

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

HINWEIS: Die folgenden kurzen Erklärungen sollen lediglich einen allerersten Eindruck der IDE vermitteln, der sich erst durch die konkrete Arbeit mit den Praxisbeispielen am Ende dieses Kapitels verfestigen wird!

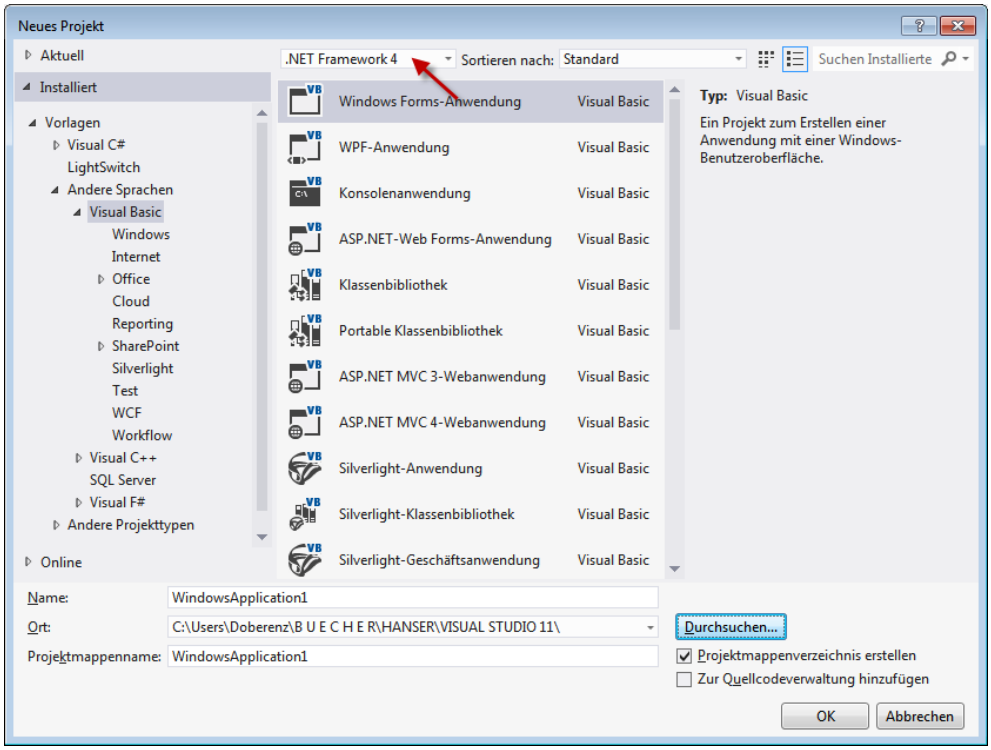
1.4.1 Der Startdialog

Wählen Sie auf der Startseite von Visual Studio 2012 den Menüpunkt *Neues Projekt...,* so öffnet sich der Startdialog *Neues Projekt*. Unter der Rubrik *Andere Sprachen* (links) werden Sie mit einem umfangreichen und zunächst verwirrenden Angebot an unterschiedlichen Vorlagen[2] für Visual Basic-Projekttypen konfrontiert, wobei für den Einsteiger zunächst die klassische *Windows Forms-Anwendung* empfohlen wird.

Wie Sie es an der linken oberen Klappbox sehen, kann man Programme für verschiedene .NET-Framework-Versionen entwickeln (Multi-Targeting).

[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.



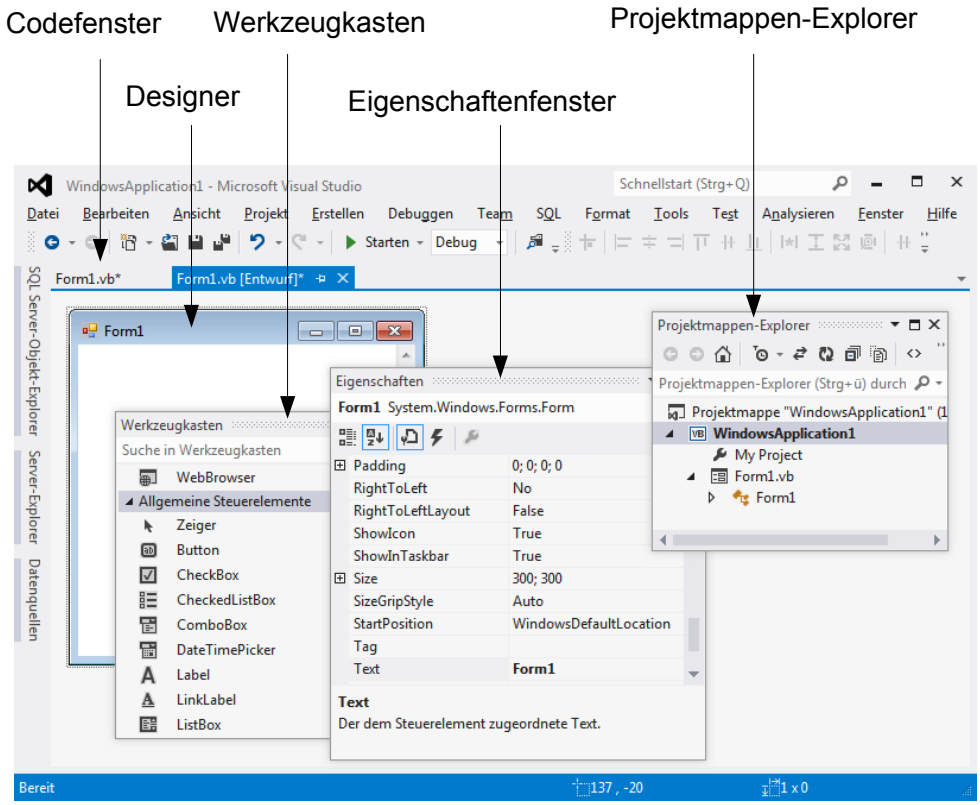
Haben Sie das Häkchen bei *Projektmappenverzeichnis erstellen* gesetzt, so erzeugt Visual Studio automatisch einen Unterordner mit dem Namen des Projekts in dem als Speicherort eingetragenen Hauptverzeichnis.

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 etwa den folgenden Anblick bieten, wobei auf die für den Einsteiger zunächst wichtigsten Fenster besonders hingewiesen wird.

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



Der Projektmappen-Explorer

Da es unter Visual Studio 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 auch als "Schaltzentrale" Ihres Projekts betrachten.

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

Zur Bedeutung der einzelnen Einträge:

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

- *Form1.vb*
  Diese Datei enthält eine partielle Klasse[1], die den von Ihnen selbst hinzugefügten Code von *Form1* kapselt.

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-Anwendungen.

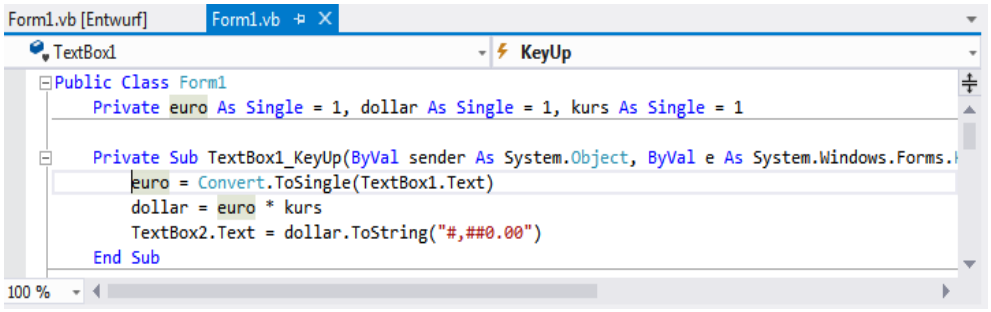
Das Eigenschaften-Fenster

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.

Das Codefenster

Für die eigentliche Programmierung ist das Codefenster zuständig. Logischerweise wird dies damit auch zu Ihrem Hauptbetätigungsfeld als VB-Programmierer. Die folgende Abbildung zeigt einen Ausschnitt des Codefensters für das Praxisbeispiel 1.7.2.



[1] Das Konzept partieller Klassen wird unter anderem in Kapitel 3 erläutert.

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

Sie öffnen das Codefenster, indem Sie z.B. im Projektmappen-Explorer oder im Designer auf einen Eintrag bzw. ein Objekt mit der rechten Maustaste klicken und im Kontextmenü *Code anzeigen* wählen (F7).

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 wir 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 sollen Sie sich mit der grundlegenden .NET-Philosophie und den damit verbundenen Konzepten, Begriffen und Features anfreunden. 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 in den letzten Jahren dramatisch verändert haben, wobei das 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.

Vom alten VB zu VB.NET

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.

Als konsequent objektorientierte Sprache erfüllt VB.NET 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).

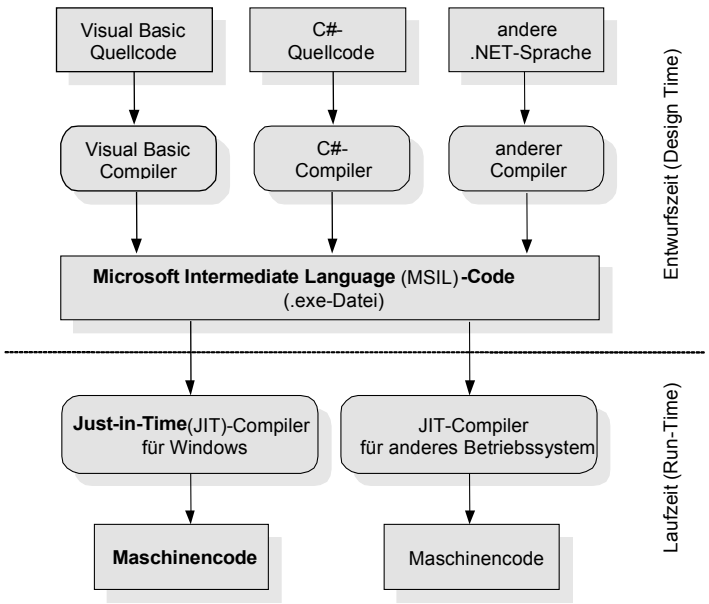
Die einst hoch gelobte COM[1]-Technologie wurde zum Auslaufmodell erklärt. Microsoft kann natürlich nicht diese Technologie auf die Müllkippe entsorgen, denn zu viele Programmierer würden dadurch verprellt werden. Aus diesem Grund wird COM auch in .NET noch lange Zeit sein Gnadenbrot erhalten.

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. Ein .NET-Programm wird also vom Entwurf bis zu seiner Ausführung auf dem Zielrechner tatsächlich zweimal kompiliert (siehe folgende Abbildung).



[1] *Component Object Model*

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.

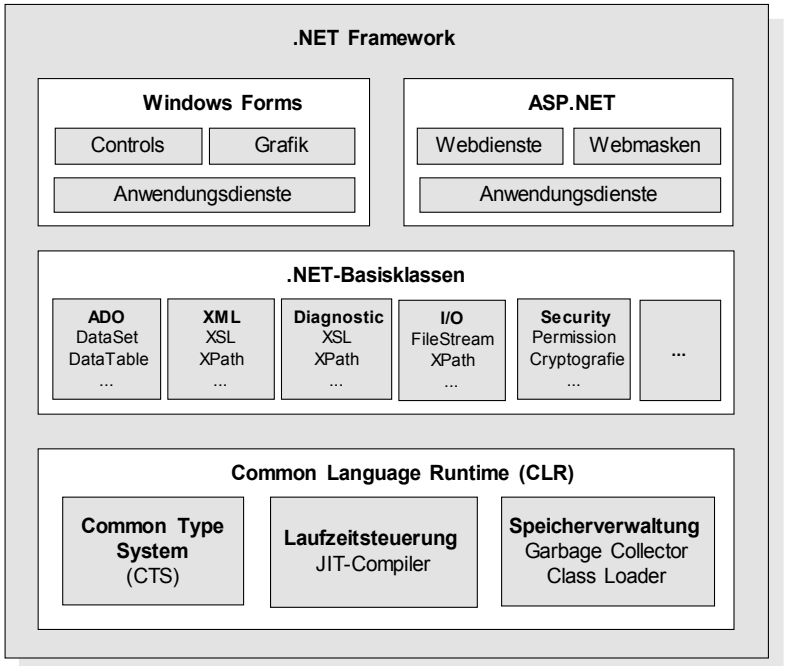
1.5.2 .NET-Features und Begriffe

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:

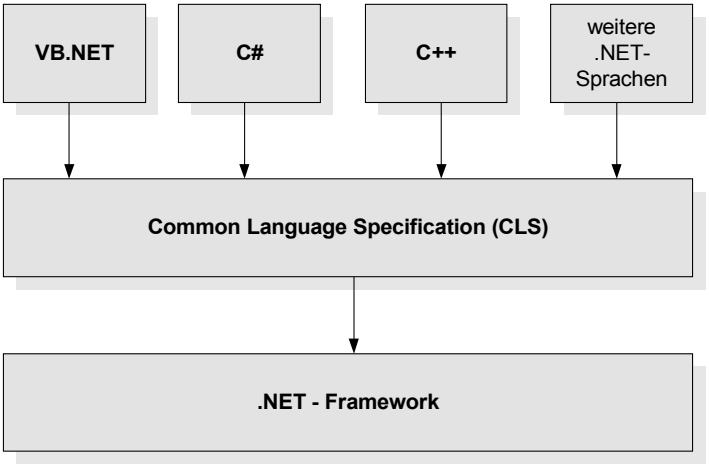
- Common Language Specification (CLS)
- Common Type System (CTS)

- Common Language Runtime (CLR)
- .NET-Klassenbibliothek
- diverse Basisklassenbibliotheken wie ADO.NET und ASP.NET
- diverse Compiler z.B. für VB, C# ...

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.

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.

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

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

oder Delphi.NET 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. Anstatt 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 den Einbau der zusätzlichen Interoperabilitätsschicht sinkt die Performance meist deutlich ab.

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 ver-

zichten 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 auch eine Liste 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 der .NET-Reflector.

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

Attribute

Wer noch in älteren objektorientierten Sprachen (z.B. VB 6, Delphi 7) zu Hause ist, 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 pur

Last, but not least wollen wir am Ende unserer kleinen Rundreise durch 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 Visual Basic fortzufahren ohne sich vorher mit dem Konzept der OOP vertraut gemacht zu haben (siehe Kapitel 3).

1.6 Wichtige Neuigkeiten in Visual Studio 2012

Den Lesern, die bereits mit der Vorgängerversion (Visual Studio 2010) gearbeitet haben, soll ein kurzer Blick auf die wichtigsten Neuerungen der Version 2012 nicht vorenthalten werden.

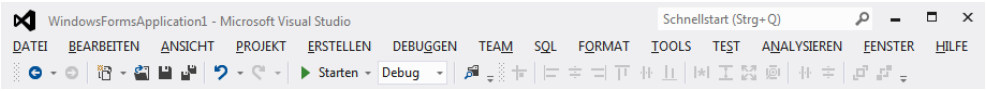
1.6.1 Die neue Visual Studio 2012-Entwicklungsumgebung

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.

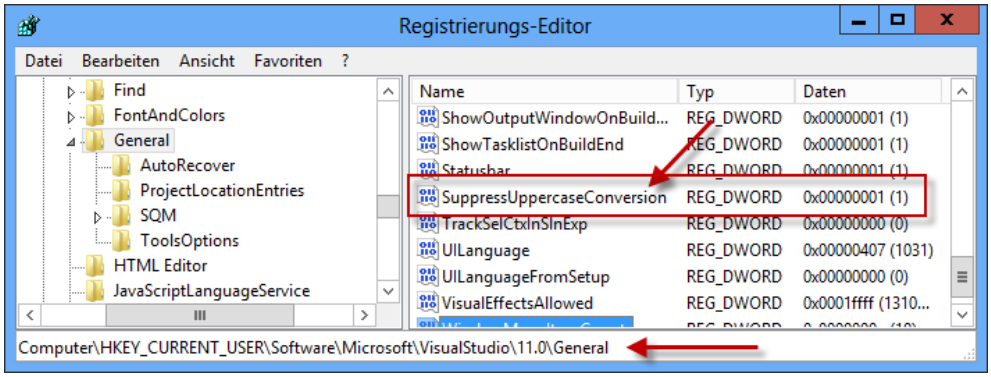
Sehr gewöhnungsbedürftig ist die komplette Großschreibung der Menü-Oberpunkte:



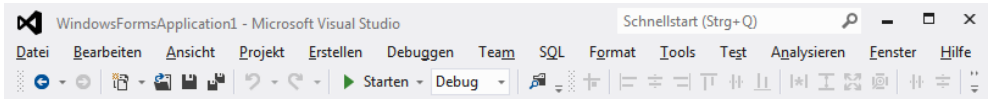
Wem dieser aufdringliche Stil nicht gefällt, für den gibt es eine Lösung: Rufen Sie die Registry auf (*regedit* im Suchfeld des Windows-Startmenüs eingeben). Unter dem Schlüssel *HKEY_CUR-*

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

RENT_USER\Software\Microsoft\VisualStudio\11.0\General erzeugen Sie einen neuen DWORD-Eintrag mit dem Namen *SuppressUppercaseConversion* und weisen diesem den Wert 1 zu.

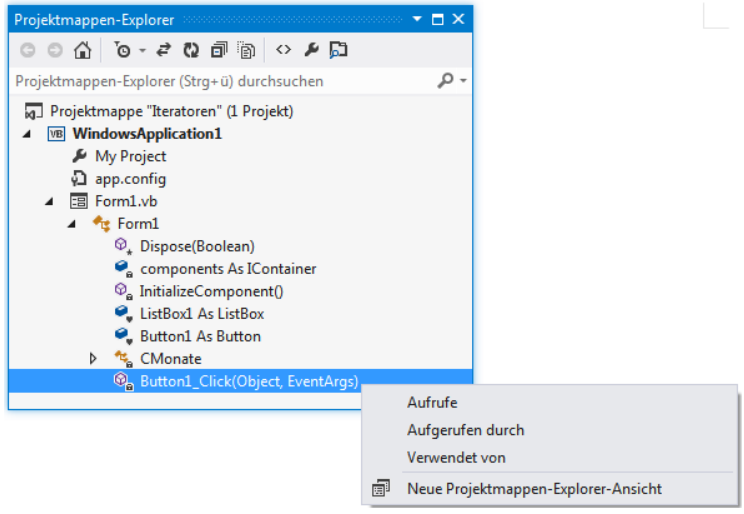


Nach dem Neustart von Visual Studio werden Sie ab jetzt von einem viel freundlicheren Menü begrüßt:



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 *.vb*-Datei expandieren, um die Klassen innerhalb der Datei zu betrachten, dann die Klasse weiter expandieren, um ihre Mitglieder und deren Aufrufhierarchien zu untersuchen:

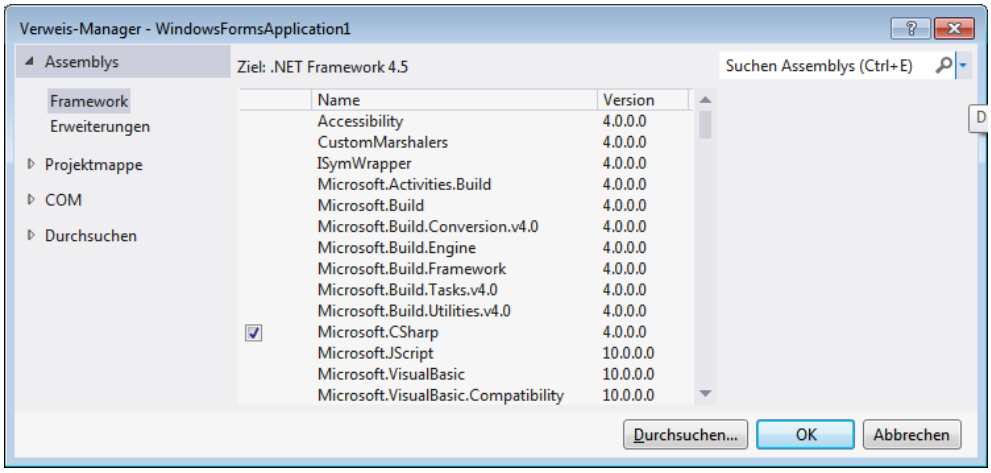


Registerkartenverwaltung der geöffneten Dateien

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.

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.



ByVal kann weggelassen werden

Der alte VB.NET Code-Editor hat immer automatisch *ByVal* hinzugefügt, wenn Sie Parameter definiert haben oder wenn Eventhandler generiert wurden, z.B.:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
                                                        Handles Button1.Click
```

Damit ist jetzt Schluss[1] und Sie können sich auf deutlich kürzere und damit übersichtlichere Codezeilen freuen, z.B. sieht obige Zeile jetzt so aus:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

Nach wie vor hat natürlich auch die alte Schreibweise ihre Gültigkeit.

[1] Diese überfällige Änderung wurde bereits mit VS 2010 SP1 eingeführt.

Suchen überall

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

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 Visual Basic, C#, 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:

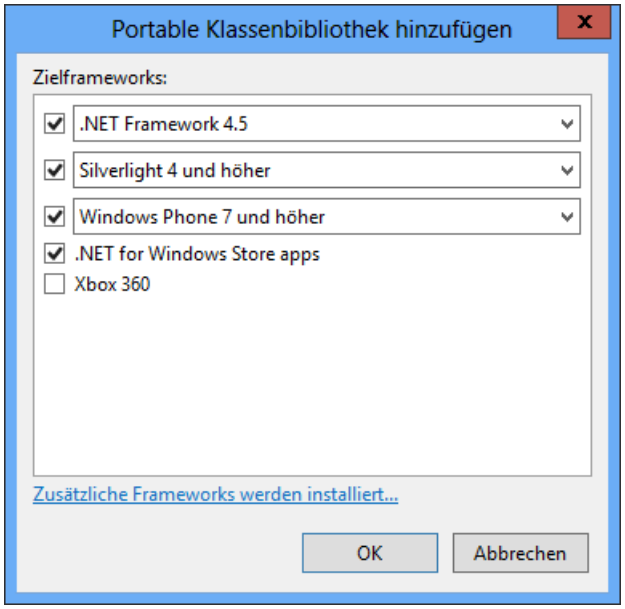
WinRT-Anwendungen

Unter Windows 8 kann eine Teilmenge vom .NET Framework 4.5 zum Erstellen von Windows-Apps im WinRT-Stil verwendet werden (siehe Kapitel 18 bis 22).

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 *Ribbon*-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 VB 2012 – Sprache und Compiler

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

IDE	Wichtigste Neuerungen in Visual Basic					
VS 2002	Managed Code					
VS 2005	Generics	Nullable Types		Operatoren-Überladung	Partielle Klassen	
VS 2008	Lambda Ausdrücke	Erweiterungs-methoden	Objekt-initialisierer	Anonyme Typen	LINQ	Typinferenz
VS 2010	Late Binding (dynamisch)	Mehrzeilige Lambda-Ausdrücke		Collection Initialisierer	Parallele Programmierung (TPL) PLINQ	
VS 2012	Asynchrone Features (Async, Await)	Caller Information		Iteratoren (Yield)		

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 Kapitel 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 Kapitel 11).

Iteratoren

Was unter C# schon lange möglich war, geht jetzt auch mit Visual Basic: Sie können das *Yield*-Statement verwenden, um mittels *For Each*-Schleife durch selbst definierte Collections zu iterieren (siehe Abschnitt 5.3.5).

1.7 Praxisbeispiele

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 alles 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-Anwendung für Einsteiger

Die bescheidene Funktionalität beschränkt sich auf ein Fensterchen mit einer Schaltfläche, über welche per Mausklick die Beschriftung der Titelleiste in "Hallo VB-Freunde" geändert werden kann. Das Beispiel demonstriert, mit welch geringem Aufwand man in Visual Studio eigene 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 schneller über eine vorher eingerichtete 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 dem Knoten *Vorlagen/Andere Sprachen/Visual Basic* zunächst *Windows* aus (siehe Abschnitt 1.4.1).

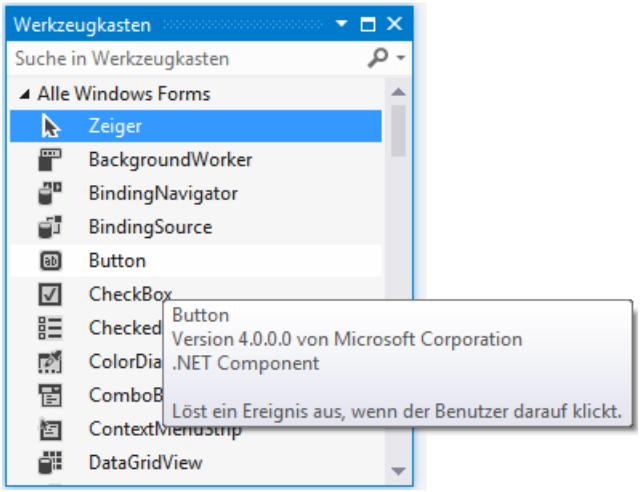
Im Mittelteil klicken Sie auf *Windows Forms-Anwendung* (ganz oben in der Liste). Nehmen Sie im unteren Teil die folgenden Einträge vor bzw. belassen es bei den Standardvorgaben:

Name:	*WindowsApplication1*
Ort:	z.B.: *C:\VB\Beispiele*
Projektmappenname:	*WindowsApplication1*

HINWEIS: In der Klappbox oben links 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 die gewünschte Komponente aus.

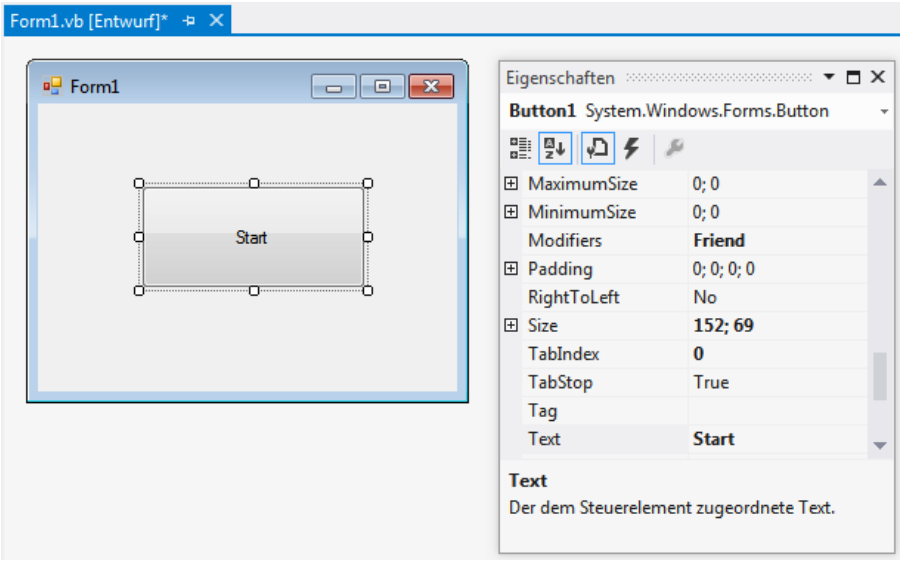


Ein schneller Doppelklick befördert das Steuerelement direkt auf das Formular. Sie können aber auch den gewünschten *Button* erst mit einem einfachen Mausklick im Werkzeugkasten aktivieren, um ihn anschließend auf dem Formular abzusetzen und auf die gewünschte Größe zu zoomen.

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 Objekt 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".

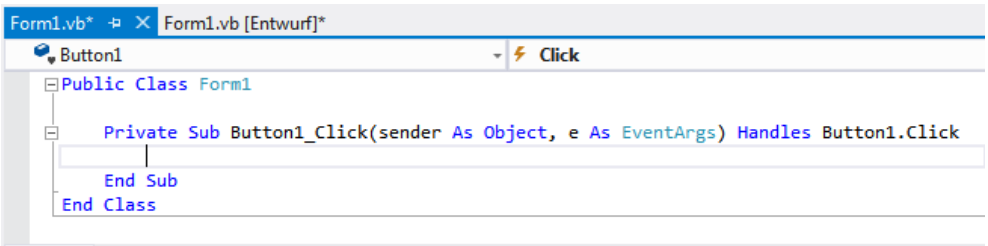
HINWEIS: Verwechseln Sie die *Text*-Eigenschaft nicht mit der *Name*-Eigenschaft. Wenn Sie ein neues Steuerelement platzieren, setzt Visual Studio standardmäßig den Wert von *Text* zunächst auf den von *Name*.



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 Ihr Augenmerk auf die Schreibmarke, welche im vorgefertigten Rahmencode für die *Click*-Ereignisbehandlungsroutine (Event-Handler) blinkt. Hier tragen Sie Ihren VB-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.

Fügen Sie dazu die fett hervorgehobene Anweisung in den Rahmencode ein:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Me.Text = "Hallo VB - Freunde!"
End Sub
```

4. Etappe: Programm kompilieren und testen

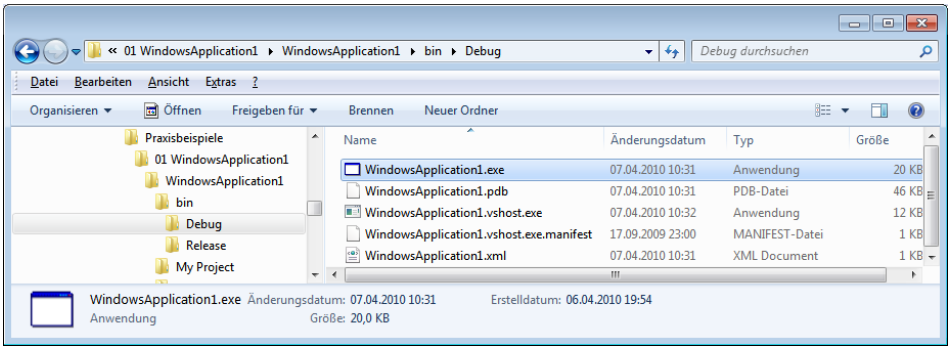
Kompilieren Sie das Programm durch Klicken auf das kleine grüne Dreieck in der Symbolleiste (bzw. Menü *Debuggen|Debugging starten* oder *F5*). Sie befinden sich jetzt im Ausführungsmodus. Ihr Programm "lebt", denn die Schaltfläche lässt sich klicken, und die Beschriftung der Titelleiste ändert sich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

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

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

Bemerkungen

- Bei diesem Prográmmchen haben Sie ganz nebenbei auch gelernt, dass man 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 (*Me*) durch einen Punkt getrennt.
- Die Properties, die Sie im Eigenschaften-Fenster zuweisen, bezeichnet man auch als *Starteigenschaften*. Zur Laufzeit können diese – wie im Beispiel für die *Text*-Eigenschaft des Formulars gezeigt – durchaus ihren Wert ändern.
- Die als Ergebnis des Kompilierprozesses generierte *.exe*-Datei finden Sie, ziemlich versteckt, im Unterverzeichnis *...\WindowsApplikation1\bin\Debug* des Projektordners. Es handelt sich hierbei allerdings **nicht** um eine klassische Exe-Datei, sondern um eine so genannte *Assemblierung* (siehe Abschnitt 1.5.2). Da diese Exe im MSIL-Code vorliegt, ist sie nur auf solchen Rechnern lauffähig, auf denen vorher die Laufzeitumgebung (Runtime) des .NET-Frameworks installiert wurde. Wenn Sie die Programmentwicklung abgeschlossen und Visual Studio beendet haben, so können Sie später jederzeit in dieses Verzeichnis wechseln, um durch Doppelklick auf die Datei *WindowsApplication1.exe* das fertige Programm zur Ausführung zu bringen.



- Direkt im Projektverzeichnis befindet sich die Projektmappendatei *WindowsApplication1.sln*. Wenn Sie auf diese Datei doppelklicken[1], so wird standardmäßig Visual Studio geöffnet und das komplette Programm in die Entwicklungsumgebung geladen.
- Falls nach Doppelklick auf die Projektmappendatei **.sln* zwar Visual Studio startet, die Entwicklungsumgebung aber leer bleibt, sollten Sie zunächst den Projektmappen-Explorer öffnen (Menü *Ansicht/Projektmappen-Explorer*) und dann durch Doppelklick (z.B. auf *Form1.vb*) die einzelnen Fenster in die Entwurfsansicht bringen. Zur Codeansicht wechseln Sie entweder mit F7 oder über das Kontextmenü (rechte Maustaste).

1.7.2 Windows-Anwendung für fortgeschrittene Einsteiger

Diesmal soll es keine Spielerei, sondern ein durchaus nützliches Prograämmchen sein – die Umrechnung von Euro in Dollar, also ein simpler Währungsrechner. Durch Vergleichen mit der von uns bereits in 1.2 geschriebenen ersten VB-Anwendung dürften auch die Unterschiede der klassischen Konsolentechnik zur visualisierten, objekt- und ereignisorientierten Windows-Programmierung deutlich werden.

1. Etappe: Visueller Entwurf der Bedienoberfläche

Öffnen Sie ein neues Visual Basic-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].

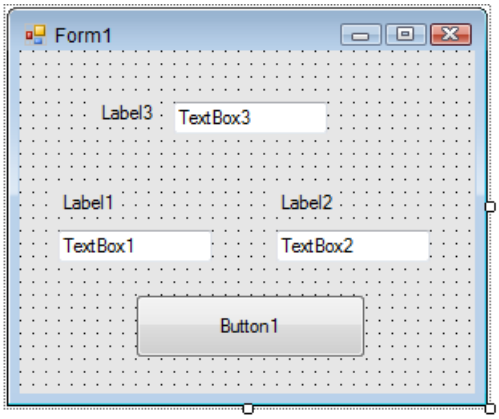
Sie brauchen außer dem bereits vorhandenen Startformular *Form1* drei *Label* zur Beschriftung, drei *TextBox*en für die Eingabe und einen *Button* zum Beenden des Programms. Für die Namensgebung sorgt Visual Studio automatisch, es sei denn, Sie möchten den Objekten eigene Namen verleihen.

---

[1] Das werden Sie z.B. häufig beim Laden von Beispielen aus der Buch- tun.

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

HINWEIS: Konzentrieren Sie sich in der ersten Etappe nur auf Lage und Abmessung der Steuerelemente, nicht auf deren Beschriftung, da die 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 *Label*-Komponente. Der Mauszeiger wechselt sein Aussehen.
- Danach bewegen Sie den Mauszeiger zu der Stelle von *Form1*, an welcher sich die linke obere Ecke von *Label1* befinden soll, drücken die Maustaste nieder und zoomen (bei gedrückt gehaltener Maustaste) das Label auf seine endgültige Größe. Analog verfahren Sie mit *Label2* und *Label3*.
- Nun klicken Sie im Werkzeugkasten auf das Symbol für die *TextBox*-Komponente und erzeugen nacheinander *TextBox1*, *TextBox2* und *TextBox3*.
- Schließlich bleibt noch *Button1*, den Sie am unteren Rand von *Form1* absetzen.

2. Etappe: Zuweisen der Objekteigenschaften

Unser Prgrämmchen 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:

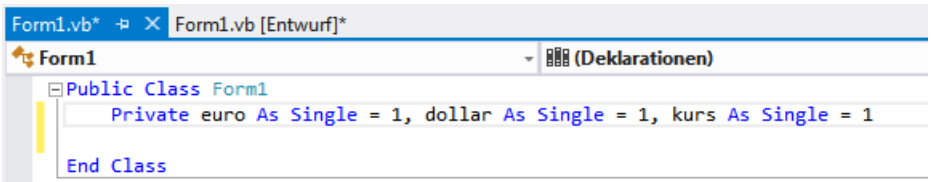
Name des Objekts	Eigenschaft	Neuer Wert
Form1	*Text* *Font.Size*	Umrechnung Euro-Dollar *10*
Label1	*Text*	Euro
Label2	*Text*	Dollar
Label3	*Text*	Kurs 1:
TextBox1	*TextAlign*	*Right*
TextBox2	*TextAlign*	*Right*
TextBox3	*TextAlign*	*Center*
Button1	*Text*	Beenden

3. Etappe: Verknüpfen der Objekte mit Ereignissen

Während Sie die beiden Vorgängeretappen noch getrost Ihrer Sekretärin überlassen konnten, beginnt jetzt Ihre Hauptarbeit als VB-Programmierer. Wechseln Sie zum Code-Fenster *Form1.vb* (auch mit *F7*, *Ansicht|Code* oder dem Kontextmenü des Formulars möglich). Was Sie hier erwartet, ist die von Visual Studio vorbereitete Klassendeklaration von *Form1*.

Zunächst fügen Sie eine Anweisung ein, mit der drei Gleitkommavariablen des *Single*-Datentyps deklariert werden. Gleichzeitig werden diese Variablen mit dem Wert 1 initialisiert:

```
Private euro As Single = 1, dollar As Single = 1, kurs As Single = 1
```

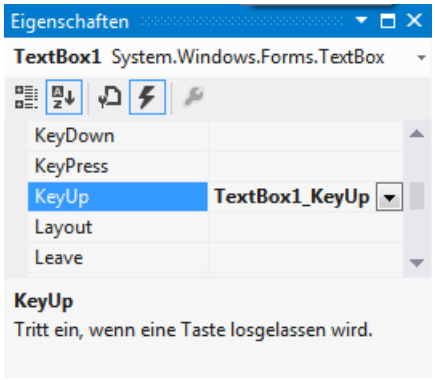


Im Unterschied zur Konsolenanwendung, bei welcher uns das Programm die Einhaltung einer bestimmten Eingabereihenfolge aufzwingt, soll in unserer Windows-Anwendung die Berechnung immer dann neu gestartet werden, wenn bei der Eingabe in eine der drei Textboxen eine Taste losgelassen wurde. Wir müssen also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

Dabei ist eine fast schon rituelle Erstellungsreihenfolge zu beachten, wie Sie 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 Event-Handlers generiert und im Code-Fenster angezeigt.
- **Ereignisbehandlungen programmieren**
  Füllen Sie den Event-Handler mit den gewünschten VB-Anweisungen aus.

Wir beginnen in unserem Beispiel mit dem *KeyUp*-Event-Handler für *TextBox1*, der immer dann ausgeführt wird, wenn der Benutzer den Euro-Betrag ändert:



Füllen Sie nun den Event-Handler wie folgt aus:

```
Private Sub TextBox1_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox1.KeyUp
        euro = Convert.ToSingle(TextBox1.Text)
        dollar = euro * kurs
        TextBox2.Text = dollar.ToString("#,##0.00")
End Sub
```

HINWEIS: Sie müssen nur die hier fett gedruckten Anweisungen selbst einfügen, der übrige Rahmencode wird automatisch erzeugt, wenn Sie die oben erläuterte Erstellungsreihenfolge beachten!

Der Prozedurkopf des Event-Handlers verweist standardmäßig vor dem Unterstrich (_) auf den Namen des Objekts (*TextBox1*) und danach auf das entsprechende Ereignis (*KeyUp*). Das vorangestellte *Private* verdeutlicht, dass auf die Prozedur nur innerhalb des *Form1*-Klasse zugegriffen werden kann.

Auf analoge Weise erzeugen Sie die Event-Handler für die Steuerelemente *TextBox2* und *TextBox3*.

Ändern des Dollar-Betrags:

```
Private Sub TextBox2_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox2.KeyUp
        dollar = Convert.ToSingle(TextBox2.Text)
        euro = dollar / kurs
        TextBox1.Text = euro.ToString("#,##0.00")
End Sub
```

Ändern des Umrechnungskurses:

```
Private Sub TextBox3_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox3.KeyUp
        kurs = Convert.ToSingle(TextBox3.Text)
        dollar = euro * kurs
        TextBox2.Text = dollar.ToString("#,##0.00")
End Sub
```

HINWEIS: Denken Sie jetzt noch nicht über den tieferen Sinn der einzelnen Anweisungen nach, denn dazu sind die späteren Kapitel da.

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

```
Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        TextBox1.Text = euro.ToString
        TextBox2.Text = dollar.ToString
        TextBox3.Text = kurs.ToString
End Sub
```

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 Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Me.Close()
End Sub
```

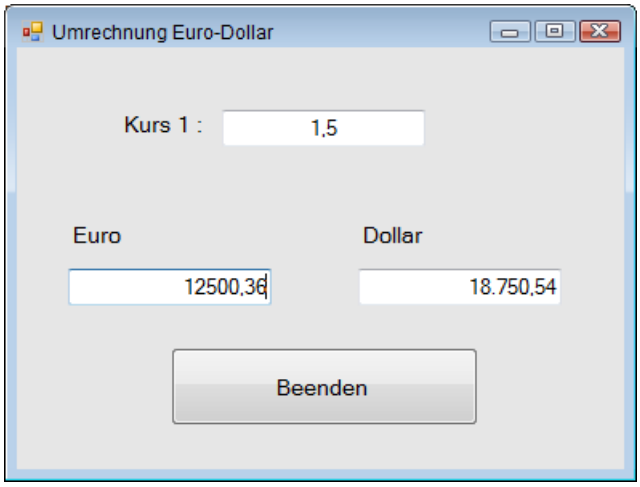
4. Etappe: Programm kompilieren und testen

Klicken Sie auf den 

-Button in der Symbolleiste (oder *F5*-Taste), und im Handumdrehen ist Ihre Windows Forms-Anwendung kompiliert und ausgeführt!

Spielen Sie ruhig ein wenig mit verschiedenen Werten herum, um sich den Unterschied zwischen Konsolen- und Windows-Programmen so richtig 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 nicht erforderlich.

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

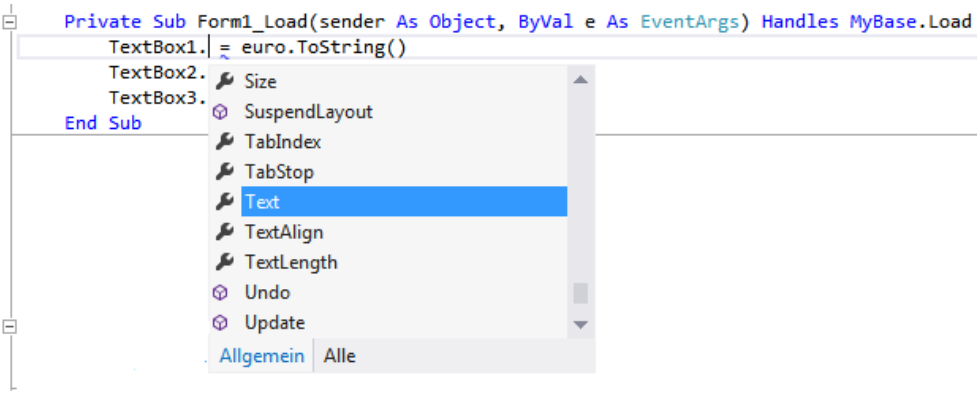


Da wir aus Gründen der Einfachheit in diesem Programm auf eine Überprüfung der Eingabewerte (Eingabevalidierung) verzichtet haben, sollten Sie auf das komplette Löschen des Inhalts eines Textfelds verzichten und nur gültige Zahlenwerte eingeben (also z.B. auch keine Buchstaben). Andernfalls ist ein Konvertieren der Eingabe in eine Zahl nicht möglich und das Programm stürzt mit einer umfangreichen Fehlermeldung ab.

Um wieder in den Entwurfsmodus zurückzukehren, wählen Sie nach solch einem Fehler das Menü *Debuggen/Debugging beenden*.

IntelliSense – die hilfreiche Fee

Eines der bemerkenswertesten Features des Code-Editors von Visual Studio ist die 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 vom lästigen Nachschlagen in der Hilfe und bewahrt Sie vor Schreibfehlern.

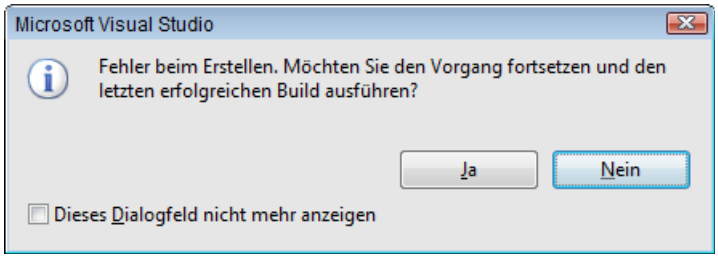


HINWEIS: Wenn Sie das markierte Element übernehmen wollen, brauchen Sie den Namen nicht zu Ende zu schreiben, da die IntelliSense den kompletten Text automatisch ergänzt.

Fehlersuche

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

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



Normalerweise 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 mit einer Wellenlinie unterstrichen sind. Auch hier erhalten Sie Hinweise zur Fehlerursache, wenn Sie mit der Maus auf die betreffende Stelle zeigen.

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 (Menü *Ansicht|Fehlerliste*).

Weitere hilfreiche Hinweise zum Debuggen finden Sie im Kapitel 11!

Kapitel 2

Einführung in Visual Basic

In diesem Kapitel wollen wir Ihnen den für den Einstieg wichtigen Sprachkern von Visual Basic erklären. 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 Sprachfeatures erst in späteren Kapiteln (objektorientiertes Programmieren in Kapitel 3, fortgeschrittene Sprachelemente in den Kapiteln 4 und 5).

HINWEIS: Testen Sie möglichst viele der kleinen Codeschnipsel selbst am eigenen PC. Als Codegerüst könnte entweder eine Konsolenanwendung (siehe dazu das Einführungskapitel, Abschnitt 1.2) oder aber eine Windows Forms-Anwendung (siehe 1.7.1/ 1.7.2) dienen.

2.1 Grundbegriffe

Zur Vorbereitung empfehlen wir ein Rückblättern zu den Praxisbeispielen des Kapitels 1, wo einige grundlegende Begriffe (Anweisungen, Klassen, Namensraum, Gültigkeitsbereiche) bereits grob erklärt wurden.

2.1.1 Anweisungen

Wir verstehen unter einer Anweisung (Statement) einen Befehl (Instruction), der sich aus bestimmten Sprachelementen, wie Schlüsselwörtern, Operatoren, Variablen, Konstanten, Ausdrücken, zusammensetzt.

Wir unterscheiden zwei Arten von Anweisungen:

- Deklarations-Anweisungen,
  diese spezifizieren eine Variable, Konstante, Prozedur oder einen Datentyp.
- Ausführbare Anweisungen,
  diese führen bestimmte Aktionen aus (Aufruf einer Prozedur, Zuweisen eines Werts, Durchlaufen einer Schleife etc.).

Wie in jeder anderen Programmiersprache auch, müssen Visual Basic-Anweisungen bestimmten Regeln entsprechen, die man in ihrer Gesamtheit als *Syntax*[1] bezeichnet.

Durch gute Strukturierung Ihres Quellcodes, wie z.B. blockweises Einrücken, machen Sie Ihre Programme übersichtlicher und verringern somit die Fehlerquote. Normalerweise brauchen Sie sich aber darum nicht selbst zu kümmern, denn das erledigt die IDE automatisch für Sie. Anderenfalls gilt:

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 (Variablen, Methoden, Klassen, ...) verwendet man so genannte Bezeichner.

Namensgebung und Schlüsselwörter

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 und ohne eingeschlossene Punkte und Typkennzeichen auskommen.
- Ein Bezeichner muss innerhalb seines Gültigkeitsbereichs eindeutig sein (also keine Mehrfachvergabe des gleichen Namens).
- Bezeichner müssen kürzer als 256 Zeichen sein und
- dürfen keine *Schlüsselwörter* sein.

Bei *Schlüsselwörtern* handelt es sich um vordefinierte reservierte Bezeichner, die den Kern von Visual Basic ausmachen, z.B. *Class, Dim*, *Sub, If, End* etc.

HINWEIS: Schlüsselwörter sind im Codefenster von Visual Studio standardmäßig blau eingefärbt.

BEISPIEL 2.1: Zulässige und unzulässige Namensgebung

VB
```
euro
_radius
Zwerg7
```

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

BEISPIEL 2.1: Zulässige und unzulässige Namensgebung

```
%Anteil                    ' unzulässig, weil "%"-Zeichen am Anfang
7Zwerge                    ' unzulässig, weil Ziffer am Anfang
dim                        ' unzulässig, weil Schlüsselwort
```

Leerzeichen etc.

Leerzeichen zwischen den Befehlswörtern sind im Allgemeinen bedeutungslos und werden in der Regel von Visual Studio automatisch korrigiert. Außerdem gibt es eine ganze Reihe weiterer Zeichen, die z.B. als Operatoren fungieren.

BEISPIEL 2.2: Berechnung eines Ausdrucks mit Divisions- und Additions-Operator

VB
```
z = x/y + 10
```

Groß-/Kleinschreibung

Da die Groß- und Kleinschreibung in Visual Basic egal ist, dürfen Sie keine Bezeichner verwenden, die sich lediglich durch die Groß- und Kleinschreibung voneinander unterscheiden. Sie sollten die Groß-/Kleinschreibung aber dafür einsetzen, um die Lesbarkeit Ihres Programms zu verbessern.

BEISPIEL 2.3: Beide Bezeichner dürfen Sie nicht nebeneinander verwenden. Der erste Bezeichner ist besser lesbar als der zweite und sollte verwendet werden.

VB
```
MeineAdresse       ' gut lesbar
meineadresse       ' doppelter Bezeichner, schlecht lesbar
```

2.1.3 Kommentare

Kommentaranweisungen (im Editor im Allgemeinen grün hervorgehoben) dienen als zusätzliche Erläuterungen für den Programmierer und werden vom Compiler überlesen. Für das Kennzeichnen von Kommentaren verwenden Sie den einfachen Apostroph (')[1].

HINWEIS: Im Unterschied zu anderen Sprachen kennt Visual Basic keine mehrzeiligen Kommentare.

BEISPIEL 2.4: Kommentare

VB
```
x = y + 10                 ' Addiere 10 zum Wert von y
' Jede neue Kommentarzeile muss extra eingeleitet werden.
```

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".

[1] Gelegentlich findet man auch noch das veraltete REM.

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.

2.1.4 Zeilenumbruch

In früheren Versionen von Visual Basic stand normalerweise pro Zeile eine Anweisung, d.h., ein Zeilenumbruch war in der Regel gleichbedeutend mit dem Anweisungsende, es sei denn, man benutzte einen Unterstrich als Zeilentrenner. Mit der Version 2010 wurde diese "eiserne Regel" total aufgeweicht. Bevor wir aber auf die zahlreichen Ausnahmen von dieser Regel eingehen, wollen wir uns mit der klassischen Vorgehensweise beschäftigen, die nach wie vor gültig ist.

Lange Zeilen teilen

Wenn eine Anweisung sehr lang ist, kann das ihre Lesbarkeit im Quelltexteditor beeinträchtigen, und es ist außerdem unbequem, da man horizontal scrollen muss. Man kann aber eine solche Zeile durch einen Unterstrich mit vorangestelltem Leerzeichen trennen.

BEISPIEL 2.5: Die erste Zeile eines Event-Handlers wird auf zwei Zeilen aufgeteilt.

VB
```
Private Sub Button1_Click(sender As Object, e As EventArgs) _
                                                    Handles Button1.Click
```

HINWEIS: Zeichenketten müssen Sie vorher in Teilketten zerlegen, bevor Sie sie trennen!

BEISPIEL 2.6: Eine Zeichenkette wird korrekt getrennt.

VB
```
Label1.Text = "Dies ist eine sehr lange Zeichenkette, die sehr unübersichtlich wäre " &
              ", wenn man sie nicht trennen würde."
```

Implizite Zeilenfortsetzung

Hier handelt es sich um eine erst mit VB 2010 eingeführte Neuerung. In folgenden Situationen kann man eine Anweisung auch ohne Verwendung eines Unterstrichs über mehrere Zeilen verteilen:

- nach einem Komma
- nach öffnenden oder vor schließenden Klammern (einfache oder geschweifte)
- nach dem Zeichen "&"
- nach Zuweisungs- und Binäroperatoren (z.B. =, +=, +, -, Mod, Or, ...)
- nach dem Punkt (.) als Objektqualifizierer

- innerhalb von LINQ-Ausdrücken
- in einigen weiteren Spezialfällen (nach *In* in *For Each*-Schleifen, nach *Is* bzw. *IsNot* ...)
- an bestimmten Stellen im XML-Code, z.B. nach dem Öffnen eines eingebetteten Ausdrucks (<%=) oder vor dem Schließen (%>)

Obige Aufzählung ist nicht ganz vollständig, im Zweifelsfall sollte man den Unterstrich verwenden oder einfach ausprobieren, ob die mehrzeilige Anweisung von der IDE angenommen bzw. richtig verarbeitet wird.

BEISPIEL 2.7: Die Kopfzeile des Eventhandlers im obigen Beispiel kann auch ohne Unterstrich auf zwei Zeilen verteilt werden

VB
```
Private Sub Button1_Click(sender As Object,
                         e As EventArgs) Handles Button1.Click
```

BEISPIEL 2.8: Eine weitere alternative Schreibweise für das Vorgängerbeispiel

VB
```
Private Sub Button1_Click(
                             sender As Object,
                             e As EventArgs
                         ) _
                         Handles Button1.Click
```

BEISPIEL 2.9: Eine LINQ-Abfrage (siehe Kapitel 6) sieht jetzt schön und übersichtlich aus

VB
```
Dim expr = From z In zahlen
           Where z > 10
           Order By z
           Select z
```

Mehrere Anweisungen pro Zeile

Um Platz zu sparen, können aber auch mehrere Anweisungen pro Zeile geschrieben werden, diese sind dann durch einen Doppelpunkt (:) voneinander zu trennen.

BEISPIEL 2.10: Die Anweisungen

VB
```
euro = CSng(TextBox1.Text)
dollar = euro * kurs
```

kann man wie folgt in eine Zeile schreiben:

```
euro = CSng(TextBox1.Text) : dollar = euro * kurs
```

2.2 Datentypen, Variablen und Konstanten

Jedes Programm "lebt" in erster Linie von seinen Variablen und Konstanten, die bestimmten Datentypen entsprechen.

2.2.1 Fundamentale Typen

Die folgende Tabelle gibt eine Übersicht der einfachen (fundamentalen) Datentypen, die Visual Basic zur Verfügung stellt[1]:

VB-Datentyp	.NET-CLR-Typ	Erläuterung	Länge [Byte]
Short	*System.Int16*	kurze Ganzzahl zwischen -2^{15} ... $2^{15}-1$ (-32.768 ... 32.767)	2
Integer	*System.Int32*	Ganzzahl zwischen -2^{31} ... $2^{31}-1$ (-2.147.483.648 ... 2.147.483.647)	4
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
Single	*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
Date	*System.DateTime*	Datum/Zeit zwischen 1. Januar 1 bis 31. Dezember 9999	8
Char	*System.Char*	ein beliebiges Unicode-Zeichen	2
String	*System.String*	beliebige Unicode-Zeichenfolge mit einer maximalen Länge von ca. 2 000 000 000 Zeichen	2 pro Zeichen plus 10
Boolean	*System.Boolean*	Wahrheitswert (*True*, *False*)	2
Byte	*System.Byte*	positive Ganzzahl zwischen 0 ... 255	1
Object	*System.Object*	universeller Datentyp	4

Wie Sie obiger Tabelle entnehmen, entsprechen alle Visual Basic-Datentypen einer Klassendefinition im .NET-Framework.

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

Die CLR-Datentypen sind im *System*-Namensraum angeordnet. Bei der Deklaration (siehe Abschnitt 2.2.4) ist es egal, welchen der beiden möglichen Typbezeichner Sie angeben.

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 VB.

Wertetypen

Dazu zählen all die einfachen Datentypen wie *Byte*, *Integer*, *Double* ..., hinzu kommen später noch andere wie beispielsweise *Structure* (siehe 2.7.2 und *DateTime* (siehe Seite 278). 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

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 möglichst mit einem Kleinbuchstaben.
- Falls Bezeichner aus mehreren Wörtern zusammengesetzt sind, so sollten ab dem zweiten Wort alle Wörter mit einem Großbuchstaben beginnen ("Kamelschreibweise").

BEISPIEL 2.11: Ein Variablenname, der sich aus mehreren Wörtern zusammensetzt.

VB
```
meinHaushaltskassenSaldo
```

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

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

HINWEIS: Die Konventionen der Namensgebung von Variablen gelten auch für Konstanten, Funktionen/Prozeduren und andere benutzerdefinierte Sprachelemente.

2.2.4 Deklaration von Variablen

Variablen sind benannte Speicherplatzstellen, der Variablenname dient dazu, die Speicheradresse im Programmcode anzusprechen.

Dim-Anweisung

Sie können Ihre Variablen mit dem Schlüsselwort *Dim* deklarieren und ihnen mit *As* einen Datentyp zuweisen[1].

SYNTAX: `Dim` *`Varaiablenname`* `As` *`Datentyp`*

BEISPIEL 2.12: Verschiedene Variablendeklarationen

VB
```
Dim a As Single
Dim anzahl As Integer
Dim breite As Double
```

In einer Anweisung lassen sich auch mehrere Variablen hintereinander deklarieren.

BEISPIEL 2.13: Die gleichen Deklarationen wie im Vorgängerbeispiel

VB
```
Dim a As Single, anzahl As Integer, breite As Double
```

Laut obiger Tabelle kann man als Datentyp auch den CLR-Typ angeben.

BEISPIEL 2.14: Die folgenden Deklarationen sind gleichwertig.

VB
```
Dim i As Integer
Dim i As System.Int32
Dim i As Int32
```

Mehrfachdeklaration

Wenn Sie mehrere Variablen vom gleichen Datentyp hintereinander deklarieren wollen, genügt es, den Datentyp nur einmal anzugeben.

BEISPIEL 2.15: Mehrfachdeklaration

VB
Statt

```
Dim i As Integer
Dim j As Integer
```

[1] Später werden Sie erfahren, dass man außer mit *Dim* auch noch mit *Private*, *Public* oder *Static* deklarieren kann.

BEISPIEL 2.15: Mehrfachdeklaration

dürfen Sie auch schreiben

```
Dim i, j As Integer
```

Anfangswerte

Zusammen mit der Deklaration können Sie die Variablen auch gleich initialisieren.

BEISPIEL 2.16: Anfangswerte

VB

Statt

```
Dim max As Single
max = 99.99
```

können Sie kürzer formulieren:

```
Dim max As Single = 99.99
```

Option Explicit

Dieser Schalter, der (für Sie zunächst unsichtbar) am Beginn eines Moduls steht

SYNTAX: `Option Explicit On|Off`

erzwingt die Deklaration einer Variablen, bevor sie verwendet wird (*On*), oder ermöglicht die sofortige Verwendung ohne vorherige Deklaration (*Off*).

BEISPIEL 2.17: Sie haben vergessen, die Variable *j* zu deklarieren

VB

```
Option Explicit On
Dim i As Integer
j = 10        ' undeklarierte Variable erzeugt Compiler-Fehler !
i = 10
```

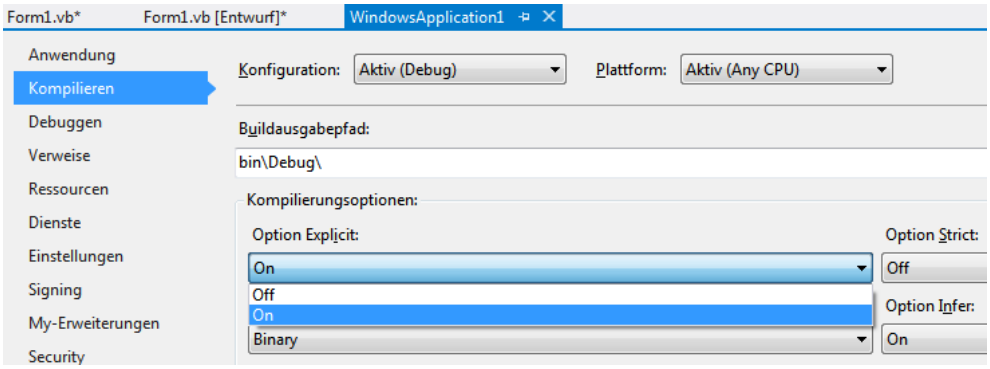
... aber so bemerken Sie Ihren Fehler nicht:

```
Option Explicit Off
Dim i As Integer
j = 10        ' neue Variable vom Object-Datentyp wird erzeugt !
i = 10
```

HINWEIS: Sie sollten grundsätzlich mit *Option Explicit On* arbeiten, da dies die Typsicherheit erhöht und die Fehlerquote senkt.

Glücklicherweise ist *Option Explicit On* die Standardeinstellung. Wenn Sie im Projektmappen-Explorer auf den Projektnamen rechtsklicken und *Eigenschaften* wählen, können Sie im Projekt-

eigenschaftenfenster unter "Kompilieren" die Kompilierungsoption *Option Explicit* für die gesamte Anwendung neu festlegen.



Option Strict

Ähnlich wie *Option Explicit* steht dieser Schalter am Beginn eines Moduls oder kann ebenfalls im Projekteigenschaftenfenster eingestellt werden (siehe vorhergehender Abschnitt).

SYNTAX: `Option Strict On|Off`

Ist die Schalterstellung *Off* (unter Visual Studio 2012 ist das der Standard), so gibt es einige (fragwürdige) Erleichterungen beim Zuweisen unterschiedlicher Datentypen. So kann z.B. bei der Variablendeklaration auf die Angabe des Datentyps verzichtet werden, die Variable ist dann automatisch vom Typ *Object*.

BEISPIEL 2.18: Der folgende Code ergibt einen Fehler, da die *As*-Klausel fehlt:

VB
```
Option Strict On
...
Dim x
```

Dieser Code aber lässt sich fehlerfrei kompilieren:

```
Option Strict Off

...
Dim x         ' x ist vom Object-Datentyp
```

HINWEIS: Der Lernende ist gut beraten, wenn er zunächst mit *Option Strict On* arbeitet, da dies das Verständnis der Programmiersprache fördert und das Portieren des Codes nach anderen .NET-Sprachen (z.B. C#) deutlich erleichtert!

2.2.5 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.

Die Deklaration erfolgt mit dem Schlüsselwort *Dim*, wobei man auf die *As*-Klausel verzichtet.

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

Manch ein VB-Entwickler wird denken, dass es sich hierbei um dasselbe Verhalten wie bei *Option Strict Off* handelt, tatsächlich aber erhalten Sie eine streng typisierte Variable.

BEISPIEL 2.19: Die Initialisierung der Variablen *a* wird vom Compiler ausgewertet und der Typ aufgrund des Wertes 35 auf *Integer* festgelegt.

VB
```
Dim a = 35
```
Obige Zeile ist semantisch identisch mit folgendem Ausdruck:
```
Dim a As Integer = 35
```

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

BEISPIEL 2.20: Da die Variable *b* vom Compiler als *Integer* festgelegt wurde, kann ihr später kein *Double*-Wert zugewiesen werden.

VB
```
Dim b = 7
b = 12.3            ' Fehler!
```

HINWEIS: Typinferenz funktioniert nur bei lokalen Variablen!

2.2.6 Konstanten deklarieren

Im Unterschied zu Variablen bleibt der Wert einer Konstanten während der gesamten Laufzeit eines Programms unverändert. Sie legen ihn einmalig mit der *Const*-Anweisung fest. Es ist allgemein üblich, den Namen einer Konstanten in Großbuchstaben zu schreiben.

BEISPIEL 2.21: Konstanten

VB
```
Const PI As Single = 3.1415
Const MELDUNG As String = "Achtung"
Const ANZAHL As Integer = 5
Const X1 As Double = 3 / 4, x2 As Double = 2 + 1 / 3
```

Sammlungen von Konstanten werden üblicherweise in so genannten *Enumerationen* "zusammengehalten" (siehe 2.7.1).

2.2.7 Gültigkeitsbereiche von Deklarationen

Bis jetzt hatten wir Variablen ausschließlich mit dem Zugriffsmodifizierer *Dim* deklariert. Es gibt allerdings noch mindestens drei weitere Alternativen (*Static*, *Private*, *Public*), die sich bezüglich *Sichtbarkeit* und *Lebensdauer* unterscheiden. Was bedeutet das?

- Sichtbarkeit
  ... bestimmt, von welchen Stellen des Programms auf die Variable zugegriffen werden darf.
- Lebensdauer
  ... beschreibt, wie lange die Variable den für sie reservierten Speicherplatz beansprucht.

Die folgende Tabelle vermittelt dazu zunächst einen allgemeinen Überblick:

Zugriffs-modifizierer	Deklaration in	Sichtbarkeit	Lebensdauer
Dim	Sub/Funktion	lokal	bis Prozedur verlassen wird
Static	Sub/Funktion	lokal statisch	Programmlaufzeit
Private	Modul	modulglobal	Programmlaufzeit
Public	Modul	programmglobal	Programmlaufzeit

Nun wollen wir einen genaueren Blick auf die einzelnen Deklarationsmöglichkeiten werfen.

2.2.8 Lokale Variablen mit Dim

Eine lokale Variable wird in der Regel mit *Dim* deklariert und ist nur innerhalb der Prozedur[1] (genauer genommen innerhalb des Blocks) bekannt, in welcher sie deklariert wurde. Das bedeutet, dass sich ihre Sichtbarkeit auf diese Prozedur beschränkt und von außerhalb der Lese- und Schreibzugriff verwehrt ist. Bei jedem neuen Aufruf der Prozedur wird die Variable mit ihrem Leerwert (0 oder Leerstring"") neu initialisiert. Beim Verlassen der Prozedur werden alle mit *Dim* deklarierten Variablen zerstört, und der von ihnen belegte Speicherplatz wird wieder freigegeben.

BEISPIEL 2.22: Platzieren Sie auf einem Formular einen Button und hinterlegen Sie dessen *Click*-Ereignis wie folgt.

VB
```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
Dim anzahl As Integer       ' lokale Variable
  anzahl = anzahl + 1
  MessageBox.Show(anzahl.ToString)
End Sub
```

[1] *Prozedur* ist der Überbegriff für *Subroutinen* (Sub) und *Funktionen* (Function), siehe 2.8.

Bei jedem Klick auf den Button zeigt das Meldungsfenster "1" an, obwohl wir eigentlich "hochzählen" wollten. Eine Lösung bietet das Deklarieren mit *Static* (siehe nächster Abschnitt).

Die Gültigkeit lokaler Variablen innerhalb einer Prozedur kann weiter eingegrenzt werden, wenn Sie diese innerhalb eines Anweisungsblocks deklarieren. Allerdings darf dann im übergeordneten Block eine gleichnamige Variable nicht nochmals auftreten.

BEISPIEL 2.23: Die Variable *s* gilt nur innerhalb des *If-Then*-Blocks.

VB
```
Dim i As Integer = 5
If i < 5 Then Dim s As String = "Ich bin in einem Block deklariert!"
MessageBox.Show(s)         ' Fehler: Variable s unbekannt
```

2.2.9 Lokale Variablen mit Static

Eine Deklaration mit *Static* anstatt *Dim* bedeutet, dass der Wert einer Variablen nicht "verlorengeht", wenn die Prozedur, in welcher die Variable deklariert wurde, zwischenzeitlich verlassen und später wieder aufgerufen wird.

BEISPIEL 2.24: Bei jedem Klick auf den *Button* erhöht sich der angezeigte Wert um eins.

VB
```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
  Static anzahl As Integer     ' lokale Variable
  anzahl = anzahl + 1
  MessageBox.Show(anzahl.ToString)
End Sub
```

HINWEIS: *Static*-Deklarationen können **nicht** für globale Variablen angewendet werden!

2.2.10 Private globale Variablen

Mit *Private* (oder ausnahmsweise auch *Dim*) deklarieren Sie globale Variablen, deren Sichtbarkeit sich auf das Modul beschränkt. Eine solche Variable kann von jeder Prozedur innerhalb des Moduls aufgerufen und modifiziert werden. Außerhalb des Moduls ist die Variable unbekannt. Logischerweise deklarieren Sie eine private globale Variable außerhalb jeglicher Prozedur, d.h., im Allgemeinteil eines Moduls.

HINWEIS: *Dim* auf Modulebene bedeutet so viel wie *Private*, sollte aber im Interesse eines sauberen Programmierstils nicht verwendet werden.

BEISPIEL 2.25: Im Einführungsbeispiel 1.7.2 hatten wir die drei privaten globalen Variablen *euro, dollar* und *kurs* deklariert und mit Anfangswerten initialisiert:

VB
```
Private euro As Single =1, dollar As Single = 1, kurs As Single =1
```

2.2.11 Public Variablen

Wenn Sie möchten, dass auf Modulebene deklarierte Variablen auch noch allen anderen Modulen Ihrer Anwendung zur Verfügung stehen sollen, dann müssen Sie diese mit *Public* deklarieren.

BEISPIEL 2.26: Der Name einer Datenbankdatei, die Sie in Ihrem Projekt von mehreren Formularen aus öffnen müssen, wird in einer anwendungsglobalen Variablen gespeichert.

VB
```
Public dbName As String = "C:\Beispiele\Datenbankanwendung1\Firma.mdb"
```

2.3 Wichtige Datentypen im Überblick

In diesem Abschnitt wollen wir die fundamentalen Datentypen (siehe Tabelle Seite 106) etwas genauer unter die Lupe nehmen und auf einige Besonderheiten aufmerksam machen.

2.3.1 Byte, Short, Integer, Long

Die Datentypen *Byte* (8 Bit), *Short* (16 Bit), *Integer* (32 Bit) und *Long* (64 Bit) dienen dem Speichern von ganzzahligen Werten. Bei der Auswahl ist darauf zu achten, ob der Datentyp für Ihre Zwecke groß genug ist und ob evtl. ein Vorzeichen benötigt (welches *Byte* nicht bietet) wird.

Wenn Sie die Werte als Literale – d.h. direkt im Quellcode – zuweisen, müssen Sie natürlich auch auf die Einhaltung der Wertebereiche achten.

BEISPIEL 2.27: Der folgende Code führt zu einer Fehlermeldung: "Der Konstantenausdruck ist im Typ "Short" nicht repräsentierbar."

VB
```
Dim i As Short
i = 100000           ' Fehler
```

HINWEIS: Bei eingeschalteter strenger Typprüfung müssen die zugewiesenen Literale dem Datentyp der Variablen entsprechen.

BEISPIEL 2.28: Der folgende Code führt zur Fehlermeldung "Option Strict On lässt keine impliziten Typkonvertierungen von Double in Short zu."

VB
```
Dim i As Short
i = 10.5              ' Fehler bei Option Strict On oder 10  bei Option Strict Off
```

Wenn Sie im Beispiel aber *Option Strict Off* einstellen, so wird die letzte Anweisung ohne Murren akzeptiert, und *i* erhält den gerundeten Wert (10).

Für ein ganzzahliges Literal können auch die hexadezimale (Präfix **&H**) oder oktale Schreibweise (Präfix **&O**) Verwendung finden.

BEISPIEL 2.29: Die drei folgenden Zuweisungen sind identisch.

VB
```
Dim b As Byte
b = 255
b = &HFF
b = &O377
```

2.3.2 Single, Double und Decimal

Der *Single*-Typ genügt nur recht bescheidenen Ansprüchen, denn bei einer Vorkommastelle kann er nur etwa sieben Nachkommastellen speichern, während es bei *Double* 15 sind. Der hochgenaue *Decimal*-Datentyp hat 29 Stellen, die sich auf Vor- und Nachkomma aufteilen.

HINWEIS: Wenn Sie Gleitkommazahlen im Quelltext zuweisen, dürfen Sie nicht das Komma, sondern müssen den Punkt als Dezimaltrenner verwenden.

BEISPIEL 2.30: Deklarieren und Zuweisen einer Gleitkommavariablen

VB
```
Dim a As Single
a = 0.45
```

Andererseits dürfen Sie sich nicht wundern, wenn z.B. bei Zahleneingaben in ein Textfeld nur das Komma als Dezimaltrennzeichen akzeptiert wird. Dies hängt mit der deutschen Ländereinstellung zusammen (Windows-Systemsteuerung).

2.3.3 Char und String

Beide Datentypen basieren auf dem Unicode-Zeichensatz, der pro Zeichen 2 Byte beansprucht (im Unterschied zu dem 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.

Den Typ *Char* verwenden Sie zum Speichern eines einzelnen Zeichens, während Zeichenketten im *String*-Datentyp gespeichert werden. Haben Sie die strenge Typprüfung eingeschaltet, so sind *Char*-Literale mit dem Literalzeichen *c* abzuschließen.

BEISPIEL 2.31: Einer *Char*-Variablen wird das Zeichen "A" zugewiesen.

VB
```
Dim z As Char = "A"c
```

Stringausdrücke schließt man in "Gänsefüßchen" ein.

```
Dim s As String = "Ich bin ein String!"
```

Einzelne Strings kann man mit dem "+"- oder "&"-Operator zusammenfügen. Letztere Variante (Kaufmanns-UND) wird aber wärmstens empfohlen.

BEISPIEL 2.32: Addition von zwei Zeichenketten

VB
```
Dim s1 As String = "Hallo ", s2 As String = " .NET Freunde!"
MessageBox.Show(s1 & s2)
```

2.3.4 Date

Datums-/Zeitwerte werden in Variablen vom *Date*-Typ als ganze 64-Bit-Zahlen (8 Bytes) gespeichert. Der Bereich umfasst den 01. Januar des Jahres 1 unserer Zeitrechnung bis zum 31. Dezember 9999 und Uhrzeiten von 0:00:00 (Mitternacht) bis 23:59:59.

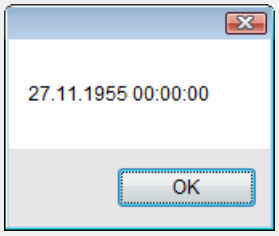
Ein *Date*-Literale muss durch das Rautenzeichen (#) eingeschlossen sein und im Format *m/d/yyyy* (US-amerikanische Schreibweise) angegeben werden.

Werden *Date*-Werte in den *String*-Typ konvertiert, wird das Datum entsprechend des in der Systemsteuerung des Computers eingestellten **kurzen** Datumsformats dargestellt, die Uhrzeit entspricht dem eingestellten Zeitformat (12 oder 24 Stunden).

BEISPIEL 2.33: Der 27.11.1955 wird einer Datumsvariablen zugewiesen und in einer MessageBox angezeigt.

VB
```
Dim dat As Date = #11/27/1955#        ' Reihenfolge: Monat/Tag/Jahr
MessageBox.Show(dat.ToString)
```

Ergebnis



Wollen Sie zusätzlich zum Datum die Uhrzeit mit angeben, muss diese dahinter im Format *hh:mm:ss AM|PM* notiert werden.

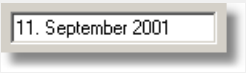
BEISPIEL 2.34: Der 17. August 1987, 19.00 Uhr

VB
```
Dim dat As Date = #8/17/1987 7:00:00 PM#
```

Um auch deutsche Datumsstrings zuweisen zu können, können Sie z.B. die *CDate*-Typkonvertierungsfunktion verwenden.

BEISPIEL 2.35: Ein Datum wird über eine *TextBox* zugewiesen.

VB




```
Dim dat As Date = CDate(TextBox1.Text)
```

HINWEIS: Weitere Informationen über Datums-/Zeitfunktionen bzw. die Methoden der *Date*-Klasse entnehmen Sie dem Kapitel 4.

2.3.5 Boolean

Ein *Boolean* beansprucht satte 16 Bit, obwohl eigentlich nur ein einziges Bit zum Speichern von *True* bzw. *False* reichen würde, aber hinter dieser scheinbar unglaublichen Verschwendung stecken rechentechnische Gründe. Standardmäßig wird eine *Boolean*-Variable bei ihrer Deklaration mit *False* initialisiert.

False entspricht einem Integer-Wert von 0, wenn Sie *Boolean* in *Integer* konvertieren, *True* entspricht dem *Integer*-Wert -1. Etwas anders sieht es aus, wenn Sie umgekehrt einen Integer in einen *Boolean* konvertieren wollen, denn dann wird jeder von null abweichende Wert zu *True* (siehe auch Typkonvertierung im Abschnitt 2.4.8).

2.3.6 Object

Dieser universelle Datentyp spielt als "Mutter für alles" eine Sonderrolle, denn ihm können Sie beliebige Datentypen zuweisen. Eine *Object*-Variable ist ein so genannter *Referenztyp*, denn sie speichert nicht den tatsächlichen Wert, sondern lediglich einen 4 Byte großen Zeiger auf die Adresse der zugewiesenen Variablen.

Vor der Ausführung arithmetischer Operationen und bei eingeschalteter strenger Typprüfung (*Option Strict On*) müssen Sie einen *Object*-Typ mit *CType* (siehe 2.4.8) immer in den gewünschten Typ konvertieren.

BEISPIEL 2.36: Zwei Objektvariablen referenzieren eine *String*- bzw. eine *Integer*-Variable und werden (nach erfolgter Typkonvertierung) zur ersten Objektvariablen addiert. Das Ergebnis wird in einen String konvertiert und angezeigt.

VB
```
Dim s As String = "16"
Dim i As Integer = 20
Dim o1 As Object = s
Dim o2 As Object = i
o1 = CType(o1, Integer) + CType(o2, Integer)     ' referenziert Integer mit Wert 36
MessageBox.Show(o1.ToString)                     ' Ergebnis ist "36"
```

HINWEIS: Den *Object*-Datentyp sollten Sie nur in Ausnahmefällen verwenden, da mit ihm die Typsicherheit (trotz *Option Strict On*) verloren geht (Fehlerquelle!).

2.4 Konvertieren von Datentypen

Visual Basic nimmt es mit den Datentypen sehr genau, zumindest wenn wir "sauber" programmieren wollen und dazu die *Option Strict On* eingeschaltet haben.

2.4.1 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 2.37: Die Zuweisung *Byte* zu *Integer* funktioniert problemlos.

VB
```
Dim b As Byte = 100
Dim i As Integer = b
```

Geradezu oberlehrerhaft verhält sich .NET im umgekehrten Fall. Egal ob der Wert in den kleineren Datentyp passen würde oder nicht – es wird halt gemeckert.

BEISPIEL 2.38: Obwohl der Wert *100* problemlos von einer *Byte*-Variablen aufgenommen werden könnte, erscheint eine Fehlermeldung.

VB
```
Dim i As Integer = 100
Dim b As Byte = i       ' Fehler!
```

Um den meckernden Compiler zu beschwichtigen, ist eine so genannte *explizite Typkonvertierung* erforderlich, für die es unter VB.NET verschiedene Wege gibt.

- Verwendung einer sprachspezifischen Konvertierungsfunktion, wie z.B. *CByte* oder *CDbl* (siehe Tabelle Seite 120).
- Konvertieren mit der *CType*-Funktion
- Konvertierungen über die *Convert*-Klasse
- Einsatz des *TryCast*-Operators
- Verwenden der *ToString*- bzw. *Parse*- Methode (für Konvertierung in/vom *String*-Datentyp)

BEISPIEL 2.39: Drei explizite Typkonvertierungen mit dem gleichen Ergebnis.

VB
```
Dim i As Integer = 100
Dim b As Byte = CByte(i)            ' Variante 1
Dim b As Byte = CType(i, Byte)      ' Variante 2
Dim b As Byte = Convert.ToByte(i)   ' Variante 3
```

HINWEIS: Zur Anwendung der Methoden *ToString* und *Parse* siehe 2.4.5, zum *TryCast*-Operator siehe 2.4.9.

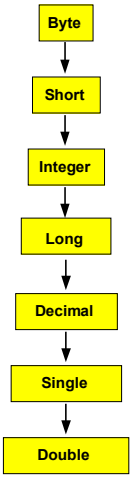
2.4.2 Welcher Datentyp passt zu welchem?

Bei einer impliziten Konvertierung unter *Option Strict On* kann stets nur der "schmalere" der beiden Datentypen in einen "breiteren" umgewandelt werden[1].

BEISPIEL 2.40: Implizite Typkonvertierung *Byte* in *Double*

VB
```
Dim d As Double
Dim b As Byte = 100
d = b                       ' d erhält den Wert 100
```

In der folgenden Abbildung entnehmen Sie der Pfeilrichtung, welcher Typ automatisch welchen Typ aufnehmen kann.



In obiger Abbildung werden Sie die Datentypen *Char*, *String* und *Boolean* vermissen. Für die beiden letzteren ist generell keinerlei automatische Typkonvertierung möglich, hier bleibt Ihnen nur die explizite Konvertierung.

Die implizite (sprich automatische) Konvertierung eines *Char* in einen *String* ist jedoch möglich.

BEISPIEL 2.41: Umwandlung eines Zeichens in einen String

VB
```
Dim zeichen As Char = "A"c
Dim s As String = zeichen          ' s ergibt sich zu "A"
```

[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 drin sind!

2.4.3 Konvertierungsfunktionen

Zur expliziten Typkonvertierung stellt Visual Basic gnädigerweise für jeden fundamentalen Datentyp eine Funktion zur Verfügung (siehe Tabelle).

Konvertierungsfunktion	Datentyp	Konvertierungsfunktion	Datentyp
CShort(*Ausdruck*)	*Short*	CBool(*Ausdruck*)	*Boolean*
CInt(*Ausdruck*)	*Integer*	CByte(*Ausdruck*)	*Byte*
CLng(*Ausdruck*)	*Long*	CChar(*Ausdruck*)	*Char*
CDec(*Ausdruck*)	*Decimal*	CDate(*Ausdruck*)	*Date*
CSng(*Ausdruck*)	*Single*	CStr(*Ausdruck*)	*String*
CDbl(*Ausdruck*)	*Double*	CObj(*Ausdruck*)	*Object*

BEISPIEL 2.42: Alle drei Variablen haben unterschiedliche Datentypen.

VB
```
Dim a As Integer = 10000, b As Double = 10000000000000.0, c As Single
c = CSng(a + b)
MessageBox.Show(b.ToString)   '  zeigt "1E+13"
```

BEISPIEL 2.43: Ein *True* entspricht hier dem Integer-Wert -1 (*False* ist 0).

VB
```
Dim b As Boolean = True
Dim i As Integer
i = CInt(b)            ' i erhält den Wert -1
```

HINWEIS: Während bei impliziten Typkonvertierungen nichts verloren geht, können bei expliziten Konvertierungen durchaus Genauigkeitsverluste auftreten.

BEISPIEL 2.44: Bei der Konvertierung *Double* zu *Integer* werden die Nachkommastellen gerundet.

VB
```
Dim i As Integer
Dim d As Double = 12.7456789
i = CInt(d)            ' i erhält den Wert 13
```

BEISPIEL 2.45: Eine *Integer*-Zahl ungleich null wird immer zu *True* konvertiert.

VB
```
Dim b As Boolean
Dim i As Integer = 10
b = CBool(i)          ' b erhält den Wert True
```

BEISPIEL 2.46: Wird ein *String* mit *CChar* in eine *Char*-Variable kopiert, so erhält diese das erstemodulglobal Zeichen des Strings.

VB
```
Dim s As String = "Hallo"
```

BEISPIEL 2.46: Wird ein *String* mit *CChar* in eine *Char*-Variable kopiert, so erhält diese das erstemodulglobal Zeichen des Strings.

```
Dim z As Char = CChar(s)      ' z erhält den Wert "H"
```

2.4.4 CType-Funktion

Mit Hilfe dieser Funktion können Sie die explizite Konvertierung der verschiedenen Standarddatentypen vornehmen.

SYNTAX: `CType(expr As Object, type As Object)`

expr = zu konvertierender Ausdruck; *type* = Typbezeichner des Zieltyps

BEISPIEL 2.47: Konvertieren eines *Double*- in einen *Integer*-Typ

VB
```
Dim i As Integer
Dim d As Double = 12.7456789
i = CType(d, Integer)            ' i erhält den Wert 13
```

2.4.5 Konvertieren von Strings

Beim *String*-Datentyp scheint es zunächst ähnlich trübe wie bei *Boolean* auszusehen. 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!

BEISPIEL 2.48: Anzeige einer Gleitkommazahl

VB
```
Dim d As Double = 12.75
MessageBox.Show(d.ToString)
```

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 2.49: Einem *Char* wird das zweite Zeichen eines *String* zugewiesen.

```
VB  Dim name As String = "Max"
    Dim c As Char = 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 2.50: Das geht leider nicht[1]

```
VB  Dim s As String = "5"
    Dim i As Integer = s           ' Fehler!
```

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

BEISPIEL 2.51: Das Vorgängerbeispiel kann wie folgt sauber gelöst werden

```
VB  Dim s As String = "5"
    Dim I As Integer = Convert.ToInt32(s)
    MessageBox.Show(i.ToString)      ' zeigt "5"
```

BEISPIEL 2.52: Ein *String* wird in eine *Double*-Zahl konvertiert

```
VB  Dim s As String = "23,50"
    Dim d As Double = Convert.ToDouble(s)    ' d erhält den Wert 23,50
```

Alternativ kann auch die *Parse*-Methode eingesetzt werden (siehe 2.4.7):

BEISPIEL 2.53: Konvertieren eines Stringliterals in eine Ganzzahl.

```
VB  Dim nr As Integer = 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 Übergabewerte als Zeichenketten vorliegen (*Text*-Eigenschaft der Ein- und Ausgabefelder), müssen sie zunächst in Zahlentypen umgewandelt werden, um dann nach ihrer Verarbeitung wieder in Zeichenketten zurückverwandelt und zur Anzeige gebracht zu werden.

[1] Wir beziehen uns auf *Option Strict On*

BEISPIEL 2.54: Ein Ausschnitt aus dem Einführungsbeispiel 1.7.2

```
VB euro = Convert.ToSingle(TextBox1.Text)          ' Eingabe: String => Single
   dollar = euro * kurs                            ' Verarbeitung
   TextBox2.Text = dollar.ToString("#,##0.00")     ' Ausgabe: Single => String
```

2.4.6 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*`(`*expr* `As Object)`

typeMethode = eine der Konvertierungsmethoden (*ToBoolean*, *ToByte, ToInt32, ...*)

expr = zu konvertierender Ausdruck

BEISPIEL 2.55: Ein *String* wird in eine *Double*-Zahl konvertiert.

```
VB Dim s As String = "55,7"
   Dim d As Double = Convert.ToDouble(s)
   MessageBox.Show(d.ToString)                ' zeigt "55,7"
```

BEISPIEL 2.56: *Boolean* wird in *Integer* und in *String* konvertiert

```
VB Dim b As Boolean = True
   Dim i As Integer = Convert.ToInt32(b)     ' 1
   b = False
   i = Convert.ToInt32(b)                    ' 0
   Dim s As String = Convert.ToString(b)     ' "False"
```

2.4.7 Die Parse-Methode

Die numerischen Typen *Byte*, *Integer*, *Single* und *Double* verfügen u.a. über eigene (statische) *Parse*-Methoden, welche die Stringdarstellung einer Zahl in den entsprechenden Typ konvertieren kann.

BEISPIEL 2.57: Der Inhalt einer *TextBox* wird in eine Gleitkommazahl konvertiert.

```
VB Dim z As Double = Double.Parse(TextBox1.Text)
```

BEISPIEL 2.58: Konvertieren eines Stringliterals in eine Ganzzahl

```
VB Dim nr As Integer = Int32.Parse("12")
```

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

2.4.8 Boxing und Unboxing

Die Begriffe *Boxing/Unboxing* gehören zu den häufig strapazierten .NET-Schlagwörtern. 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 2.59: Boxing und Unboxing

VB

Eine *Boolean*-Variable wird in ein *Object* "verpackt" (Boxing) und dieses anschließend einer zweiten *Boolean*-Variablen zugewiesen (Unboxing).

```
Dim b1 As Boolean = True
Dim o As Object = b1                      ' ok, implizite Konvertierung (Boxing)
Dim b2 As Boolean  = o                    ' Fehler, implizite Konvertierung schlägt fehl
Dim b2 As Boolean = Convert.ToBoolean (o)  ' ok, explizite Konvertierung (Unboxing, True)
```

Um den tieferen Sinn von Boxing/Unboxing zu verstehen, sollten Sie sich in Abschnitt 2.2.2 nochmals den Unterschied zwischen den beiden fundamentalen Arten von Datentypen verdeutlichen, d.h., zwischen Werte- und den Verweistypen.

Boxing

Es erhebt sich die Frage, was denn passiert, wenn man einer *Object*-Variablen einen Wertetyp zuweist, der naturgemäß im Stack gespeichert ist.

BEISPIEL 2.60: Ein Integer wird einem *Object*-Datentyp zugewiesen.

VB

```
Dim i As Integer = 25
Dim o As Object = i
```

Die genaue 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 2.61: Das Vorgängerbeispiel wird fortgesetzt

VB
```
...
Dim j As Integer = o          ' Fehler!
Dim j As Integer = CInt(o)    ' ok
```

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 2.62: Ja, auch das funktioniert!

VB
```
Dim i As Integer = New Integer()
i = 12
```

2.4.9 TryCast-Operator

Eine weitere Alternative zur expliziten Typumwandlung (bzw. Unboxing) bietet der *TryCast*-Operator, der allerdings nur auf Verweis- und nicht auf Wertetypen anwendbar ist. Auch alle Steuerelemente gehören zu den Verweistypen.

BEISPIEL 2.63: Konvertieren des *sender*-Parameters eines Eventhandlers.

VB
```
Me.Text = (TryCast(sender, TextBox)).Text
```

Misslingt die Konvertierung, so wird kein Fehler ausgelöst, sondern der Variablen wird der Wert *Nothing* zugewiesen.

2.4.10 Nullable Types

Ein weiterer Unterschied zwischen Wertetypen, wie *Integer* oder *Structure*, und Referenztypen, wie *Form* oder *String*, ist der, dass Referenztypen so genannte Null-Werte unterstützen. Eine Referenztyp-Variable kann also den Wert *Nothing* enthalten, d.h., die Variable referenziert im Moment keinen bestimmten Wert. Demgegenüber enthält eine Wertetyp-Variable immer einen Wert, auch wenn dieser, wie bei einer *Integer*-Variablen, den Wert 0 (null) hat. Falls Sie einer Wertetyp-Variablen *Nothing* zuweisen, wird diese auf ihren Default-Wert zurückgesetzt, bei einem *Integer* wäre das 0 (null)).

Der Compiler kann aber durch ein der Typdeklaration nachgestelltes Fragezeichen (?) einen Wertetyp in eine generische *System.Nullable(Of T As Structure)*-Struktur verpacken.

BEISPIEL 2.64: Einige Deklarationen von Nullable-Typen

VB
```
Dim i As Integer? = 10
Dim j As Integer? = Nothing
Dim k As Integer? = i + j             ' Nothing
```

BEISPIEL 2.65: Ein an die Subroutine *PrintValue* übergebener *Integer*-Wert wird nur angezeigt, wenn ihm ein Wert zugewiesen wurde. Ansonsten erfolgt die Ausgabe "Null Wert".

VB
```
Sub PrintValue(i As Nullable(Of Integer))
    If i.HasValue Then
      Console.WriteLine(CInt(i))
    Else
      Console.WriteLine("Null Wert!")
    End If
End Sub
```

Von Nutzen dürften Nullable-Typen besonders dann sein, wenn Daten aus einer relationalen Datenbank gelesen bzw. dorthin zurückgeschrieben werden sollen (siehe dazu Kapitel 10 und 11).

2.5 Operatoren

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

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

Doch bevor es richtig losgeht sollten wir uns mit dem Begriff *Ausdruck* etwas anfreunden. Man versteht darunter die kleinste ausführbare Einheit eines Programms. Ein Ausdruck setzt zumindest einen Operator voraus und benötigt meist zwei Operanden.

BEISPIEL 2.66: Operatoren

VB
Im Ausdruck

```
i = 12
```

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

2.5.1 Arithmetische Operatoren

Neben den vier Grundrechenarten werden folgende Operatoren unterstützt:

Operator	Beispielausdruck	Erklärung
+	x + y	Addition
-	x - y -x	a) Subtraktion b) negative Zahl

Operator	Beispielausdruck	Erklärung
*	x * y	Multiplikation
/	x / y	Division
\	x \ y	Integer-Division (liefert nur ganzzahligen Anteil)
Mod	x Mod y	Modulo-Division (liefert Restwert)
^	x ^ y	Potenzoperator
&	"a" & "b"	Addition von Zeichenketten

BEISPIEL 2.67: Die Ergebnisse einiger arithmetischer Operationen werden kommentiert.

VB
```
Dim i As Integer, d As Double
i = 3 *(4 + 5) * 6       ' 162
d = 3 ^ 2                ' 9
d = 7 / 3                ' 2.3333333333333
d = 7 \ 3                ' 2 (ganzzahlige Division!)
i = 7 Mod 3              ' 1 (Rest!)
```

2.5.2 Zuweisungsoperatoren

In Ergänzung zum normalen Zuweisungsoperator (=) gibt es für fast alle arithmetischen Operatoren eine Kurzformnotation (siehe folgende Tabelle).

Operator	Beispielausdruck	Erklärung
=	x = y	x wird der Wert von y zugewiesen
+=	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 zu x / y (Nachkommastellen bleiben)
\=	x \= y	x ergibt sich zu x \ y (Nachkommastellen abgeschnitten)
^=	x ^= y	x wird mit y potenziert
&=	x &= y	an String x wird String y angehängt

Der einfache Zuweisungsoperator ist für den Einsteiger immer etwas problematisch, da er sich rein äußerlich nicht vom Vergleichsoperator unterscheidet.

BEISPIEL 2.68: Das erste "=" wird als Zuweisungs-, das zweite "=" hingegen als Vergleichsoperator interpretiert, d.h., *x* wird *True*, falls *y* gleich *z* ist, sonst *False*.

VB
```
x = y = z
```

Um die Lesbarkeit zu verbessern, sollte man den Vergleichsausdruck klammern:

```
x = (y = z)
```

Die verkürzten Zuweisungsoperatoren nur bringen relativ bescheidene Verbesserungen.

BEISPIEL 2.69: Verkürzter Zuweisungsoperator

VB

Anstatt

```
i = i + 1
```

kann man schreiben

```
i += 1
```

2.5.3 Logische Operatoren

Logische Operatoren verknüpfen zwei Boolesche Variablen bzw. Ausdrücke miteinander, um daraus einen neuen *True-/False*-Wert zu ermitteln.

Operator	Beispielausdruck	Erklärung
And	*x And y*	**Und**: liefert *True*, wenn beide Operanden *True* sind
Or	*x Or y*	**Oder**: liefert *True*, wenn mindestens einer der Operanden *True* ist
Xor	*x Xor y*	**Exklusiv Oder**: liefert *True*, wenn genau einer der beiden Operanden *True* ist
AndAlso	*x AndAlso y*	**Spezielles Und**: ist der erste Operator *False*, wird der zweite nicht mehr überprüft
OrElse	*x OrElse y*	**Spezielles Oder**: ist der erste Operator *True*, wird der zweite nicht mehr überprüft
Not	*Not x*	**Negation**: negiert den Wahrheitswert

Hier die Wahrheitstabelle der wichtigsten zweistelligen logischen Operationen:

Operand 1	Operand 2	And	Or	Xor
False	*False*	*False*	*False*	*False*
False	*True*	*False*	*True*	*True*
True	*False*	*False*	*True*	*True*
True	*True*	*True*	*True*	*False*

BEISPIEL 2.70: Der Variablen *b* wird der Wert *True* zugewiesen[1].

VB

```
Dim b As Boolean = True And True Or False And False
```

[1] Umständlicher geht es sicherlich kaum, aber Sie wollen ja schließlich etwas lernen!

2.5.4 Vergleichsoperatoren

Vergleichs- oder relationale Operatoren vergleichen zwei Ausdrücke miteinander und liefern als Ergebnis einen Wahrheitswert. In Visual Basic ist das übliche Angebot enthalten.

Operator	Beispielausdruck	Erklärung
=	x = y	Arithmetische Vergleiche. (der Stringvergleich wird von links nach rechts entsprechend dem ANSI-Code der Zeichen durchgeführt)
<	x < y	
<=	x <= y	
>	x > y	
>=	x >= y	
<>	x <> y	

BEISPIEL 2.71: Und-Verknüpfung von zwei Ausdrücken.

VB
```
Dim x As Integer = 5, b As Boolean
b = (x < 5) And (x >= 0)                ' b ist False
```

Im obigen Beispiel steht das Ergebnis bereits nach dem Auswerten des ersten Ausdrucks fest. Man könnte also Rechenzeit einsparen, wenn man auf das Auswerten des zweiten Ausdrucks verzichten würde. Um ein solches Verhalten zu erreichen, stehen die logischen Operatoren *AndAlso* und *OrElse* zur Verfügung.

BEISPIEL 2.72: Das Vorgängerbeispiel wird rationeller programmiert.

VB
```
Dim x As Integer = 5, b As Boolean
b = (x < 5) AndAlso (x >= 0)            ' b ist False
```

2.5.5 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
()
^
–
/
\
Mod

Operator
+ -
= < >
<> <= >=
Not
And AndAlso *Or OrElse*
Xor

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

BEISPIEL 2.73: Arithmetische Operationen

```
VB  Dim y As Double = 2 ^ 2 + 1 + 2 / 4              ' y = 5,5
```

BEISPIEL 2.74: Boolesche Operationen

```
VB  Dim b As Boolean = Not True And False Or 5 > 6    ' b = False
```

HINWEIS: Durch Klammern kann die hierarchische Reihenfolge außer Kraft gesetzt werden[1].

2.6 Kontrollstrukturen

Schleifen- und Verzweigungsanweisungen unterbrechen den linearen Programmablauf und gehören zum Einmaleins des Programmierens.

2.6.1 Verzweigungsbefehle

"Programmweichen" werden durch Entscheidungsanweisungen (Verzweigungen) gestellt. Die folgende Tabelle gibt einen Überblick über die zum "Basic-Urgestein" zählenden Entscheidungsbefehle.

Verzweigungsanweisung	Erklärung

`If` *Bedingung* `Then` *Anweisungen*
`[Else` *Anweisungen*`]`
`End If`
	Bedingte Verzweigung, wenn die gesamte *If Then*-Anweisung in einer einzigen Zeile steht, kann auf *End If* verzichtet werden.

`If` *Bedingung1* `Then`
Anweisungen
`[ElseIf` *Bedingung2* `Then`
	Blockstruktur *If...ElseIf...End If*
	Jede Zeile muss mit *Then* enden!

[1] Da Sie aber in der Schule gut aufgepasst haben, werden wir auf weitere Beispiele verzichten.

Verzweigungsanweisung	Erklärung
Anweisungen **ElseIf** *Bedingung3* **Then** *Anweisungen ...*] [**Else** *Anweisungen*] **End If**	*Else*-Anweisungen werden dann ausgeführt, wenn keine der *If*- bzw. *ElseIf*-Bedingungen zutrifft.
Select Case *Ausdruck* **Case** *Ausdruck1* *Anweisungen* [**Case** *Ausdruck2* *Anweisungen ...*] [**Case Else** *Anweisungen*] **End Select**	Blockstruktur *Select Case/Case/End Select* Der Ausdruck kann eine Variable oder ein beliebiger Ausdruck sein, der mit den hinter *Case* angeführten Ausdrücken verglichen wird. Nach erstem Erfolg wird der Block verlassen!

In den meisten Fällen werden Sie zum Prüfen von Bedingungen die *If-Then*-Anweisung verwenden.

HINWEIS: Wenn Sie die *If-Then*-Anweisung in **einer** Programmzeile unterbringen, braucht das Blockende nicht mit *End If* markiert zu werden.

BEISPIEL 2.75: In dieser einzeiligen *If-Then*-Anweisung wird im *Label* "Verbessern!" angezeigt.

VB
```
Dim zensur As Byte = 3
If zensur = 1 Then Label1.Text = "Gratuliere!" Else Label1.Text = "Verbessern!"
```

BEISPIEL 2.76: Die Alternative für das Vorgängerbeispiel braucht zwar mehr Platz, ist aber übersichtlicher.

VB
```
Dim zensur As Byte = 3
If zensur = 1 Then
   Label1.Text = "Gratuliere!"
Else
   Label1.Text = "Verbessern!"            ' zutreffende Bedingung
End If
```

Optional können Sie im *If-Then*-Block noch *ElseIf*- und *Else*-Zweige verwenden, wobei die *ElseIf*-Bedingung nur dann geprüft wird, wenn keine der vorstehenden *If*-Bedingungen erfüllt war.

BEISPIEL 2.77: Im *Label* wird "Befriedigend" angezeigt.

VB
```
Dim zensur As Byte = 3
If zensur = 1 Then
   Label1.Text =  "Sehr gut!"
ElseIf zensur = 2 Then
   Label1.Text =  "Gut"
```

BEISPIEL 2.77: Im *Label* wird "Befriedigend" angezeigt.

```
ElseIf zensur = 3 Then                ' zutreffende Bedingung
   Label1.Text =  "Befriedigend"
 '(usw.)
End If
```

Mit *Select Case* wird ein Ausdruck auf mehrere mögliche Ergebnisse hin überprüft. Im Testausdruck kann ein beliebiger arithmetischer oder logischer Ausdruck stehen.

BEISPIEL 2.78: Diese Kontrollstruktur leistet das Gleiche wie das Vorgängerbeispiel.

VB
```
Dim zensur As Byte = 3
Select Case zensur
   Case 1: Label1.Text = "Sehr gut"
   Case 2: Label1.Text =  "Gut"
   Case 3: Label1.Text =  "Befriedigend"     ' zutreffende Bedingung
   '(usw.)
End Select
```

In einem *Case*-Zweig können auch mehrere Bedingungen geprüft werden.

BEISPIEL 2.79: Das *Label* zeigt "Frühling" an.

VB
```
Dim monat As Byte = 5
Select Case monat
   Case 12,1,2: Label1.Text =  "Winter"
   Case 3,4,5: Label1.Text =  "Frühling"       ' zutreffende Bedingung
   Case 6,7,8: Label1.Text =  "Sommer"
   Case 9,10,11: Label1.Text =  "Herbst"
   Case Else
      Label1.Text =  "kein gültiger Monat!"
End Select
```

Alternativ zum Aufzählen mehrerer Bedingungen kann mittels *To* auch ein Bereich angegeben werden.

BEISPIEL 2.80: Ein Zweig aus dem Vorgängerbeispiel könnte wie folgt ersetzt werden.

VB
```
...
  Case 3 To 5: Label1.Text = "Frühling"
...
```

HINWEIS: Sie sollten, wo immer es geht, anstelle einer *If-Then*-Anweisung mit eingeschachtelten *ElseIf*-Verzweigungen eine *Select Case*-Anweisung verwenden. *Select Case* wird wesentlich schneller ausgeführt, da die Prüfbedingung nur einmal auszuwerten ist.

2.6.2 Schleifenanweisungen

Visual Basic kennt zwei Grundtypen[1]:

- *For-Next*- und
- *Do-Loop*-Schleifen .

Die folgende Tabelle gibt einen Überblick über alle möglichen Schleifenkonstruktionen:

Schleifenanweisung	Erklärung

`For` *Zähler=Anfangs* `To` *Endwert* `[Step` *Schritt*`]`
  Anweisungen
  `[Exit For]`
  Anweisungen
`Next [`*Zähler*`]`
	For...Next-Zählschleife,
	Abbruch mit *Exit For*, ohne *Step* ist Schritt 1

`Do [While|Until` *Bedingung*`]`
  Anweisungen
  `[Exit Do]`
  Anweisungen
`Loop`
	Do While...Loop-Bedingungsschleife,
	Abbruchbedingung am Schleifenanfang

`Do`
  Anweisungen
  `[Exit Do]`
  Anweisungen
`Loop [While|Until` *Bedingung*`]`
	Do...Loop While-Bedingungsschleife,
	Abbruchbedingung am Schleifenende

`While` *Bedingung* *Anweisungen* `End While`	gleichbedeutend mit *Do While ... Loop*-Bedingungsschleife

For-Next-Schleifen

Bei diesem klassischen Schleifentyp wird die Zählervariable automatisch hochgezählt (inkrementiert) bzw. heruntergezählt (dekrementiert).

BEISPIEL 2.81: Zehnmal untereinander den laufenden Index und einen Text in einer *ListBox* ausgeben

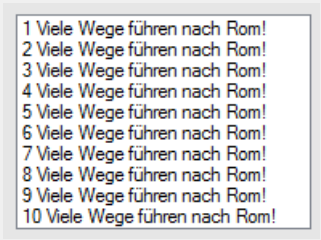
VB
```
Dim i As Integer
For i = 1 To 10
       ListBox1.Items.Add(i.ToString & " Viele Wege führen nach Rom!")
Next i
```

Die Angabe der Zählervariablen *i* nach *Next* kann auch weggelassen werden, wird aber aus Gründen der Übersichtlichkeit empfohlen.

[1] Auf *For-Each*-Schleifen gehen wir en passant erst im Rahmen der objektorientierten Programmierung ein, siehe auch Praxisbeispiel 2.9.3.

BEISPIEL 2.81: Zehnmal untereinander den laufenden Index und einen Text in einer *ListBox* ausgeben

Ergebnis



Do-Loop-Schleifen

Schleifen dieses Typs erlauben eine flexible Programmierung, da die Abbruchbedingung sehr variabel ist. Um das Inkrementieren/Dekrementieren der Zählervariablen muss man sich allerdings selbst kümmern.

Bei *While* wird die Schleife ausgeführt, **während** die Bedingung erfüllt ist, bei *Until* nur so lange, **bis** die Bedingung erfüllt ist.

In Abhängigkeit davon, ob die Abbruchbedingung am Schleifenanfang oder an deren Ende kontrolliert wird, spricht man auch von *kopfgesteuerten* bzw. *fußgesteuerten* Schleifen.

BEISPIEL 2.82: Ein identisches Resultat wie die obige *For-Next*-Schleife ergibt die folgende *Do While ... Loop*-Schleife

VB

```
Dim i As Integer = 1
Do While i <= 10
   ListBox1.Items.Add(i.ToString & " Viele Wege führen nach Rom!")
   i += 1
Loop
```

Ein umfassendes Testprogramm mit weiteren möglichen *Do ... Loop*-Schleifenkonstruktionen finden Sie im Praxisbeispiel

► 2.9.3 Schleifenanweisungen kennen lernen

2.7 Benutzerdefinierte Datentypen

Sie sind als Programmierer natürlich nicht nur auf die einfachen Datentypen *Integer, Single, ...* angewiesen, sondern können auch selbst neue, komplexere Datentypen kreieren. Wir wollen in diesem Abschnitt Aufzählungstypen (Enumerationen) und strukturierte Datentypen behandeln.

2.7.1 Enumerationen

Sammlungen von miteinander verwandten Konstanten können in so genannten Enumerationen (*Enum*s) zusammengefasst werden.

HINWEIS: Alle in der Enumeration enthaltenen Konstanten müssen vom gleichen Datentyp sein, zulässig sind nur *Byte*, *Short*, *Integer* und *Long*.

```
SYNTAX: [Private|Public] Enum Name As Datentyp
            Konstante1 = wert1
            Konstante2 = wert2
            ...
        End Enum
```

Falls die Angabe des Datentyps weggelassen wird, handelt es sich automatisch um *Integer*-Konstanten.

BEISPIEL 2.83: Eine Enumeration für drei Konstanten vom *Byte*-Datentyp

VB
```
Public Enum erstesQuartal As Byte
        JANUAR = 1
        FEBRUAR
        MÄRZ
End Enum
```

Es genügt, wenn nur der ersten Mitgliedskonstanten (*JANUAR*) ein (*Byte*-)Wert zugewiesen wird, der Wert der Nachfolger wird automatisch um eins erhöht.

Auf die deklarierten Konstanten kann direkt zugegriffen werden.

BEISPIEL 2.84: Verwendung der oben deklarierten Enumeration

VB
```
Const monat As Byte = erstesQuartal.FEBRUAR
```

Auch eine vorherige Variablendeklaration ist möglich.

BEISPIEL 2.85: Eine Modifikation des Vorgängerbeispiels

VB
```
Dim quartal As erstesQuartal
Const monat As Byte = quartal.FEBRUAR
```

2.7.2 Strukturen

Mit der *Structure*-Anweisung definieren Sie komplexe Datentypen, die ein oder auch mehrere Element(e) enthalten dürfen.

HINWEIS: In diesem Abschnitt wollen wir Strukturen nicht tiefgründiger behandeln, sondern nur einen ersten Einblick gewähren.

Deklaration

Vom Prinzip her entspricht die Definition einer *Struktur* der einer *Klasse* (siehe Kapitel 3), allerdings sind Strukturen *Wertetypen*, während Klassen zu den *Verweis- bzw. Referenztypen* gehören.

Eine einfache Struktur hat folgenden Aufbau:

```
SYNTAX: [Private|Public] Structure Name
            Dim  Mitglied1 As Datentyp
            ...
        End Structure
```

Beachten Sie:

- Die *Structure*-Anweisung ist auf lokaler bzw. Prozedurebene unzulässig und nur auf globaler Ebene anwendbar.
- Damit die Elemente einer Struktur von außerhalb sichtbar sind, muss der *Public*-Modifizierer vorangestellt werden.[1]

BEISPIEL 2.86: Structure-Deklaration

VB

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

```
Public Structure Person
    Dim Vorname, Nachname As String
    Dim Alter As Byte
End Structure
```

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 2.87: *(Fortsetzung)* Sie erzeugen zwei Variablen des oben definierten Datentyps *Person*:

VB

```
Dim person1, person2 As Person
```

Datenzugriff

Um auf den Wert einer Strukturvariablen zuzugreifen, müssen Name und Element durch einen Punkt (so genannter *Qualifizierer*) voneinander getrennt sein[2].

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

[2] Dies entspricht der Schreibweise beim Zugriff auf Objekteigenschaften.

BEISPIEL 2.88: Fortsetzung des Vorgängerbeispiels

```
VB  Dim a As Integer, name As String
    person1.Vorname = "Max"        ' Schreibzugriff
    person1.Nachname = "Müller"
    person1.Alter = 50
    a = person1.Alter              ' Lesezugriff
    name = person1.Vorname & "  " & person1.Nachname
```

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

BEISPIEL 2.89: Die Variable *person1* wird "geklont".

```
VB  person2 = person1
```

With-Anweisung

Ein vereinfachter Zugriff auf Strukturvariablen ist mit der *With- End With*-Anweisung möglich.

SYNTAX:
```
With Strukturvariable
  ' Anweisungen
End With
```

BEISPIEL 2.90: Das Vorgängerbeispiel könnte auch so geschrieben werden:

```
VB  Dim a As Integer, name As String
    With person1
       .Vorname = "Max"
       .Nachname = "Müller"
       .alter = 50
       a = .alter
       name = .Vorname & "  " & .Nachname
    End With
```

HINWEIS: Mit der *With*-Anweisung kann auch der Zugriff auf Eigenschaften und Methoden beliebiger Objekte vereinfacht bzw. übersichtlicher gestaltet werden (siehe dazu Kapitel 3).

Bemerkungen

Wie bereits erwähnt, 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*, *Nothing* oder *False* initialisiert.

BEISPIEL 2.91: Die Strukturvariable *person1* wird mit ihrem Standard-Konstruktor erzeugt und initialisiert. Das Feld *alter* hat automatisch den Wert *0*.

VB
```
Dim person1 As New Person()
Dim a As Integer = person1.Alter              ' a erhält 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*), der Heap ist für Wertetypen tabu!

2.8 Nutzerdefinierte Funktionen/Prozeduren

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

Funktionen und Prozeduren sind unter Visual Basic zunächst das, was man in den Anfangszeiten der Programmierung als "Unterprogramm" bezeichnet hat. Später, im OOP-Kapitel 3, werden Sie feststellen, dass Funktionen und Prozeduren zu Methoden einer Klasse mutiert sind.

2.8.1 Deklaration und Syntax

Falls Sie nur an einem einzigen Rückgabeparameter interessiert sind, sollten Sie einer Funktion den Vorzug vor einer Prozedur geben, andernfalls ist meistens eine Prozedur die richtige Entscheidung.

Function

Funktionen deklarieren Sie mit dem Schlüsselwort *Function*. Als Funktionstyp bzw. Rückgabewert kommt eigentlich jeder Datentyp in Frage. Jede *Function* hat einen Namen und einen Körper. Letzterer enthält die beim Methodenaufruf auszuführenden Anweisungen. Ein vorzeitiges Verlassen ist mit *Exit Function* möglich. Den Rückgabewert können Sie entweder mittels *Return* übergeben oder aber direkt in den Funktionsnamen schreiben.

SYNTAX:
```
[Private|Public] Function Name ([Parameterliste]) As Datentyp
   ' Code definieren
   ' ...
   [Exit Function]       ' Vorzeitiges Verlassen
   ' ...
   Return Ausdruck       ' Rückgabe des Funktionswertes
End Function
```

Die *Parameterliste* ist eine durch Kommas separierte Liste mit den einzelnen Parametern (Argumenten), von denen jeder wie folgt notiert wird:

SYNTAX:
```
[ByVal|ByRef] Parameter [As Datentyp]
```

Zur Bedeutung der Übergabeart *ByVal* bzw. *ByRef* siehe Abschnitt 2.8.3.

BEISPIEL 2.92: Funktion zur Berechnung des Kugelgewichts. Übergabeparameter sind der Radius und das spezifische Gewicht.

VB
```
Private Function Kugel(ra As Double, sg As Double) As Double
    Dim vol As Double = 4 / 3 * Math.PI * Math.Pow(ra, 3)
    Return sg * vol
End Function
```

Beim Aufruf der Funktion müssen Reihenfolge und Datentyp der Parameter exakt übereinstimmen. Im aufrufenden Code muss der Rückgabewert einer Variablen gleichen Datentyps wie die Funktion zugewiesen werden.

BEISPIEL 2.93: *(Fortsetzung)* Obige Funktion wird mit einer eisernen Kugel von 20 cm Durchmesser getestet.

VB
```
Dim g As Double = Kugel(10, 7.87)             ' Radius ist 10cm, spez. Gewicht = 7,87 gr/cm3
MessageBox.Show(g.ToString("#,#0.000 Gramm"))  ' das Gewicht der Kugel ist 32.965,779 Gramm
```

Sub

Prozeduren werden mit dem *Sub*-Schlüsselwort (kommt von *Sub*routine) deklariert und haben keinen Rückgabewert. Ansonsten gelten die gleichen syntaktischen Regeln wie bei Funktionen.

SYNTAX:
```
[Private|Public] Sub Name ([Parameterliste])
    ' Code definieren
    ' ...
    [Exit Sub]       ' Vorzeitiges Verlassen
    ' ...
End Sub
```

Da eine Prozedur keinen Rückgabewert hat, arbeitet sie oft mit globalen Variablen zusammen.

BEISPIEL 2.94: Anwenden einer Prozedur zum Berechnen des Kugelgewichts

VB
```
Private gew As Double            ' globale Variable

Private Sub Kugel(ra As Double, sg As Double)
  Dim vol As Double = 4 / 3 * Math.PI * Math.Pow(ra, 3)
```

Die globale Variable *gew* wird mit dem Ergebnis gefüttert:

```
  gew = sg * vol
End Sub
```

Beim Aufruf einer Prozedur können Sie, müssen aber nicht, das Schlüsselwort *Call* voranstellen. Wir bevorzugen diese Variante nur gelegentlich in der Lernphase, da so keinerlei Verwechslungsgefahr mit dem Aufruf einer Funktion besteht.

BEISPIEL 2.95: *(Fortsetzung)* Unsere Prozedur wird nun ebenfalls mit der legendären Eisenkugel getestet.

VB
```
Call Kugel(10, 7.87)
MessageBox.Show(gew.ToString("#,#0.000 Gramm"))   '  zeigt "32.965,779 Gramm"
```

2.8.2 Parameterübergabe allgemein

In unseren bisherigen *Kugel*-Beispielen hatten wir beim Aufruf der Funktion bzw. der Prozedur feste Zahlenwerte (Literale) übergeben. In der Praxis treten an diese Stelle aber meistens Variablen oder sogar Ausdrücke.

Signatur der Parameterliste

Normalerweise muss beim Aufruf einer Funktion/Prozedur die so genannte *Signatur* der Parameterliste beachtet werden, d.h., die Reihenfolge der Parameter und ihr Datentyp müssen zur deklarierten Parameterliste passen (Ausnahme: benannte Parameter, siehe unten).

BEISPIEL 2.96: *(Fortsetzung)* Die in 2.8.1 deklarierte *Kugel*-Funktion wird aufgerufen.

VB
```
Dim radius As Double = 5
Dim spezGew As Double = 7.87
Dim gew As Double = Kugel(2 * radius, spezGew)
```

Benannte Parameter

Die Verwendung *benannter Parameter* macht es möglich, dass die in der Signatur festgelegte Reihenfolge bei der Parameterübergabe nicht unbedingt eingehalten werden muss. Allerdings sind dann der deklarierte Name des Parameters und die Zeichenfolge ":=" voranzustellen.

BEISPIEL 2.97: Das Vorgängerbeispiel wird mit benannten Parametern ausgeführt, die Reihenfolge der übergebenen Werte ist bewusst vertauscht.

VB
```
Dim radius As Double = 5
Dim spezGew As Double = 7.87
Dim gew As Double = Kugel(sg := spezGew, ra := 2 * radius)
```

HINWEIS: Die Verwendung benannter Parameter vereinfacht die Programmierung besonders dann, wenn Sie Funktionen/Prozeduren aufrufen, die über viele optionale Parameter verfügen (siehe Abschnitt 2.8.4).

2.8.3 Übergabe mit ByVal und ByRef

Jedem Mitglied der Parameterliste einer Funktion/Prozedur ist entweder *ByVal* oder *ByRef* vorangestellt[1]. Dadurch wird festgelegt, ob die Parameterübergabe von Wertetypen "als Wert" (*ByVal*) oder "als Referenz" (*ByRef*) erfolgen soll. Was bedeuten diese Übergabearten?

ByVal

Mit der standardmäßigen *ByVal*-Übergabe wird der Wert des übergebenen Parameters (ein Wertetyp) in eine vom Compiler erzeugte lokale Variable des Funktions- bzw. Prozedurkörpers kopiert. Die aufrufende Variable und die lokale Variable im Inneren der Funktion/Prozedur haben danach nichts mehr miteinander zu tun, eine Veränderung des Wertes der lokalen Variablen bleibt ohne Rückwirkung auf den Übergabeparameter, so dass ein "Zurückgeben" von Ergebnissen – wie im obigen Beispiel gezeigt – nicht möglich ist.

Wenn Sie hingegen *ByRef* übergeben, so verfügt die aufgerufene Funktion/Prozedur nur scheinbar über eine eigene lokale Variable. In Wirklichkeit steht der ursprüngliche Parameter weiter zur Verfügung, eine "Entkopplung" hat nicht stattgefunden.

HINWEIS: Im Interesse der Fehlersicherheit Ihrer Programme sollten Sie Parameter grundsätzlich *ByVal* übergeben (das gilt auch für Referenztypen wie Arrays, Strings und sonstige Objekte). *ByRef* sollten Sie nur dann verwenden, wenn Sie es tatsächlich benötigen und Werte nach außen hin verändern wollen (siehe Praxisbeispiel 2.9.4).

ByRef

Wenn Sie eine Variable *ByRef* übergeben, so wird nicht der Wert, sondern der Zeiger auf die Speicherplatzadresse der Variablen übermittelt. Der aufgerufenen Funktion/Prozedur ist es dadurch möglich, am Wert der Variablen "herumzudoktern". Auf diese Weise können auch Prozeduren ihre Ergebnisse ohne Verwendung globaler Variablen (siehe obiges Beispiel) zurückliefern.

BEISPIEL 2.98: ***By Reference***

VB

In obiger Prozedur zur Kugelgewichtsbestimmung wird die Übergabeart des ersten Parameters in *ByRef* geändert. Dadurch wird es möglich, über diesen Parameter auch das Berechnungsergebnis zu übertragen.

```
Private Sub Kugel(ByRef ra As Double, sg As Double)
  Dim vol As Double = 4 / 3 * Math.PI * Math.Pow(ra, 3)
  ra = sg * vol                                    ' Radius wird zu Gewicht!
End Sub
```

Zum Testen der Prozedur: Wir haben den ersten Parameter bewusst nicht mit *radius*, sondern mit *x* bezeichnet, da er seine Bedeutung wechselt und nach dem Aufruf der *Kugel*-Prozedur eigentlich *gewicht* heißen müsste.

[1] *ByVal* kann weggelassen werden, da es Standard ist.

BEISPIEL 2.98: *By Reference*

```
Dim x As Double = 10
Dim spezGew As Double = 7.87

Kugel(x, spezGew)
MessageBox.Show(x.ToString("#,#0.000 Gramm"))     ' zeigt "32.965,779 Gramm"
```

2.8.4 Optionale Parameter

Wenn Sie Parameter mit dem Schlüsselwort *Optional* deklarieren, können Sie diese beim Aufruf auch weglassen, sie werden dann durch Standardwerte aufgefüllt.

BEISPIEL 2.99: Unsere *Kugel*-Funktion wird modifiziert, so dass der zweite Parameter optional ist und den Standardwert 7.87 (spezifisches Gewicht von Eisen) erhält.

VB
```
Public Function Kugel(ra As Double, Optional sg As Double = 7.87) As Double
    Dim vol As Double = 4 / 3 * Math.PI * Math.Pow(ra, 3)
    Return sg * vol
End Function
```

Für den Aufruf der Funktion gibt es nun zwei Versionen:

```
Dim g As Double = Kugel(10)                       ' Berechnung mit Standardwert (Eisen)
MessageBox.Show(g.ToString("#,#0.000 Gramm"))     ' zeigt "32.965,779 Gramm"
```

oder

```
Dim g As Double = Kugel(10, 11.34)                ' Berechnung für 11.34gr/cm3 (Blei)
MessageBox.Show(g.ToString("#,#0.000 Gramm"))     ' zeigt "47.500,881 Gramm"
```

HINWEIS: Einem optionalen Parameter müssen Sie **immer** einen Standardwert zuweisen!

Eine modernere Alternative zu optionalen Parametern ist das Überladen von Funktionen/Prozeduren (siehe folgender Abschnitt).

2.8.5 Überladene Funktionen/Prozeduren

Wenn mehrere gleichnamige Funktionen/Prozeduren ohne Namenskonflikt friedlich nebeneinander existieren, so spricht man von *überladenen* Funktionen. Die Unterscheidung zwischen ihnen wird vom Compiler anhand der Parameterliste, sprich Signatur, getroffen. Unter Signatur verstehen wir die Reihenfolge der übergebenen Parameter und ihr Datentyp.

HINWEIS: Überladene Funktionen/Prozeduren müssen eine unterschiedliche Signatur haben!

BEISPIEL 2.100: Die *Kugel*-Funktion steht in zwei überladenen Versionen zur Verfügung. Die erste Version verlangt als Parameter den Radius und das spezifische Gewicht.

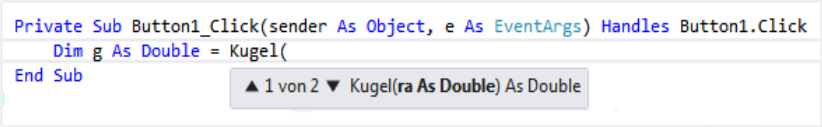
VB

```
Private Function Kugel(ra As Double, sg As Double) As Double
    Dim vol As Double = 4 / 3 * Math.PI * Math.Pow(ra, 3)
    Return sg * vol
End Function
```

Eine zweite Version soll nur das Volumen der Kugel berechnen.

```
Private Function Kugel(ra As Double) As Double
    Dim vol As Double = 4 / 3 * Math.PI * Math.Pow(ra, 3)
    Return vol
End Function
```

Wenn Sie jetzt in der IDE von Visual Studio die Funktion verwenden wollen, werden Ihnen automatisch beide überladenen Versionen angeboten:



Der Aufruf ist entweder so

```
Dim g As Double = Kugel(10, 7.87)       ' liefert  "32.965,779"
```

oder so möglich

```
Dim v As Double = Kugel(10)             ' liefert  "4.188,790"
```

Wie Sie bestimmt schon bei der Arbeit im Codefenster von Visual Studio bemerkt haben, werden für die meisten Objektmethoden mehr oder weniger viele Überladungen angeboten.

Die *Show*-Methode der *MessageBox* hat beispielsweise insgesamt 21 Überladungen!

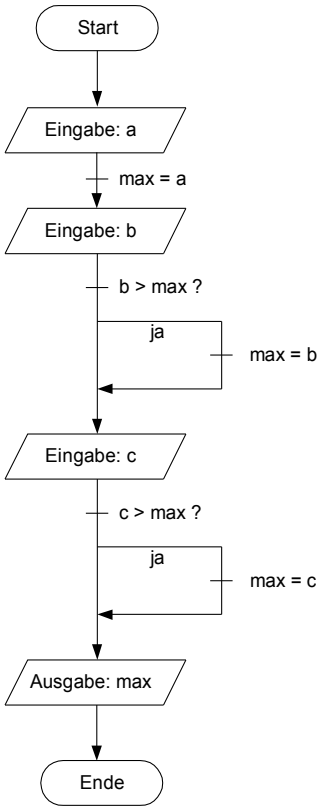
2.9 Praxisbeispiele

2.9.1 Vom PAP zum Konsolen-Programm

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 abgebildete Programmablaufplan (PAP) zeigt die Berechnungsvorschrift (Algorithmus).



Sie erkennen hier die typische EVA-Grundstruktur, bei der die Anweisungen in der Reihenfolge Eingabe, Verarbeitung, Ausgabe ausgeführt werden.

Programmierung

Beim Eintippen der Befehle (z.B. mit Notepad) gibt obiger PAP eine nützliche Orientierung:

```
Imports System
Module Maximum3
  Sub Main()
      Console.WriteLine("Maximum von drei Zahlen")  ' Überschrift
      Console.WriteLine()                           ' Leerzeile
      Dim a, b, c, max As Integer                   ' Variablendeklaration
      Console.WriteLine("Geben Sie die erste Zahl ein!")
      a = CInt(Console.ReadLine())                 ' Eingabe a
      max = a                                      '  Initialisieren von max
      Console.WriteLine("Geben Sie die zweite Zahl ein!")
```

```
      b = CInt(Console.ReadLine())                ' Eingabe b
      If b > max Then max = b                     ' Bedingung
      Console.WriteLine("Geben Sie die dritte Zahl ein!")
      c = CInt(Console.ReadLine())                ' Eingabe c
      If c > max Then max = c                     ' Bedingung
      Console.WriteLine("Das Maximum ist " + max.ToString)   ' Ergebnisausgabe
      Console.ReadLine()                         ' Programm wartet auf <Enter>, um zu beenden
   End Sub
End Module
```

Speichern Sie die Textdatei z.B. als *Maximum3.vb* im Projektverzeichnis ab.

Kompilieren

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

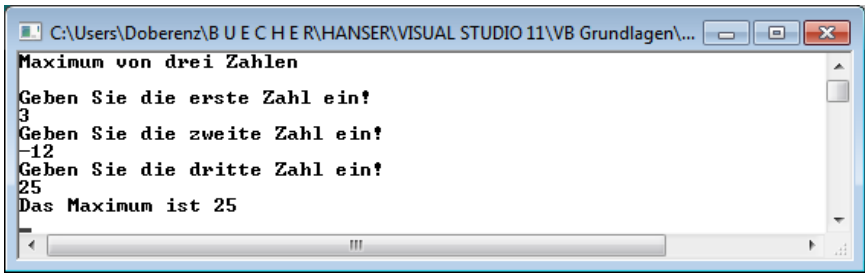
Geben Sie an der Kommandozeile ein:

```
vbc /t:exe Maximum3.vb
```

Haben Sie beim Eintippen des Quellcodes keine Fehler gemacht, so verläuft das Kompilieren anstandslos, im Projektverzeichnis finden Sie nun die Datei *Maximum3.exe* vor.

Test

Starten Sie *Maximum3.exe* durch Doppelklick!



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

2.9.2 Vom Konsolen- zum Windows-Programm

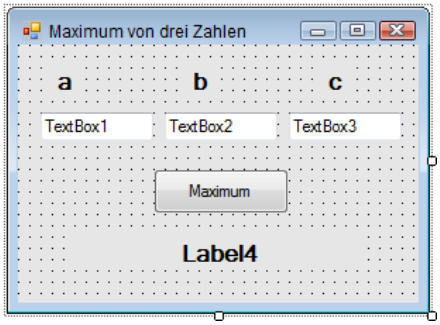
Eine Windows-Anwendung ist natürlich viel attraktiver als eine triste Konsolen-Applikation und schließlich wollen Sie ja zukünftig mit dem Komfort von Visual Studio anstatt mit einem simplen Texteditor arbeiten!

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 2012 und öffnen Sie ein neues Projekt (Projekttyp *Visual Basic*, 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 obiger Abbildung auf *Form1* und stellen auch hier bestimmte Eigenschaften (Beschriftungen) neu ein.

Quelltext

Durch einen Doppelklick auf *Button1* wird automatisch das Codefenster von *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 nie selbst eintippen, sondern immer durch die Visual Studio-Entwicklungsumgebung erzeugen lassen (siehe auch Einführungsbeispiele 1.7.1 und 1.7.2)!

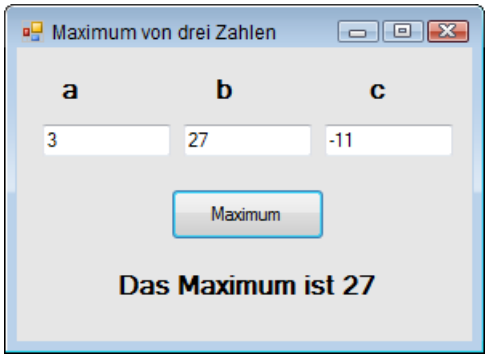
Füllen Sie den Körper des Eventhandlers mit den erforderlichen Anweisungen aus, so dass der komplette Eventhandler schließlich folgendermaßen aussieht:

```
Private Sub Button1_Click(sender As Object, e As EventArgs)  Handles Button1.Click
        Dim a, b, c, max As Integer                    ' Variablendeklaration
        a = CInt(TextBox1.Text)                        ' Eingabe a
        max = a                                        ' Initialisieren von max
        b = CInt(TextBox2.Text)                        ' Eingabe b
        If b > max Then max = b                        ' Bedingung
        c = CInt(TextBox3.Text)                        ' Eingabe c
        If c > max Then max = c                        ' Bedingung
        Label4.Text = "Das Maximum ist " + max.ToString    ' Ergebnisausgabe
End Sub
```

Beim Vergleich mit der Konsolenanwendung erkennen Sie, dass die Programmierung von Ein- und Ausgabe deutlich einfacher geworden sind!

Test

Mittels *F5*-Taste kompilieren und starten Sie das Programm:



Bemerkungen

Neben dem attraktiveren Outfit 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.

2.9.3 Schleifenanweisungen kennen lernen

Visual Basic ist an Schleifenanweisungen reich gesegnet. Dieses Beispiel demonstriert Ihnen die prinzipielle Anwendung jedes Schleifentyps (siehe 2.6.2). Dabei soll zehnmal untereinander der Text "Viele Wege führen nach Rom!" in einer *ListBox* auszugeben werden, wobei acht verschiedene Schleifenkonstruktionen zur Anwendung kommen.

Oberfläche

Wie die folgende Laufzeitansicht zeigt, brauchen Sie neben einer *ListBox* auch eine ganze Menge von *Button*s (für jeden Schleifentyp einen und je einen zum Löschen des *ListBox*-Inhalts und zum Beenden des Programms).

Quellcode

Wir beginnen mit der altbekannten *For...Next*-Schleife:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    Dim i As Integer
    For i = 1 To 10
```

```
            ListBox1.Items.Add(i.ToString & " Viele Wege führen nach Rom!")
        Next i
    End Sub
```

Do While...Loop-Schleife:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        Dim i As Integer = 1
        Do While i <= 10
            ListBox1.Items.Add(i.ToString & " Viele Wege führen nach Rom!")
            i += 1
        Loop
    End Sub
```

Do Until...Loop-Schleife:

```
    Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
        Dim i As Integer = 1
        Do Until i > 10
            ListBox1.Items.Add(i.ToString & " Viele Wege führen nach Rom!")
            i += 1
        Loop
    End Sub
```

Do...Loop While-Schleife:

```
    Private Sub Button4_Click(sender As Object, e As EventArgs) Handles Button4.Click
        Dim i As Integer = 1
        Do
            ListBox1.Items.Add(i.ToString & " Viele Wege führen nach Rom!")
            i += 1
        Loop While i <= 10
    End Sub
```

Do...Loop Until-Schleife:

```
    Private Sub Button5_Click(sender As Object, e As EventArgs) Handles Button5.Click
        Dim i As Integer = 1
        Do
           ListBox1.Items.Add(i.ToString & " Viele Wege führen nach Rom!")
           i += 1
        Loop Until i > 10
    End Sub
```

Do...Loop-Schleife mit *Exit Do*-Abbruch:

```
    Private Sub Button6_Click(sender As Object, e As EventArgs) Handles Button6.Click
        Dim i As Integer = 1
        Do
            ListBox1.Items.Add(i.ToString & " Viele Wege führen nach Rom!")
            If i = 10 Then Exit Do
            i += 1
        Loop
```

```
    End Sub
```

While...End While-Schleife:

```
    Private Sub Button7_Click(sender As Object,e As EventArgs) Handles Button7.Click
        Dim i As Integer = 1
        While i <= 10
            ListBox1.Items.Add(i.ToString & " Viele Wege führen nach Rom!")
          i += 1
        End While
    End Sub
```

Obwohl es sich hier um einen völlig anderen Schleifentyp handelt, soll vergleichsweise noch die Funktion der *For Each*-Schleife demonstriert werden:

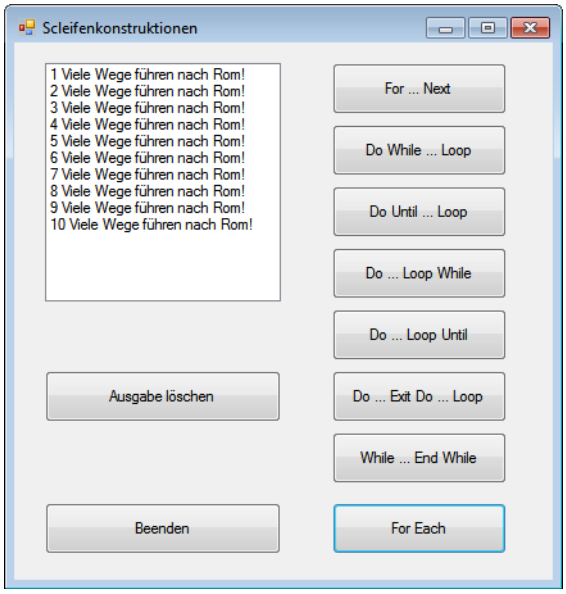
```
Private Sub Button8_Click(sender As Object, e As EventArgs) Handles Button8.Click
    Dim zahlen = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    For Each i As Integer In zahlen
        ListBox1.Items.Add(i.ToString & " Viele Wege führen nach Rom!")
    Next
End Sub
```

Eher nebensächlich das Löschen des Inhalts der *ListBox*:

```
Pivate Sub Button9_Click(sender As Object, e As EventArgs) Handles Button9.Click
    ListBox1.Items.Clear()
End Sub
```

Test

Alle acht Schleifenvarianten sollten ein absolut identisches Ergebnis erzeugen:



HINWEIS: Die *For Each*-Schleife ist mit der *For ... Next*-Schleife vergleichbar. Diesmal zählt sie allerdings keine Variable hoch, sondern die Schleife greift auf alle Elemente einer Auflistung zu. Es ist allerdings nur Lesezugriff erlaubt.

2.9.4 Methoden überladen

In VB spricht man im klassischen Sinn noch von Funktionen (*Function*s) und Prozeduren (*Sub*s), je nachdem ob es sich um Methoden mit und ohne Rückgabewert handelt. Beschränkt man sich auf einen privaten Gültigkeitsbereich (Aufruf nur innerhalb einer Klasse), so spielen diese quasi die Rolle von "Unterprogrammen", sind also strenggenommen keine Methoden im Sinne der OOP. Trotzdem können auch sie 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 (*Structure*), *If ... ElseIf*-Statements, *RadioButton*-Auswahl ... 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 Laufzeitabbildung).

Quellcode

```
Public Class Form1
```

Die *Kugel*-Struktur kapselt Radius und spezifisches Gewicht einer Kugel:

```
    Private Structure Kugel
        Dim radius, sg As Double
    End Structure
```

Eine globale Variable dient lediglich als Rückgabewert für Variante 2:

```
    Private gew As Double
```

Die folgende Hilfsfunktion liest die Kugelwerte aus der Eingabemaske in eine Strukturvariable vom oben definierten *Kugel*-Typ:

```
    Private Function getKugel() As Kugel
        Dim kug As Kugel
```

Den Kugelradius zuweisen:

```
        kug.radius = Convert.ToDouble(TextBox1.Text) / 2
```

Das spezifische Gewicht zuweisen:

```
        If RadioButton1.Checked Then
            kug.sg = 1.4 ' Holz
        ElseIf RadioButton2.Checked Then
            kug.sg = 2.7      ' Alu
        ElseIf RadioButton3.Checked Then
            kug.sg = 3D       ' Glas
        ElseIf RadioButton4.Checked Then
            kug.sg = 7.87D    ' Eisen
        Else
            kug.sg = 11.3     ' Blei
        End If
        Return kug
    End Function
```

Variante 1 zeigt als erste Überladung eine normale Funktion zur Bestimmung des Kugelgewichts:

```
    Private Function KugelGewicht(ra As Double, sg As Double) As Double
        Dim vol As Double = 4 / 3 * Math.PI * Math.Pow(ra, 3)
        Return sg * vol     ' Rückgabe des Gewichts als Funktionswert
    End Function
```

Der Aufruf:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim kug As Kugel = getKugel()
        Dim r As Double = kug.radius, sg = kug.sg      ' Radius und spez. Gewicht
        Dim gew As Double = KugelGewicht(r, sg)        ' Aufruf einer Funktion
        Label3.Text = gew.ToString("#,#0.000 Gramm")   ' lokale Variable gew
    End Sub
```

Variante ist eine Prozedur (Sub), die das Ergebnis der globalen Variablen *gew* direkt zuweist:

```
    Private Sub KugelGewicht(kug As Kugel)
        Dim vol As Double = 4 / 3 * Math.PI * Math.Pow(kug.radius, 3)
        gew = kug.sg * vol
    End Sub
```

Der Aufruf:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        Dim kug As Kugel = getKugel()
        KugelGewicht(kug)
        Label4.Text = gew.ToString("#,#0.000 Gramm")
    End Sub
```

Variante 3 ist ebenfalls eine Prozedur (Sub), hier aber wird das Gewicht über den Parameter *g* per Referenz zurückgegeben:

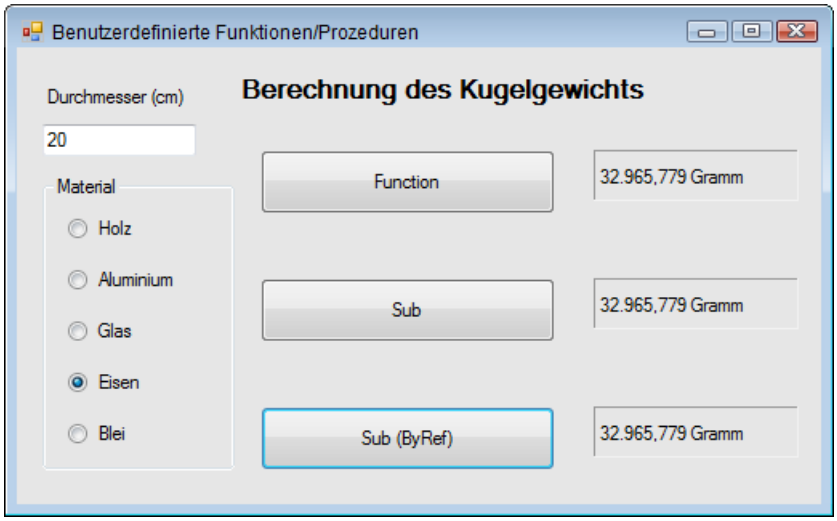
```
    Private Sub KugelGewicht(kug As Kugel, ByRef g As Double)
        Dim vol As Double = 4 / 3 * Math.PI * Math.Pow(kug.radius, 3)
        g = kug.sg * vol
    End Sub
```

Aufruf Variante 3:

```
    Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
        Dim kug As Kugel = getKugel()
        Dim gew As Double
        KugelGewicht(kug, gew)
        Label5.Text = gew.ToString("#,#0.000 Gramm")
    End Sub
End Class
```

Test

Geben Sie das Material und den Durchmesser der Kugel ein und lassen Sie sich das Gewicht anzeigen. Alle drei Überladungen liefern identische Ergebnisse[1]:



2.9.5 Eine Iterationsschleife verstehen

Eigentlich ist ein Computer ja "dumm" und erscheint nur dadurch "intelligent", dass er primitive Rechenoperationen in hoher Geschwindigkeit erledigt. Der pfiffige Programmierer kann dies ausnutzen, indem er dem Computer Aufgaben stellt, die sich nicht sofort, sondern nur durch schrittweises Ausprobieren lösen lassen. Typisch für diese Sorte von Aufgaben ist eine so genannte *Iterationsschleife*, die mit einer "über den Daumen gepeilten" *Startnäherung* beginnt und an deren Ende

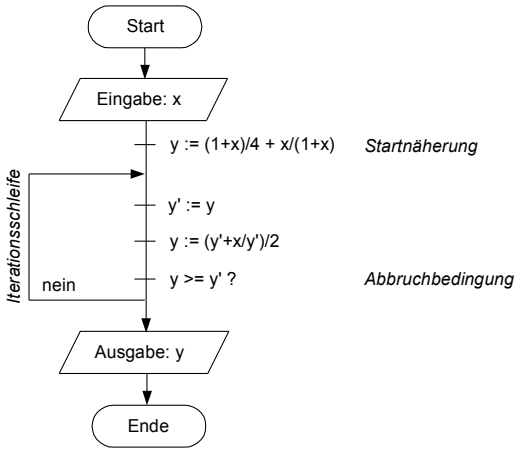
[1] Kaum zu glauben aber wahr: Eine Eisenkugel mit nur 20 cm Durchmesser wiegt stolze 33 Kilogramm!

eine *Abbruchbedingung* überprüft wird. Wie viele Male die Schleife durchlaufen wird, kann nicht exakt vorausbestimmt werden.

Ohne auf die mathematischen Grundlagen näher einzugehen, wollen wir in diesem Rezept eine Iterationsschleife für das Ziehen der Quadratwurzel demonstrieren. Wir verzichten also auf die *Sqrt*-Funktion, wie sie standardmäßig von der *Math*-Klasse bereitgestellt wird, und programmieren stattdessen eine eigene Lösung.

Programmablaufplan

Obwohl in der objekt- und ereignisorientierten Programmierung der PAP völlig aus der Mode gekommen ist, eignet er sich nach wie vor gut zur Veranschaulichung von Iterationszyklen.



Oberfläche

Für die Eingabe findet eine *TextBox*- und für die Ausgabe eine *Label*-Komponente Verwendung. Die Iterationsschleife starten wir mit einem *Button*. Eine *ListBox* ist nicht unbedingt erforderlich, aber wir sind ja neugierig und wollen auch die Zwischenergebnisse betrachten (siehe Laufzeitabbildung).

Quellcode

Die programmtechnische Umsetzung des obigen PAP führt (in Verbindung mit dem Code für die Ein- und Ausgabe und für die Anzeige von Zwischenergebnissen) zu folgender Lösung:

```
Public Class Form1
```

Die Schaltfläche *Quadratwurzel>>*:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

Einige Variablen deklarieren und den Eingabewert explizit in einen *Double*-Wert konvertieren:

```
        Dim x, y, ya As Double
        x = Convert.ToDouble(TextBox1.Text)
```

Die Quadratwurzel darf nur aus positiven Zahlen gezogen werden:

```
        If x > 0 Then
```

Inhalt der *ListBox* löschen:

```
            ListBox1.Items.Clear()
```

Die Startnäherung:

```
            y = (1 + x) / 4 + x / (1 + x)
```

Wie geschaffen für unsere Iterationsschleife ist die *Do...Loop Until*-Anweisung:

```
            Do
                ya = y
                y = (ya + x / ya) / 2
```

Das Zwischenergebnis anzeigen:

```
                ListBox1.Items.Add(y.ToString)
```

Die Abbruchbedingung prüfen:

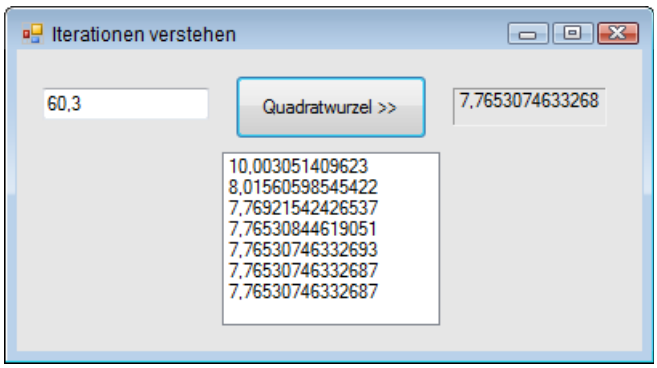
```
            Loop Until y >= ya
```

Das Endergebnis anzeigen:

```
            Label1.Text = y.ToString
        Else
            Label1.Text = "Bitte geben Sie einen positiven Wert ein!"
        End If
    End Sub
End Class
```

Test

Die Zwischenergebnisse nähern sich schrittweise der endgültigen Lösung. Sie werden feststellen, dass circa fünf bis sieben Iterationen notwendig sind, um die Abbruchbedingung zu erfüllen, d.h., die Quadratwurzel in einer für *Double*-Zahlen ausreichenden Genauigkeit zu ermitteln:



Ergänzung

Der fortgeschrittene Programmierer wird obigen Code – besonders im Hinblick auf seine Wiederverwendbarkeit – in eine Funktion *qWurzel* verpacken. Da die Testphase abgeschlossen ist, kann auch auf die Anzeige der Zwischenergebnisse in der *ListBox* verzichtet werden:

```
Public Function qWurzel(x As Double) As Double
    Dim y, ya As Double
    y = (1 + x) / 4 + x / (1 + x)   ' Startnäherung
    Do
        ya = y
        y = (ya + x / ya) / 2
    Loop Until y >= ya              ' Abbruchbedingung
    Return y
End Function
```

Der Aufruf gestaltet sich nun wesentlich übersichtlicher:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    Dim x As Double = Convert.ToDouble(TextBox1.Text)
    If x > 0 Then
        Label1.Text = qWurzel(x).ToString
    Else
        Label1.Text = "Bitte geben Sie einen positiven Wert ein!"
    End If
End Sub
```

Kapitel 3

OOP-Konzepte

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.

Visual Basic erlaubt es Ihnen aber, 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 (Event-Handler) 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 Visual Basic im Überblick.

3.1 Strukturierter versus objektorientierter Entwurf

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.

3.1.1 Was bedeutet 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 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.

Dass VB 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 VB 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. anstatt mit Methoden mit Funktionen und Prozeduren und anstatt mit Klassen mit strukturierten Datentypen (*Structure*) gearbeitet. Tatsächlich können Sie aber mit OOP alles machen, was auch die strukturierte Programmierung erlaubt.

Anstatt 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.

3.1.2 Was heißt objektorientierte Programmierung?

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.

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.

Das Konzept der objektorientierten Programmierung (OOP) überwindet den prozeduralen Ansatz der klassischen strukturellen Programmierung zugunsten einer realitätsnahen Modellierung.

3.2 Grundbegriffe der OOP

Bevor wir uns den Details zuwenden, sollen die wichtigsten Begriffe der objektorientierten Programmierung zunächst allgemein, d.h. ohne Bezug auf eine konkrete Programmiersprache, erörtert werden.

3.2.1 Objekt, Klasse, Instanz

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.

Eine *Klasse*[1] 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: In VB haben Sie grundsätzlich die Möglichkeit, zwischen Klassen und Strukturen zu wählen. Letztere wurden bereits im Sprachkapitel (Abschnitt 2.7.2) 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 beschäftigen.

[1] Oft wird anstatt "Klasse" mit völlig gleichwertiger Bedeutung auch der Begriff "Objekttyp" (oder auch "Typ") verwendet.

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: Objekt, Klasse, 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.

3.2.2 Kapselung und Wiederverwendbarkeit

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.

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.

3.2.3 Vererbung und Polymorphie

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 (auch oft als Superklasse bezeichnet) 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".

Visual Basic unterstützt das Überschreiben (*Overriding*) von Methoden[1] der Oberklasse mit alternativen Methoden der Unterklasse.

[1] Nicht zu verwechseln mit dem Überladen (Overloading) von Methoden.

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: Vererbung

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.

3.2.4 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.
Friend	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.

Klassenmitglieder

Die folgende Tabelle zeigt die Zugriffsmöglichkeiten auf die Klassenmitglieder (Member).

Modifizierer	Sichtbarkeit
Public	Unbeschränkt.
Friend	Innerhalb der aktuellen Anwendung wie *Public*, sonst wie *Private*.
Protected	Wie *Private*, Mitglieder dürfen aber auch in abgeleiteten Klassen verwendet werden.
Protected Friend	Innerhalb der aktuellen Anwendung wie *Protected*, sonst wie *Private*.
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 3.9 (Vererbung und Polymorphie).

3.2.5 Allgemeiner Aufbau einer Klasse

Bevor der Einsteiger seine erste Klasse schreibt, sollte er sich zunächst im einführenden Sprachkapitel mit den Strukturen (siehe Abschnitt 2.7.2) anfreunden, die in Aufbau und Anwendung starke Ähnlichkeiten zu Klassen aufweisen (der wesentliche Unterschied ist, dass Strukturen Wertetypen, Klassen hingegen Referenztypen sind). Auch im Aufbau von Funktionen bzw. Methoden sollte sich der Lernende auskennen (Abschnitt 2.8).

Im Unterschied zu einer Struktur (Schlüsselwort *Structure*) wird eine Klasse mit dem Schlüsselwort *Class* deklariert. Hier die (stark vereinfachte) Syntax:

SYNTAX:
```
Modifizierer Class Bezeichner
        ' ... Felder
        ' ... Konstruktoren
        ' ... Eigenschaften
        ' ... Methoden
        ' ... Ereignisse
End Class
```

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.

VB
```
Public Class CKunde
   Private _anrede As String          '    Feld
   Private _name As String            '    dto.

   Public Sub New(anr As String, nam As String)    ' Konstruktor
      _anrede = anr
```

BEISPIEL 3.3: Eine einfache Klasse *CKunde* wird deklariert und implementiert.

```
      _name = nam
   End Sub
   Public Property name() As String                ' Eigenschaft
      Get
         Return _name
      End Get
      Set(value As String)
         _name = value
      End Set
   End Property

   Public Function adresse() As String             ' Methode
       Dim s As String = _anrede & " " & _name
       Return s
   End Function
End Class
```

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.

3.3 Ein Objekt erzeugen

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 *Nothing* 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 *Nothing*)
- 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.

3.3.1 Referenzieren und Instanziieren

Es stehen zwei Varianten zur Verfügung.

Variante 1 erfordert zwei Schritte:

SYNTAX: *Modifizierer myObject* As *Klasse*
myObjekt = New *Klasse(Parameter)*

BEISPIEL 3.4: Ein Objekt *kunde1* wird referenziert und erzeugt.

VB
```
Private kunde1 As CKunde                    ' 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: *Modifizierer myObject* = New *Klasse()*

BEISPIEL 3.5: Das Äquivalent zum Vorgängerbeispiel.

VB
```
Private kunde1 As 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 (*Imports*-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].

BEISPIEL 3.6: Eine mögliche Fehlerbehandlung

VB
```
Private kunde1 As CKunde
Try
   kunde1 = New CKunde()
Catch e As Exception
   MessageBox.Show(ex.Message)
End Try
```

[1] Aus Platz- und Bequemlichkeitsgründen halten sich auch die Autoren nicht immer an diese Empfehlung.

3.3.2 Klassische Initialisierung

Anstatt die Anfangswerte einzeln zuzuweisen, können Sie diese zusammen mit einem Konstruktor übergeben. Zunächst ein Beispiel ohne eigenen Konstruktor, wobei der parameterlose Standardkonstruktor zum Einsatz kommt.

BEISPIEL 3.7: Das Objekt *kunde1* wird erzeugt (Standardkonstruktor), zwei Eigenschaften werden einzeln zugewiesen.

VB
```
Dim kunde1 As New CKunde()
kunde1.anrede = "Frau"
kunde1.name = "Müller"
```

BEISPIEL 3.8: Das Objekt *kunde1* wird erzeugt und mit einem Konstruktor initialisiert.

VB
```
Dim kunde1 As New CKunde("Frau", "Müller")
```

HINWEIS: Weitere Einzelheiten entnehmen Sie dem Abschnitt 3.8.1.

3.3.3 Objekt-Initialisierer

Man kann ein Objekt auch dann erzeugen und seine Eigenschaften (keine privaten Felder!) initialisieren, wenn es dazu keinen Konstruktor gibt.

BEISPIEL 3.9: Das Vorgängerbeispiel mit Objekt-Initialisierer

VB
```
Private kunde1 As New CKunde With {.anrede = "Frau", .name = "Müller"}
```

HINWEIS: Mehr zu Objekt-Initialisierern siehe Abschnitt 3.8.2!

3.3.4 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.

VB
```
kunde1.Guthaben = 10
Label1.Text = kunde1.adresse()
```

3.3.5 Zerstören des Objekts

Wenn Sie das Objekt nicht mehr brauchen, können Sie die Objektvariable auf *Nothing* setzen. Vorher können Sie (müssen aber nicht) die *Dispose*-Methode des Objekts aufrufen, vorausgesetzt, Sie haben Sie auch implementiert.

BEISPIEL 3.11: Der *kunde1* wird entfernt.

VB
```
kunde1.Dispose()
kunde1 = Nothing
```

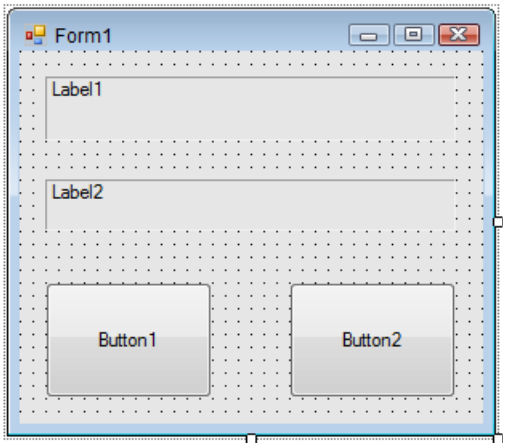
HINWEIS: Das Objekt wird allerdings erst dann zerstört, wenn der Garbage Collector festgestellt hat, dass es nicht länger benötigt wird.

3.4 OOP-Einführungsbeispiel

Raus aus dem muffigen Hörsaal, lassen Sie uns endlich einmal selbst eine einfache Klasse erstellen und beschnuppern!

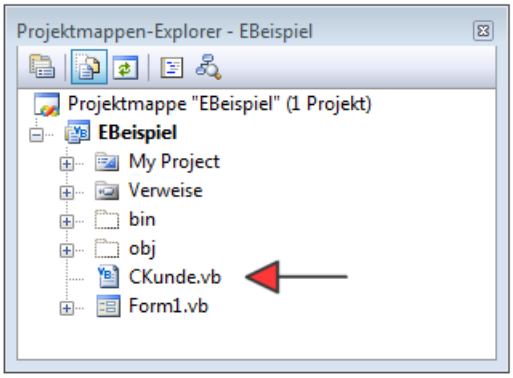
3.4.1 Vorbereitungen

- 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.vb* ein und klicken "Hinzufügen".

Der Projektmappen-Explorer zeigt jetzt die neue Klasse:



HINWEIS: 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.vb*) hinzufügen. Das Verwenden eigener Klassenmodule (idealerweise eins pro Klasse) steigert aber die Übersichtlichkeit des Programmcodes und erleichtert dessen Wiederverwendbarkeit.

3.4.2 Klasse definieren

Tragen Sie in den Klassenkörper die Implementierung der Klasse ein, sodass der komplette Code der Klasse schließlich folgendermaßen ausschaut:

```
Public Class CKunde
    Public Anrede As String             ' einfache Eigenschaften
    Public Name As String               '      dto.
    Public PLZ As Integer               '      dto.
    Public Ort As String                '      dto.
    Public Stammkunde As Boolean        '      dto.
    Public Guthaben As Decimal          '      dto.

    Public Function getAdresse() As String              ' erste Methode
        Dim s As String = Anrede & " " & Name & vbCrLf & PLZ.ToString & " " & Ort
        Return s
    End Function

    Public Sub addGuthaben(betrag As Decimal)     ' zweite Methode
        If Stammkunde Then Guthaben += betrag
    End Sub
End Class
```

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* (eine Funktion und eine Prozedur). Die Funktion *getAdresse()* liefert als Rückgabewert die komplette Anschrift des Kunden.
- Die Prozedur (*Sub*) *addGuthaben()* hingegen liefert keinen Wert zurück, sie erhöht den Wert des *Guthaben*-Felds bei jedem Aufruf um einen bestimmten Betrag.

3.4.3 Objekt erzeugen und initialisieren

Wechseln Sie nun in das Code-Fenster von *Form1*.

Zu Beginn deklarieren Sie eine Objektvariable *kunde1*:

```
Private kunde1 As CKunde            ' Objekt referenzieren
```

Dem linken Button geben Sie die Beschriftung "Objekt erzeugen und initialisieren" und belegen sein *Click*-Ereignis wie folgt:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        kunde1 = New CKunde()          ' Objekt erzeugen (instanziieren)
        With Kunde1                    ' Objekt initialisieren:
            .Anrede = "Herr"
            .Name = "Müller"
            .PLZ = 12345
            .Ort = "Berlin"
            .Stammkunde = True
        End With
    End Sub
```

3.4.4 Objekt verwenden

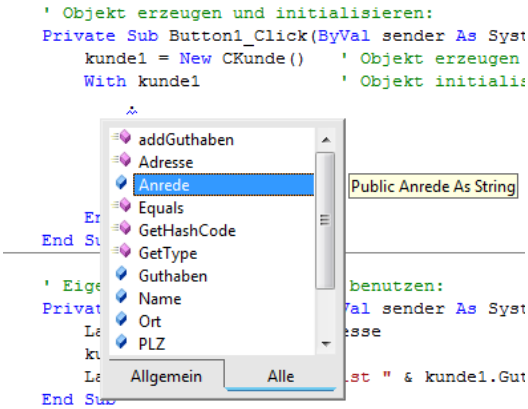
Hinterlegen Sie nun den rechten Button mit der Beschriftung "Eigenschaften und Methoden verwenden" wie folgt:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        Label1.Text = Kunde1.getAdresse    ' erste Methode aufrufen
        Kunde1.addGuthaben(50D)            ' zweite Methode aufrufen
        Label2.Text = "Guthaben ist " & Kunde1.Guthaben.ToString("C")    ' Eigenschaft lesen
    End Sub
```

3.4.5 IntelliSense – die hilfreiche Fee

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 doppelt auf den gewünschten Eintrag klicken.

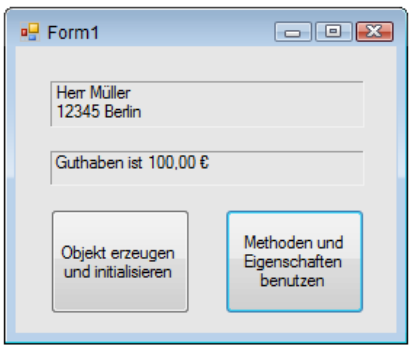


Falls das gewünschte Klassenmitglied nicht erscheint, müssen Sie sofort stutzig werden und es keinesfalls mit dem gewaltsamen Eintippen des Namens versuchen, denn dann gibt es wahrscheinlich einen Fehler beim Kompilieren. Überprüfen Sie stattdessen lieber nochmals die Klassendeklaration, z.B. ob vielleicht nicht doch der *Public*-Modifizierer vergessen wurde.

3.4.6 Objekt testen

Nun ist es endlich so weit, dass Sie Ihr erstes eigenes VB-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.



3.4.7 Warum unsere Klasse noch nicht optimal ist

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 Visual Basic 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.
- Weiterhin war das Initialisieren der Eigenschaften über mehrere Codezeilen ziemlich mühselig (von einem hilfreichen Konstruktor haben wir noch keinerlei Gebrauch gemacht).
- 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, können und wollen wir aber nicht weiter aufschieben und es gleich im folgenden Abschnitt behandeln.

3.5 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.

3.5.1 Eigenschaften 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 *Get*- und für den Schreibzugriff eine *Set*-Zugriffsmethode. Während die *Get*-Methode ihren Rückgabewert über *Return* liefert, erhält die *Set*-Methode den zu schreibenden Wert über den Parameter *value*.

```
SYNTAX:  [Public|Friend|Protected] Property Eigenschaftsname As Type
                Get
                     ' hier Lesezugriff (Wert=priv.Felder) implementieren
                     Return Wert
                End Get
                Set(value As Type)
                      ' hier Schreibzugriff (priv.Felder=value) implementieren
                 End Set
          End Property
```

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.

```
Public Class CKunde
    Private _anrede As String        '    privates Feld
    Private _name As String          '      dto.
    ...
```

Der Schreibzugriff auf die Eigenschaft *Anrede* soll so kontrolliert werden, dass nur die Werte "Herr" oder "Frau" zulässig sind. Geben Sie die erste Zeile der *Property*-Deklaration ein, so generiert Visual Studio automatisch den kompletten Rahmencode:

```
Public Property Anrede As String
    Get

    End Get
    Set(value As String)

    End Set
End Property
```

Sie brauchen dann nur noch den Lese- und den Schreibzugriff zu implementieren, sodass die komplette Eigenschaftsdefinition schließlich folgendermaßen aussieht:

```
    Public Property Anrede() As String
        Get
            Return _anrede
        End Get
        Set(value As String)
            If (value = "Herr") Or (value = "Frau") Then
                _anrede = value
            Else
```

```
            MessageBox.Show("Die Anrede '" & value & "' ist nicht zulässig!",
                                              "Fehler bei der Eingabe!")
        End If
    End Set
End Property
```

Beim Implementieren der Eigenschaft *Name* machen wir es uns etwas einfacher. Hier soll uns die einfache Kapselung genügen (es gibt also keinerlei Zugriffskontrolle):

```
Public Property Name() As String
    Get
        Return _name
    End Get
    Set(value As String)
        _name = value
    End Set
End Property
```

Zugriff

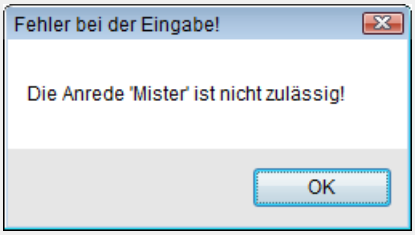
Wenn Sie ein Objekt verwenden, merken Sie auf Anhieb natürlich nicht, ob es noch über "einfache" oder schon über "richtige" Eigenschaften verfügt, es sei denn, die in die *Get*- bzw. *Set*-Methoden eingebauten Zugriffsbeschränkungen werden verletzt und Sie erhalten entsprechende Fehlermeldungen.

BEISPIEL 3.12: Sie wollen die Anrede "Mister" zuweisen, was zu einem Laufzeitfehler führt.

VB

```
Dim kunde1 As 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.
- 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 bei der *Name*-Eigenschaft nur wegen des Lerneffekts betrieben.

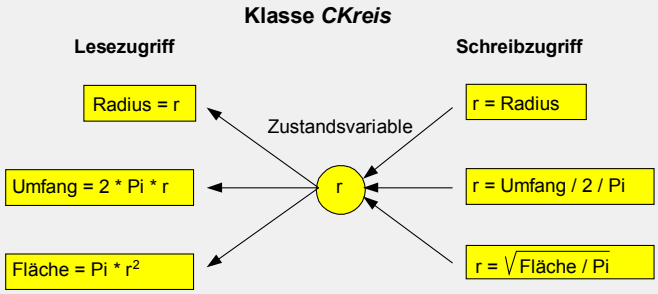
3.5.2 Eigenschaften mit Zugriffsmethoden kapseln

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.

BEISPIEL 3.13: Eigenschaften mit Zugriffsmethoden kapseln

VB

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 r As Double                 ' die einzige Zustandsvariable
```

Die Eigenschaft *Radius*:

```
    Public Property Radius() As String
        Get
            Return r.ToString("#,#0.00")
        End Get
        Set(value As String)
            If value <> String.Empty Then
                r = CDbl(value)
            Else
                r = 0
            End If
        End Set
    End Property
```

Die Eigenschaft *Umfang*:

```
    Public Property Umfang() As String
        Get
            Return (2 * Math.PI * r).ToString("#,#0.00")
        End Get
        Set(value As String)
```

BEISPIEL 3.13: Eigenschaften mit Zugriffsmethoden kapseln

```
            If value <> String.Empty Then
                r = CDbl(value) / 2 / Math.PI
            Else
                r = 0
            End If
        End Set
    End Property
```

Die Eigenschaft *Fläche*:

```
    Public Property Fläche() As String
        Get
            Return (Math.PI * Math.Pow(r, 2)).ToString("#,#0.00")
        End Get
        Set(value As String)
            If value <> String.Empty Then
                r = Math.Sqrt(CDbl(value) / Math.PI)
            Else
                r = 0
            End If
        End Set
    End Property
End Class
```

Das komplette Programm finden Sie unter

► 3.12.1 Eigenschaften sinnvoll kapseln

3.5.3 Lese-/Schreibschutz für Eigenschaften

Es kommt häufig vor, dass bestimmte Eigenschaften nur gelesen oder nur geschrieben werden dürfen (*ReadOnly* bzw. *WriteOnly*). Um diese Art der Zugriffsbeschränkung zu realisieren, ist keinerlei 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 3.14: Lese-/Schreibschutz für Eigenschaften

VB

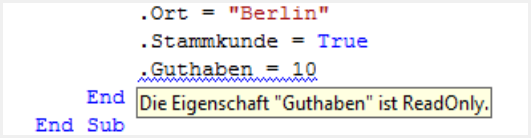
In unserer *CKunde*-Klasse soll das Guthaben für den direkten Schreibzugriff gesperrt werden. Das klingt logisch, da zur Erhöhung des Guthabens bereits die Methode *addGuthaben* existiert.

```
Public Class CKunde
    ...
     Public ReadOnly Property Guthaben() As Decimal
        Get
```

BEISPIEL 3.14: Lese-/Schreibschutz für Eigenschaften

```
            Return _guthaben
        End Get
    End Property
    ...
End Class
```

Bereits in der Entwicklungsumgebung von Visual Studio wird nun der Versuch abgewiesen, dieser Eigenschaft einen Wert zuzuweisen:



3.5.4 Statische Eigenschaften

Mitunter gibt es 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 Eigenschaften (*Klasseneigenschaften*) werden mit dem Schlüsselwort *Shared* deklariert.

Außer dem *Shared*-Schlüsselwort gibt es beim Deklarieren keine Unterschiede zu den normalen Instanzeneigenschaften.

Statische Eigenschaften können benutzt werden, ohne dass dazu eine Objektvariable deklariert und ein Objekt instanziiert werden muss! Es genügt das Voranstellen des Klassenbezeichners.

BEISPIEL 3.15: Die Klasse *CKunde* soll zusätzlich eine "einfache" Eigenschaft *Rabatt* bekommen, die für jedes Kundenobjekt immer den gleichen Wert hat.

VB
```
Public Class CKunde
    ...
    Public Shared Rabatt As Double
    ...
End Class
```

Der Zugriff ist sofort über den Klassenbezeichner möglich, ohne dass dazu eine Objektvariable erzeugt werden müsste.

BEISPIEL 3.16: Allen Kunden wird ein Rabatt von 15% zugewiesen.

VB
```
CKunde.Rabatt = 0.15
```

Vielen Umsteigern, die aus der strukturierten Programmierung kommen, bereitet es Schwierigkeiten, auf ihre globalen Variablen zu verzichten, mit denen sie Werte zwischen verschiedenen Programmmodulen ausgetauscht haben. Genau hier bieten sich statische Eigenschaften an, die z.B. in einer extra für derlei Zwecke angelegten Klasse *CAllerlei* abgelegt werden könnten.

3.5.5 Selbst implementierende Eigenschaften

Bei sehr einfachen Eigenschaften (vergleichbar mit denen, die Sie bislang "unsauber" als *Public*-Felder deklariert haben) können Sie seit VB 2010 so genannte *Auto-implemented Properties* verwenden. Der VB-Compiler generiert im Hintergrund für Sie die entsprechenden *Get*- und *Set*-Zugriffsmethoden und erzeugt außerdem ein privates Feld, um den Wert der Eigenschaft zu speichern.

Wie viel lästige Schreibarbeit Sie sparen können, soll das folgende Beispiel verdeutlichen.

BEISPIEL 3.17: Die folgende Deklaration einer selbst implementierenden Eigenschaft

VB
```
Property Ort As String = "München"
```

ist äquivalent zu

```
Private _Ort As String = "München"                  ' backing field
Property Ort As String
    Get
        Return _Ort
    End Get
    Set(value As String)

        _Ort = value
    End Set
End Property
```

Wie Sie sehen, kann der Eigenschaft auch ein Standardwert zugewiesen werden. Der Name des automatisch angelegten "backing fields" entspricht dem Namen der Eigenschaft mit vorangestelltem Unterstrich.

HINWEIS: Achten Sie auf Namenskonflikte, die entstehen können, wenn Sie eigene Felder zur Klasse hinzufügen, die auch mit einem Unterstrich (_) beginnen!

Komplette Eigenschaftsdeklarationen lassen sich in einer einzigen Zeile erledigen, wie es die folgenden Beispiele zeigen.

BEISPIEL 3.18: Einige selbst implementierende Eigenschaften

VB
```
Public Property FullName As String
Public Property FullName As String = "Max Muster"
Public Property ID As New Guid()
Public Property ErstesQuartal As New List(Of String) From {"Januar", "Februar", "März"}
```

3.6 Methoden

Methoden bestimmen die dynamischen Attribute eines Objekts, also sein Verhalten. Eine Methode ist eine Funktion, die im Körper der Klasse implementiert ist.

3.6.1 Öffentliche und private Methoden

Bereits im Kapitel 2 haben Sie 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 3.19: Die beiden öffentlichen Methoden *getAdresse()* und *addGuthaben()* arbeiten mit sechs privaten Feldern zusammen.

VB

```
Public Class CKunde
```

Private Variablen:

```
    Private _anrede As String
    Private _name As String
    Private _plz As Integer
    Private _ort As String
    Private _stammkunde As Boolean
    Private _guthaben As Decimal
```

Öffentliche Methoden:

```
Public Function getAdresse() As String
        Dim s As String = _anrede & " " & _name & vbCrLf & _plz.ToString & " " & _ort
        Return s
End Function

Public Sub addGuthaben(betrag As Decimal)
        If _stammkunde Then _guthaben += betrag
End Sub
```

Der Aufruf:

```
Private kunde1 As New CKunde()
...
Label1.Text = kunde1.getAdresse         ' erste Methode (Funktion) aufrufen
kunde1.addGuthaben(50)                  ' zweite Methode (Prozedur) aufrufen
```

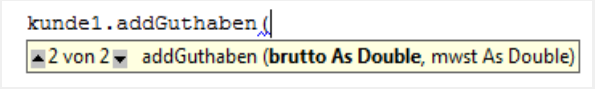
3.6.2 Überladene Methoden

Innerhalb des Klassenkörpers dürfen zwei und mehr gleichnamige Methoden konfliktfrei nebeneinander existieren, wenn sie eine unterschiedliche Signatur (Reihenfolge und Datentyp der Übergabeparameter) besitzen.

BEISPIEL 3.20: Zwei überladene Versionen einer Methode in der Klasse *CKunde*, die erste hat nur den Nettobetrag als Parameter die zweite den Bruttobetrag und die Mehrwertsteuer.

VB
```
Public Class CKunde
    ...
    Public Sub addGuthaben(betrag As Decimal)
        If _Stammkunde Then _Guthaben += betrag
    End Sub

    Public Sub addGuthaben(brutto As Double, mwst As Double)
        If _Stammkunde Then _Guthaben += CDec(brutto / (1 + mwst))
    End Sub
    ...
End Class
```

Wenn Sie diese Methoden verwenden wollen, so fällt die Auswahl im Code-Fenster leicht:



3.6.3 Statische Methoden

Genauso wie s*tatischen Eigenschaften* können *statische Methoden* (auch als *Klassenmethoden* bezeichnet) ohne Verwendung eines Objekts aufgerufen werden. Statische Methoden werden ebenfalls mit dem *Shared*-Modifizierer gekennzeichnet und 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 3.21: Wir bauen eine Klasse, in der wir wahllos einige von uns häufig benötigte Berechnungsformeln verpacken.

VB
```
Public Class MeineFormeln
    Public Shared Function kreisUmfang(radius As Double) As Double
        Return 2 * Math.PI * radius
    End Function
```

BEISPIEL 3.21: Wir bauen eine Klasse, in der wir wahllos einige von uns häufig benötigte Berechnungsformeln verpacken.

```
    Public Shared Function kugelVolumen(radius As Double) As Double
        Return 4 / 3 * Math.PI * Math.Pow(radius, 3)
    End Function

    Public Shared Function Netto(brutto As Decimal, mwst As Decimal) As Decimal
        Return brutto / (1 + mwst)
    End Function
    ...

End Class
```

Der Zugriff von außerhalb ist absolut problemlos, weil man sich nicht mehr um das lästige Instanziieren einer Objektvariablen kümmern muss.

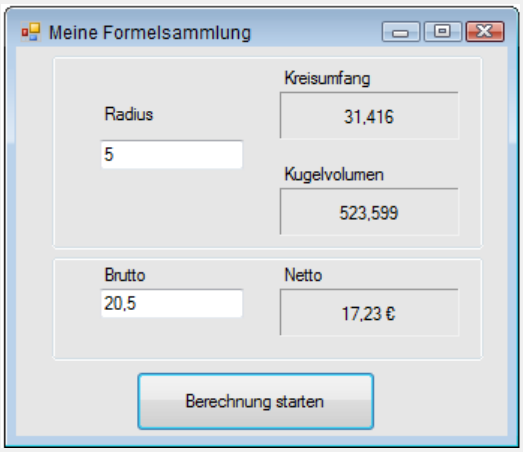
HINWEIS: Leider kann bei Klassen die *With*-Anweisung nicht verwendet werden, da diese nur bei Objekten funktioniert.

BEISPIEL 3.22: *(Fortsetzung)* Die statischen Methoden der Klasse *MeineFormeln* werden in einer Eingabemaske aufgerufen.

VB

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
  Dim r As Double = Convert.ToDouble(TextBox1.Text)          ' Kreisradius konvertieren

  Label1.Text = MeineFormeln.kreisUmfang(r).ToString("0.000")
  Label2.Text = MeineFormeln.kugelVolumen(r).ToString("0.000")

  Dim b As Double = Convert.ToDouble(TextBox2.Text)          ' Brutto konvertieren
  Label3.Text = MeineFormeln.Netto(b, 0.19).ToString("C")
End Sub
```

Ergebnis



3.7 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 .NET-Ereignismodell erhalten Sie im Kapitel 26 (Microsoft Event Pattern).

Im Folgenden werden deshalb nur die wichtigsten Grundlagen der Ereignismodellierung erläutert.

3.7.1 Ereignise deklarieren

Ein Ereignis fügen Sie der Klassendefinition über das *Event*-Schlüsselwort zu.

SYNTAX: `Public Event Ereignisname([Parameterdeklarationen])`

BEISPIEL 3.23: In der Klasse *CKunde* wird ein Ereignis mit dem Namen *GuthabenLeer* deklariert.

VB
```
Public Class CKunde
    Private _guthaben As Decimal

    Public Event GuthabenLeer(sender As Object, e As String)

    Public Sub New(betrag As Decimal)       ' Konstruktor
        ...
        _guthaben = betrag
    End Sub
    ...
```

3.7.2 Ereignis auslösen

Ausgelöst wird das Ereignis ebenfalls im Klassenkörper über das Schlüsselwort *RaiseEvent*.

SYNTAX: `RaiseEvent Ereignisname([Parameterwerte])`

BEISPIEL 3.24: *(Fortsetzung)* Das Ereignis *GuthabenLeer* "feuert" innerhalb der Methode *addGuthaben* dann, wenn das Guthaben den Wert von 10 € unterschreitet.

VB
```
    Public Sub addGuthaben(betrag As Decimal)
        _guthaben += betrag
        If _guthaben <= 10 Then
            Dim msg As String = "Das Guthaben beträgt nur noch" &
                                _guthaben.ToString("C") & "!"
            RaiseEvent GuthabenLeer(Me, msg)
        End If
    End Sub
    ...
End Class
```

Die Ereignisdefinition ist in diesem Beispiel bewusst einfach gehalten, um den Einsteiger nicht zu verschrecken. Normalerweise sollten Ereignisse immer zwei Parameter an die aufrufende Instanz übergeben: eine Referenz auf das Objekt, welches das Ereignis ausgelöst hat, und ein Objekt der *EvenArgs*- oder einer davon abgeleiteten Klasse (siehe 20.2.2). Auf Letzteres haben wir der Einfachheit wegen verzichtet und stattdessen einen einfachen Meldungsstring übergeben.

3.7.3 Ereignis auswerten

Um dem Compiler mitzuteilen, dass das Objekt über Ereignisse verfügt, müssen Sie bei der Referenzierung der Objektvariablen vor dem Objektbezeichner das Schlüsselwort *WithEvents* einfügen.

SYNTAX: `Private|Public WithEvents Objektname As Klassenname`

BEISPIEL 3.25: *(Fortsetzung)* Wir verwenden die im Vorgängerbeispiel definierte Klasse *CKunde* in einem Formular *Form1* mit einem *Button*

VB
```
Public Class Form1
```

Der Einfachheit wegen erledigen wir die Referenzierung und die Instanziierung der Objektvariablen *kunde1* in einem Schritt und weisen dabei dem Kunden ein Anfangsguthaben von 100 Euro zu.

```
  Private WithEvents kunde1 As New CKunde(100)
```

Der Eventhandler für das Ereignis *GuthabenLeer* des Kunden:

```
  Private Sub kunde1_GuthabenLeer(sender As Object, e As String) Handles kunde1.GuthabenLeer
      Label1.Text = e.ToString
  End Sub
End Class
```

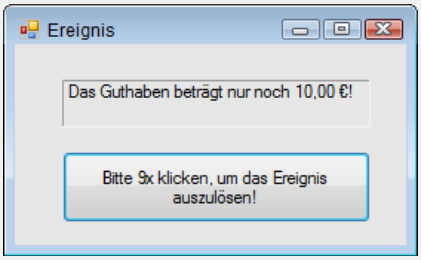
Bei jedem Klick auf den *Button* wird das Guthaben des Kunden um 10 € verringert:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Kunde1.addGuthaben(-10)
    End Sub
```

BEISPIEL 3.25: *(Fortsetzung)* Wir verwenden die im Vorgängerbeispiel definierte Klasse *CKunde* in einem Formular *Form1* mit einem *Button*

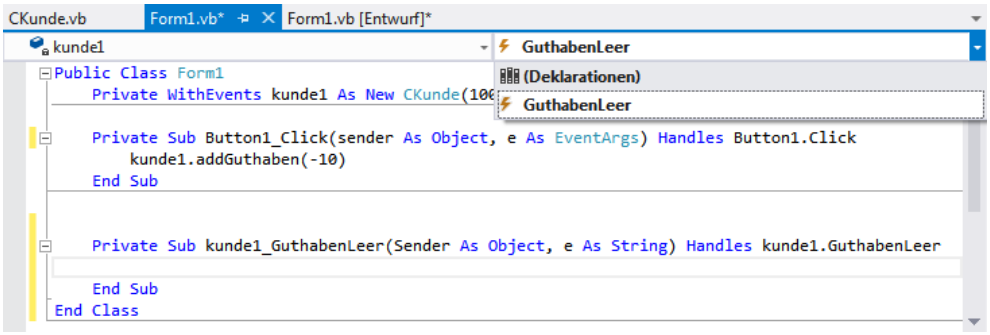
Ergebnis

Nachdem Sie den *Button* neunmal geklickt haben, feuert das Ereignis und im *Label* erscheint eine Meldung:



Visual Studio unterstützt Sie beim Erstellen des Rahmencodes der Event-Handler für selbst definierte Ereignisse natürlich genauso, wie Sie das z.B. für das *Click*-Ereignis eines *Button*s zur Genüge gewöhnt sind:

Nach dem Deklarieren der Objektvariablen wird in der linken oberen Klappbox das Objekt *kunde1* selektiert und in der rechten das Ereignis *GuthabenLeer*. Der Rahmencode des Eventhandlers wird automatisch generiert:



3.7.4 Benutzerdefinierte Ereignisse (Custom Events)

Meistens deklarieren Sie ein Ereignis mittels *Event*-Schlüsselwort unter Angabe eines Ereignisdelegaten, die Codegenerierung für die Ereignisverwaltung erfolgt automatisch.

BEISPIEL 3.26: Standardmäßige Ereignisdefinition in zwei Schritten

VB

```
Public Delegate Sub NumberChangedHandler(i As Integer)
Public Event NumberChanged As NumberChangedHandler
```

In der Regel ist diese Art der Deklaration auch völlig ausreichend, aber in einigen Fällen möchten Sie die Ereignisverwaltung doch lieber selbst in die Hand nehmen.

Benutzerdefinierte Ereignis-Accessoren (Custom Events) erlauben dem Programmierer die genaue Definition der Vorgänge, die beim Hinzufügen bzw. Entfernen eines Eventhandlers und beim Auslösen eines Events ablaufen sollen (siehe dazu auch die Ausführungen zum Microsoft Event Pattern im Kapitel 26).

Deklaration

Durch Deklarieren eines Custom Events wird dem Compiler mitgeteilt, dass er die Codegenerierung für die Ereignisverwaltung außer Kraft setzen soll, der Programmierer muss sich also nun selbst darum kümmern wie die Ereignishandler an-/abgemeldet und aufgerufen werden.

Ein Custom Event wird durch das der *Event*-Deklaration vorangestellte *Custom*-Schlüsselwort unter Angabe eines Ereignisdelegaten definiert.

BEISPIEL 3.27: Deklaration eines benutzerdefinierten Ereignisses *NumberChanged*

VB
```
Public Delegate Sub NumberChangedHandler(i As Integer)
Public Custom Event NumberChanged As NumberChangedHandler
```

Wenn Sie nach einer solchen Ereignisdeklaration die *Enter*-Taste drücken, erstellt Visual Studio automatisch den Rahmencode für die Ereignis-Acessoren (auf ähnliche Weise wie für Properties):

```
Public Custom Event NumberChanged As NumberChangedHandler
    AddHandler(value As NumberChangedHandler)

    End AddHandler

    RemoveHandler(value As NumberChangedHandler)

    End RemoveHandler

    RaiseEvent(i As Integer)

    End RaiseEvent
End Event
```

Wie Sie sehen, besteht die Deklaration eines Custom Events aus drei Sektionen, die unter folgenden Bedingungen abgearbeitet werden:

- *AddHandler*, wenn ein Ereignishandler entweder mittels *Handles*-Schlüsselwort oder *AddHandler*-Methode hinzugefügt wird.
- *RemoveHandler*, wenn ein Handler mittels *RemoveHandler*-Methode wieder entfernt wird.
- *RaiseEvent*, wenn ein Ereignis ausgelöst wird.

Sie müssen sich jetzt also selbst um das Hinzufügen und Entfernen der Handler zu bzw. aus ihrem Container (üblicherweise eine Collection), sowie um die Benachrichtigung der angeschlossenen Handler beim Auslösen des Ereignisses kümmern. Obwohl das mehr Codezeilen erfordert als eine standardmäßige Implementierung, haben Sie eine größere Flexibilität beim Programmieren.

Anwendung

Da das alles ziemlich verwirrend klingen mag, soll ein Beispiel Licht in die Dunkelheit bringen.

BEISPIEL 3.28: Benutzerdefinierte Ereignisse

VB

Eine Klasse *CCounter* erzeugt im Sekundentakt eine Zahl von 1 bis 10 und löst dabei das benutzerdefinierte Ereignis *NumberChanged* aus. Als Container für die angemeldeten Eventhandler dient eine (generische) *List*:

```
Public Class CCounter

        Public Delegate Sub NumberChangedHandler(i As Integer)
        Private handlers As New List(Of NumberChangedHandler)

        Public Custom Event NumberChanged As NumberChangedHandler
            AddHandler(value As NumberChangedHandler)
                If handlers.Count <= 3 Then handlers.Add(value)
            End AddHandler

            RemoveHandler(value As NumberChangedHandler)
                handlers.Remove(value)
            End RemoveHandler

            RaiseEvent(i As Integer)
                If i > 30 Then
                    For Each handler As NumberChangedHandler In handlers
                        handler.Invoke(i)
                    Next
                End If
            End RaiseEvent
        End Event
```

Die Methode, in welcher das Ereignis ausgelöst wird:

```
        Public Sub DoCount()
            For i As Integer = 1 To 10
                System.Threading.Thread.Sleep(1000)
                RaiseEvent NumberChanged(i * 10)
            Next
        End Sub
    End Class
```

Ein Eventhandler zeigt die erzeugten Zahlen in einer *ListBox* an:

```
    Private Sub c_NumberChanged(i As Integer)
        ListBox1.Items.Add(i.ToString)
    End Sub
```

Schließlich der Test der Klasse *CCounter*:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

BEISPIEL 3.28: Benutzerdefinierte Ereignisse

Instanziierung:

```
Dim c As New CCounter
```

Eventhandler anmelden:

```
AddHandler c.NumberChanged, AddressOf c_NumberChanged
```

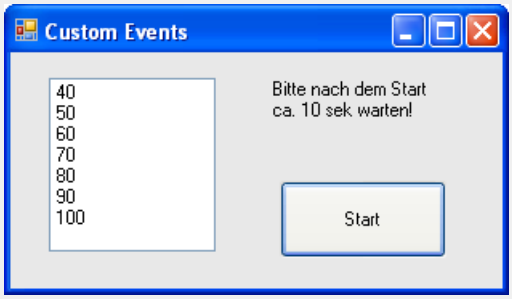
Methode aufrufen, die das Ereignis auslöst:

```
c.DoCount()
```

Eventhandler abmelden:

```
RemoveHandler c.NumberChanged, AddressOf c_NumberChanged
End Sub
```

Ergebnis



Wie Sie an diesem Beispiel sehen, ermöglichen Ihnen benutzerdefinierte Ereignisse eine freizügige Programmierung der Verwaltung Ihrer Ereignishandler. Im obigen Code haben wir als Container für die Ereignisdelegaten eine typisierte *List* gewählt, wir hätten aber auch z.B. eine (untypisierte) *ArrayList* nehmen können. In die *AddHandler*-Sektion wurde von uns eine Bedingung eingebaut, die die Anzahl der angeschlossenen Eventhandler auf 3 limitiert. Die *RaiseEvent*-Sektion bewirkt, dass die angeschlossenen Handler nur dann benachrichtigt werden, wenn die übergebene Zahl größer als 30 ist.

Ein nicht zu unterschätzender Vorteil von Custom Events ist auch das Vorhandensein eines zentralen Containers (im Beispiel die *handlers*-Liste), welcher alle angemeldeten Eventhandler innerhalb der Eventdeklaration kapselt.

Die doppelte Verwendung von *RaiseEvent* (innerhalb der *DoCount*-Methode und innerhalb der *Custom Event*-Ereignisdefinition) kann möglicherweise Verwirrung stiften. Deshalb sei hier nochmals auf den grundlegenden Unterschied hingewiesen: *RaiseEvent* in der *DoCount*-Methode löst das *NumberChanged*-Ereignis aus, die *RaiseEvent*-Sektion hingegen spezifiziert den beim Auslösen des Ereignisses auszuführenden "organisatorischen" Code, d.h., alle im Container (*List*) enthaltenen Eventhandler (Delegates) werden durchlaufen und aufgerufen.

3.8 Arbeiten mit Konstruktor und Destruktor

Eine "richtige" objektorientierte Sprache, zu der Visual Basic ja mittlerweile auch gehört, realisiert das Erzeugen und Entfernen von Objekten mit Hilfe von Konstruktoren und Destruktoren.

HINWEIS: Wenn Sie in einigen bisherigen Beispielen in den von Ihnen selbst entwickelten Klassen keinen eigenen Konstruktor definiert hatten, so wurde automatisch der von *System.Object* geerbte parameterlose *New*-Standardkonstruktor verwendet.

3.8.1 Der Konstruktor erzeugt das Objekt

Der Konstruktor ist gewissermaßen die Standardmethode der Klasse und kann in mehreren Überladungen vorhanden sein.

HINWEIS: Der Konstruktor ist immer eine *Sub* mit dem Namen *New()*.

SYNTAX:
```
Public Sub New([Parameterliste])
        ' Parameter den Feldern der Klasse zuweisen
        ' Initialisierungscode ausführen
End Sub
```

Der Konstruktor wird automatisch bei der Instanziierung eines Objekts aufgerufen und dient vor allem dazu, den Feldern des neu erzeugten Objekts Anfangswerte zuzuweisen.

HINWEIS: 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, d.h., der von *System.Object* geerbte parameterlose Konstruktor steht nicht mehr zur Verfügung!

Deklaration

Einen Konstruktor fügen Sie dem Klassenkörper ähnlich wie eine *Set*-Eigenschaftsprozedur mit dem Namen *New* hinzu. Als Parameter übergeben Sie die Werte für die Felder, die Sie initialisieren möchten.

Wie bei jeder anderen Methode können Sie auch hier mehrere überladene Konstruktoren implementieren.

BEISPIEL 3.29: Unserer Klasse *CKunde* werden zwei überladene Konstruktoren hinzugefügt.

VB
```
Public Class CKunde
```

Die (natürlich privaten) Zustandsvariablen:

```
    Private _anrede As String
    Private _name As String
    Private _plz As Integer
```

BEISPIEL 3.29: Unserer Klasse *CKunde* werden zwei überladene Konstruktoren hinzugefügt.

```
    Private _ort As String
    Private _stammkunde As Boolean
    Private _guthaben As Decimal
```

Der erste Konstruktor initialisiert nur zwei private Variablen:

```
    Public Sub New(Anrede As String, Nachname As String)
        _anrede = Anrede
        _name = Nachname
    End Sub
```

Der zweite Konstruktor initialisiert alle Variablen der Klasse. Um den Code etwas zu kürzen, wird für die ersten beiden Variablen der erste Konstruktor bemüht:

```
    Public Sub New(Anrede As String, Nachname As String, PLZ As Integer,
                   Ort As String, Stammkunde As Boolean, Guthaben As Decimal)
        Me.New(Anrede, Nachname)
        _plz = PLZ
        _ort = Ort
        _stammkunde = Stammkunde
        _guthaben = Guthaben
    End Sub

End Class
```

Aufruf

Da wir der Klasse *CKunde* zwei Konstruktoren hinzugefügt haben, ist die bisher gewohnte parameterlose Instanziierung von Objekten mit dem *New()*-Standardkonstruktor nicht mehr möglich (es sei denn, Sie fügen selbst eine weitere Überladung hinzu, die keine Parameter entgegen nimmt)!

BEISPIEL 3.30: (Fortsetzung) Zwei Objekte der oben deklarierten Klasse *CKunde* werden erzeugt und mit Anfangswerten initialisiert. Für jedes Objekt wird ein anderer überladener Konstruktor verwendet.

VB
```
Try
      Dim kunde1 As New CKunde("Herr", "Müller")
      Dim kunde2 As New CKunde("Frau", "Hummel", 12345, "Berlin", True, 100)
      ' Dim kunde3 As New CKunde()             ' geht nicht mehr!!!
      MessageBox.Show("Objekte erfolgreich erzeugt!")
Catch ex As Exception
      MessageBox.Show("Fehler beim Erzeugen des Objekts!")
End Try
```

Sie sehen, dass das Initialisieren der Objekte viel einfacher geworden ist. Anstatt umständlich eine Eigenschaft nach der anderen zuzuweisen, geht das jetzt mit einer einzigen Anweisung.

3.8.2 Bequemer geht's mit einem Objekt-Initialisierer

Vor allem in Hinblick auf die in der neuen LINQ-Technologie erforderlichen anonymen Typen (siehe Kapitel 6) wurden so genannte Objekt-Initialisierer eingeführt. Damit können nun öffentliche Eigenschaften und Felder von Objekten ohne das explizite Vorhandensein des jeweiligen Konstruktors in beliebiger Reihenfolge initialisiert werden.

Der Objekt-Initialisierer erwartet das Schlüsselwort *With* unmittelbar nach dem Erzeugen des Objekts. Anschließend folgt die in geschweiften Klammern eingeschlossene Liste der zu initialisierenden Mitglieder.

HINWEIS: Einem Objektinitialisierer können nur Eigenschaften oder öffentliche Felder übergeben werden, das Initialisieren privater Felder, wie im obigen Konstruktor-Beispiel, ist nicht möglich!

BEISPIEL 3.31: Erzeugen einer Instanz der Klasse *CPerson*

VB
```
Public Class CPerson
   Public Name As String
   Public Strasse As String
   Public PLZ As Integer
   Public Ort As String
End Class
```

Der Objektinitialisierer:

```
Dim person1 As New CPerson With {.Name = "Müller", .Strasse = "Am Waldesrand 7",
                                 .PLZ = 12345, .Ort = "Musterhausen"}
```

BEISPIEL 3.32: Verschachtelte Objektinitialisierung beim Erzeugen einer Instanz der Klasse *Rectangle*

VB
```
Dim rect As New Rectangle With {.Location = New Point With {.X = 3, .Y = 7},
                           .Size = New Size With {.Width = 19, .Height = 34} }
```

BEISPIEL 3.33: Initialisieren einer Collection aus Objekten

VB
```
Dim personen() = {New CPerson With {.Name = "Müller", .Strasse = "Am Wald 7",
                                    .PLZ = 12345, .Ort = "Musterhausen"},
                  New CPerson With {.Name = "Meier", .Strasse = "Hauptstr. 2",
                                    .PLZ = 2344, .Ort = "Walldorf"},
                  New CPerson With {.Name = "Schulz", .Strasse = "Wiesenweg 5",
                                    .PLZ = 32111, .Ort = "Biesdorf"}}
```

3.8.3 Destruktor und Garbage Collector räumen auf

Das Pendant zum Konstruktor ist aus objektorientierter Sicht der Destruktor. Da der Lebenszyklus eines Objektes bekanntlich mit dessen Zerstörung und der Freigabe der belegten Speicherplatz-

ressourcen 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.

Der *Public*-Zugriffsmodifizierer entfällt, da Sie selbst den Destruktor nicht aufrufen dürfen, auch Parameter dürfen nicht übergeben werden.

SYNTAX:
```
Protected Overrides Sub Finalize()
        ' hier Code für Aufräumarbeiten implementieren
End Sub
```

BEISPIEL 3.34: Destruktor und Garbage Collector

VB

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 Shared anzahl As Integer = 0
```

Konstruktor:

```
    Public Sub New()
        anzahl += 1
    End Sub
```

Quasi-Destruktor:

```
    Protected Overrides Sub Finalize()
        anzahl -= 1
        MyBase.Finalize()        ' Aufruf der Basisklassenmethode
    End Sub

End Class
```

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:

```
    Private kunde1 As CKunde            ' Objekt referenzieren
```

BEISPIEL 3.34: Destruktor und Garbage Collector

Objekt erzeugen:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    kunde1 = New CKunde
End Sub
```

Objekt entfernen:

```
Private Sub Button2_Click(sender As Object, e AsEventArgs) Handles Button2.Click
    kunde1 = Nothing                    ' Objekt dereferenzieren
End Sub
```

Anzeige der im Speicher befindlichen Instanzen im Sekundentakt:

```
Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
    Label1.Text = CKunde.anzahl.ToString
End Sub
```

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 ziemlich lange 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.

[1] Der Garbage Collector läuft in einem eigenen Thread, er wird nur dann aufgerufen, wenn sich die anderen Threads in einem sicheren Zustand befinden.

HINWEIS: Obiges Beispiel sollten Sie aufgrund seiner Unberechenbarkeit keinesfalls als Vorbild für ähnliche Zählaufgaben verwenden!

Da wegen der Unberechenbarkeit der Objektvernichtung der Umgang mit der *Finalize*-Methode ziemlich problematisch ist, sollten Sie für das definierte Freigeben von Objekten besser eine separate Methode verwenden (siehe *Dispose*-Methode).

3.8.4 Mit Using den Lebenszyklus des Objekts kapseln

Auch mit dem Schlüsselwort *Using* kann man selbst für das sichere Erzeugen und Vernichten von Objekten sorgen. Voraussetzung dafür ist, dass das Objekt die *IDisposable*-Schnittstelle implementiert. Hinter den Kulissen wird ein *Try-Finally*-Block um das entsprechende Objekt generiert und beim Beendigen für das Objekt *Dispose()* aufgerufen.

BEISPIEL 3.35: Sicheres Erzeugen und Freigeben von ADO.NET-Objekten (siehe 23.3.5)

VB
```
...
Using conn As New SqlConnection(connString)
   Using cmd As New SqlCommand(cmdString, conn)
      conn.Open()
      cmd.ExecuteNonQuery()
   End Using
End Using
...
```

Ein weiteres Beispiel für *Using* finden Sie im Praxisbeispiel

► 8.8.3 Ein Memory Mapped File verwenden

3.9 Vererbung und Polymorphie

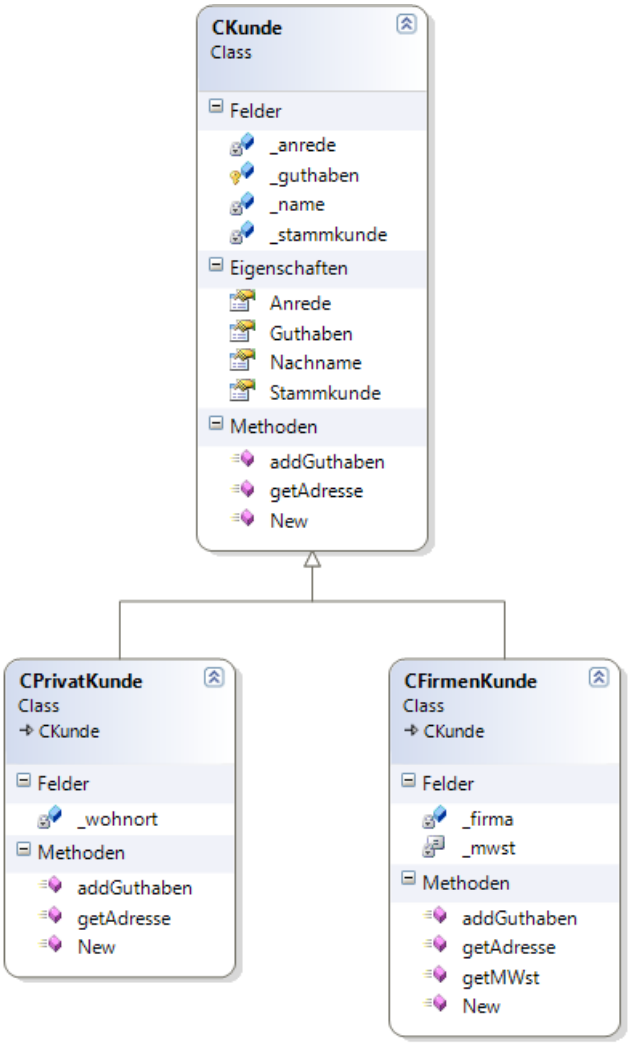
Ein zentrales OOP-Thema ist die *Vererbung*, die es ermöglicht, Klassen zu definieren, die von anderen Klassen abhängen. Eng mit der Vererbung verknüpft ist die *Polymorphie* (Vielgestaltigkeit). Man versteht darunter die Fähigkeit von Subklassen, die Methoden der Basisklasse mit unterschiedlichen Implementierungen zu verwenden. Visual Basic 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!

3.9.1 Vererbungsbeziehungen im Klassendiagramm

Mittels *Unified Modeling Language* (UML) lassen sich Vererbungsbeziehungen zwischen verschiedenen Klassen grafisch darstellen.



Das obige, 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,

[1] Der Umgang mit dem Klassendesigner wird ausführlich in Kapitel 25 beschrieben.

der den Kunden in prozentualer Abhängigkeit von den getätigten Einkäufen gewährt wird). Die *New*-Methoden sind nichts weiter als die Konstruktoren der entsprechenden Klassen.

3.9.2 Überschreiben von Methoden (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* tauchen allerdings nochmals in den beiden Subklassen auf, warum? In unserem Beispiel handelt es sich um so genannte *überschriebene Methoden*, d.h., Adresse und Guthaben sollen für Privatkunden auf andere Weise als für Firmenkunden ermittelt werden. Genaueres dazu erfahren Sie im nächsten Abschnitt.

HINWEIS: Verwechseln Sie das *Überschreiben* von Methoden nicht mit dem in 2.8.5 beschriebenen *Überladen* von Methoden. Beides hat nichts, aber auch gar nichts, miteinander zu tun!

3.9.3 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 *Overridable*-Methodendeklarationen[1].

```
Public Class CKunde
```

Die privaten Felder:

```
    Private _anrede As String
    Private _name As String
    Private _stammkunde As Boolean
```

Auf das folgende Feld soll auch eine Subklasse zugreifen können:

```
    Protected _guthaben As Decimal
```

Der eigene Konstruktor:

```
    Public Sub New(Anrede As String, Nachname As String)
        _anrede = Anrede
        _name = Nachname
    End Sub
```

[1] Eigentlich hätten wir die Klasse auch noch als *MustInherit* deklarieren müssen (siehe dazu 3.6.6).

Die Eigenschaften:

```
    Public WriteOnly Property Anrede()
        Set(value)
            _anrede = value
        End Set
    End Property

    Public WriteOnly Property Nachname()
        Set(value)
            _name = value
        End Set
    End Property

    Public Property Stammkunde() As Boolean
        Get
            Return _stammkunde
        End Get
        Set(value As Boolean)
            _stammkunde = value
        End Set
    End Property

    Public ReadOnly Property Guthaben() As Decimal
        Get
            Return _guthaben
        End Get
    End Property
```

Die folgenden beiden Methoden sollen von den Subklassen überschrieben werden können:

```
    Public Overridable Function getAdresse() As String
        Return _anrede & " " & _name
    End Function

    Public Overridable Sub addGuthaben(betrag As Decimal)
        If _stammkunde Then _guthaben += betrag
    End Sub

End Class
```

Subklassen

Die erste Methode in der Subklasse ist in der Regel der Konstruktor. Dieser Konstruktor benutzt das *MyBase*-Schlüsselwort, um den Konstruktor der Basisklasse aufzurufen. Falls aber die Basisklasse über keinen eigenen Konstruktor verfügt, wird der Standardkonstruktor automatisch aufgerufen, wenn ein Objekt aus einer Subklasse erzeugt wird.

Das *Overrides*-Schlüsselwort der beiden Funktionen bedeutet, dass hier die in der Basisklasse definierten Funktionen überschrieben werden. Das erlaubt der Subklasse, eine eigene Implementierung der Funktionen zu realisieren.

HINWEIS: Wenn Sie das *Overrides*-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.

Der Code für die Subklasse *CPrivatKunde:*

```
Public Class CPrivatKunde
    Inherits CKunde
    Private _wohnort As String
```

Der Konstruktor ist notwendig, weil auch die Basisklasse einen eigenen Konstruktor verwendet:

```
    Public Sub New(Anr As String, Name As String, Ort As String)
        MyBase.New(Anr, Name)  ' Aufruf des Konstruktors der Basisklasse
        Me._wohnort = Ort      ' klassenspezifische Ergänzung
    End Sub
```

Die Methoden werden überschrieben:

```
    Public Overrides Function getAdresse() As String
        Return MyBase.Adresse & vbCrLf & _wohnort
    End Function

    Public Overrides Sub addGuthaben(geld As Decimal)
```

5% des Rechnungsbetrags werden als Guthaben angerechnet (Direktzugriff auf die *Protected*-Variable der Basisklasse *CKunde*):

```
        _guthaben += 0.05 * geld
     End Sub
End Class
```

Der Code für die Subklasse *CFirmenKunde* unterscheidet sich in einigen Details:

```
Public Class CFirmenKunde
    Inherits CKunde
    Private _firma As String
    Private Const _mwst As Double = 0.19
```

Konstruktor (notwendig, weil Basisklasse eigenen Konstruktor verwendet):

```
    Public Sub New(Anr As String, Name As String, Frm As String)
        MyBase.New(Anr, Name)           ' Aufruf des ererbten Konstruktors
        Me._firma = Frm
    End Sub
```

Überschreiben der überschreibbaren Methoden:

```
    Public Overrides Function getAdresse() As String
        Return MyBase.Adresse & vbCrLf & _firma
    End Function

    Public Overrides Sub addGuthaben(brutto As Decimal)
        Dim netto As Decimal = brutto / CDec((1 + _mwst))         ' Netto berechnen
        MyBase.addGuthaben(netto * 0.01D)        ' 1% als Guthaben angerechnet
    End Sub
```

Eine normale Methode:

```
    Public Function getMWst() As Double
        Return _mwst
    End Function

End Class
```

Subklasse CPrivatKunde

Diese Klasse erbt alle Eigenschaften und Methoden der Basisklasse, wird also sozusagen um deren Code "erweitert". Das *Overrides*-Schlüsselwort der beiden Methoden bedeutet, dass hier die in der Basisklasse als *Overridable* definierten Funktionen überschrieben werden. Das erlaubt der Subklasse, eine eigene Implementierung der Funktionen zu realisieren.

Der Code für die Subklasse *CPrivatKunde:*

```
Public Class CPrivatKunde
    Inherits CKunde                          ' erbt von der Basisklasse CKunde!
    Private _wohnort As String
```

Ein eigener Konstruktor ist notwendig, weil auch die Basisklasse einen eigenen Konstruktor verwendet:

```
    Public Sub New(Anr As String, Name As String, Ort As String)
        MyBase.New(Anr, Name)                 ' Aufruf des Konstruktors der Basisklasse
        Me._wohnort = Ort      ' klassenspezifische Ergänzung
    End Sub
```

Die Methode *getAdresse()* wird so überschrieben, dass zusätzlich zu Anrede und Name (von der Basisklasse geerbt) noch der Wohnort des Privatkunden angezeigt wird:

```
    Public Overrides Function getAdresse() As String
        Return MyBase.getAdresse & vbCrLf & _wohnort
    End Function
```

Die Methode *addGuthaben()* wird komplett neu überschrieben. Ohne Rücksicht auf die Zugehörigkeit zur Stammkundschaft werden jedem Privatkunden 5% vom Rechnungsbetrag als Bonusguthaben angerechnet:

```
    Public Overrides Sub addGuthaben(geld As Decimal)
```

Hier erfolgt ein Direktzugriff auf die *Protected*-Variable *_guthaben* der Basisklasse *CKunde*:

```
        _guthaben += 0.05 * geld
    End Sub

End Class
```

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
    Inherits CKunde

    Private _firma As String
    Private Const _mwst As Double = 0.19
```

Auch hier ist ein Konstruktor notwendig (weil Basisklasse eigenen Konstruktor verwendet):

```
    Public Sub New(Anr As String, Name As String, Frm As String)
        MyBase.New(Anr, Name)           ' Aufruf des ererbten Konstruktors
        Me._firma = Frm
    End Sub
```

Überschreiben der überschreibbaren Methoden:

```
    Public Overrides Function getAdresse() As String
        Return MyBase.getAdresse & vbCrLf & _firma
    End Function

    Public Overrides Sub addGuthaben(brutto As Decimal)
        Dim netto As Decimal = brutto / CDec((1 + _mwst))  ' Netto berechnen
        MyBase.addGuthaben(netto * 0.01)                   ' 1% als Guthaben angerechnet
    End Sub
```

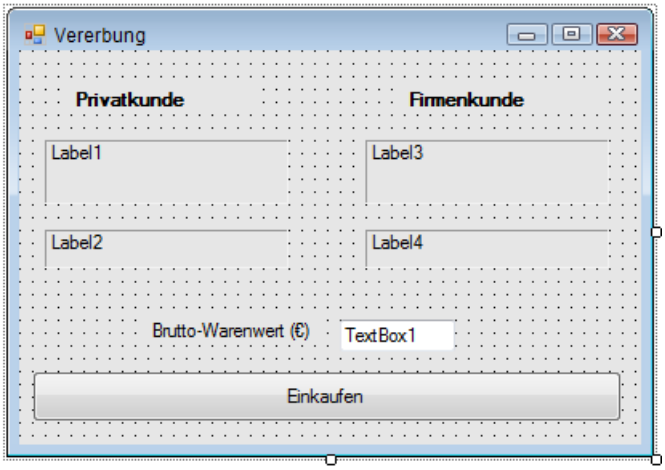
Und zum Schluss noch eine ganz normale Methode:

```
    Public Function getMWst() As Double
        Return _mwst
    End Function
End Class
```

Die Implementierung unserer drei Klassen ist geschafft!

Testoberfläche

Um die Funktionsfähigkeit der drei Klassen zu testen, gestalten Sie folgende Benutzerschnittstelle:



3.9.4 Objekte implementieren

Es genügt, wenn wir mit nur zwei Objekten (ein Privat- und ein Firmenkunde) arbeiten:

```
Public Class Form1
    Private kunde1 As CPrivatKunde
    Private kunde2 As CFirmenKunde
```

Im Konstruktor des Formulars werden die beiden Objekte erzeugt. Fügen Sie zunächst den Rahmencode des Konstruktors hinzu, indem Sie in den beiden Comboboxen am oberen Rand des Codefenster Folgendes einstellen: *Klassenname* = *Form1*, *Methodenname* = *New*.

Die Ja-/Nein-Eigenschaft *StammKunde* muss allerdings extra zugewiesen werden, da es dazu keinen passenden Konstruktor gibt.

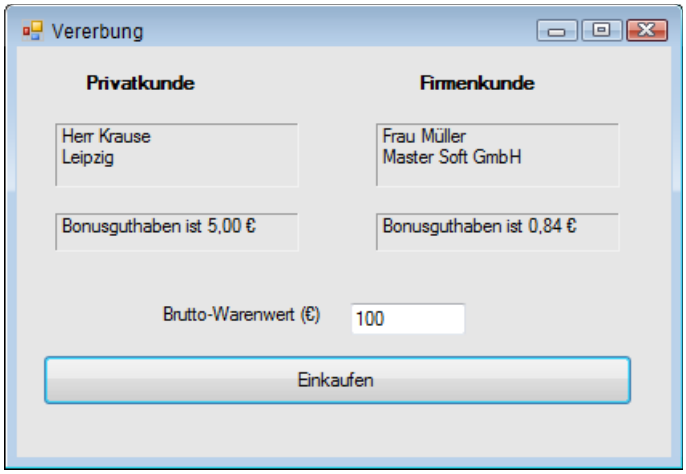
```
Public Sub New()

        ' Dieser Aufruf ist für den Windows Form-Designer erforderlich.
        InitializeComponent()
        ' Fügen Sie Initialisierungen nach dem InitializeComponent()-Aufruf hinzu.

        kunde1 = New CPrivatKunde("Herr", "Krause", "Leipzig")
        kunde1.Stammkunde = False

        kunde2 = New CFirmenKunde("Frau", "Müller", "Master Soft GmbH")
        kunde2.Stammkunde = True
        TextBox1.Text = "100"
    End Sub
```

Bei Klick auf den "Einkaufen"-Button werden für jedes Objekt diverse Eigenschaften abgefragt und Methoden aufgerufen:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim brutto As Decimal = 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")
    End Sub
End Class
```

Praxistest

Überzeugen Sie sich nun davon, dass die drei Klassen wie gewünscht zusammenarbeiten und dass Vererbung tatsächlich funktioniert.



Die Werte in der 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.

3.9.5 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 3.36: Zur Klasse *CKunde* fügen wir eine Methode *test* hinzu

VB

```
Public Class CKunde                    ' Basisklasse
    ...
    Public Sub test()
        MessageBox.Show("Hallo Kunde!")
    End Sub
End Class
```

Eine Methode gleichen Namens fügen wir auch zur Klasse *CPrivatkunde* hinzu:

```
Public Class CPrivatkunde              ' abgeleitete Klasse
    Inherits CKunde

    Public Sub test()
        MessageBox.Show("Hallo Privatkunde!")
    End Sub

End Class
```

Im Quellcode-Editor erscheint der Name *test()* grün unterschlängelt. Der entsprechende Warnhinweis lautet: *Sub "test" führt Shadowing für einen überladbaren Member durch, der in der Basisklasse CKunde deklariert ist. Wenn Sie die Basismethode überladen möchten, muss die Methode als "Overloads" deklariert werden.*

Sie lassen diese Warnung unbeachtet. Der Test erfolgt in *Form1*:

```
Public Class Form1
 Private kunde1 As New CPrivatkunde()

 Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
     kunde1.test()
 End Sub
End Class
```

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 Schlüsselwort *Overloads* markieren.

BEISPIEL 3.37: Die folgende Änderung macht das Vorgängerbeispiel transparenter (Ergebnis bleibt dasselbe)

VB
```
Public Class CPrivatkunde           ' abgeleitete Klasse
  Inherits CKunde
  ...
  Public Overloads Sub test()
      MessageBox.Show("Hallo Privatkunde!")
  End Sub

End Class
```

3.9.6 Allgemeine Hinweise und Regeln zur Vererbung

Nachdem wir nun am praktischen Beispiel die Programmierung von Vererbungsbeziehungen kennen gelernt 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 *Overridable* deklariert werden.
- Fehlt das Schlüsselwort *Overridable* 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 *Overrides* deklariert werden.
- Wenn Sie das *Overrides*-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 das der Klassendeklaration nachgestellte Schlüsselwort *Inherits* bekannt gemacht:

SYNTAX:
```
Public Class SubKlasse
Inherits Basisklasse
     ' ... Implementierungscode
End Class
```

- Eine Subklasse kann immer nur von einer einzigen Basisklasse abgeleitet werden (keine multiple Vererbung möglich).
- Mit dem *MyBase*-Objekt kann von den Subklassen auf die Basisklasse zugegriffen werden, mit dem *Me*-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 (*MyBase*-Schlüsselwort).
- Falls aber die Basisklasse über keinen eigenen Konstruktor verfügt, wird der Standardkonstruktor automatisch aufgerufen, wenn ein Objekt aus einer Subklasse erzeugt wird.

3.9.7 Polymorphe Methoden

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 Kompilierens 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 3.38: Polymorphe Methoden

VB

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, die diesmal in einem Array mit drei Feldern vom Typ der Basisklasse *CKunde* abgelegt werden sollen:

```
Public Class Form1
    Private Kunden(2) As CKunde          ' Referenz auf Basisklasse CKunde!
```

Im Konstruktorcode des Formulars erzeugen wir einen Privat- und zwei Firmenkunden:

```
    Public Sub New()
        InitializeComponent()
        Dim kunde1 As New CPrivatKunde("Herr", "Krause", "Leipzig")
        kunde1.Stammkunde = False
```

BEISPIEL 3.38: Polymorphe Methoden

```
        Dim kunde2 As New CFirmenKunde("Frau", "Müller", "Master Soft GmbH")
        kunde2.Stammkunde = True
        Dim Kunde3 As New CFirmenKunde("Herr", "Maus", "Manfreds Internet AG")
        Kunde3.Stammkunde = False
```

Unser eingangs deklariertes Array nimmt nun Privat- und Firmenkunden in wahlloser Reihenfolge auf:

```
        Kunden(0) = kunde1
        Kunden(1) = kunde2
        Kunden(2) = Kunde3
        TextBox1.Text = "100"
    End Sub
```

Das Array wird zunächst in einer *For-Next*-Schleife durchlaufen. Dabei werden die polymorphen Methoden (das sind die mit *Overridable* bzw. *Overrides* deklarierten) für alle Objekte aufgerufen. Es werden also sowohl die Adressen ausgegeben als auch für jeden Kunden das Guthaben berechnet, nachdem er für 100 € Waren gekauft hat.

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim brutto As Decimal = Convert.ToDecimal(TextBox1.Text)
        Label1.Text = String.Empty
        For i As Integer = 0 To Kunden.Length - 1
            Kunden(i).addGuthaben(brutto)
            Label1.Text = Label1.Text & vbCrLf & Kunden(i).getAdresse & vbCrLf &
                          Kunden(i).Guthaben.ToString("C") & vbCrLf
        Next
    End Sub
```

Obwohl im Array die Objekte bunt durcheinander gewürfelt sind, "weiß" das Programm zur Laufzeit genau, welche Implementierung der Methoden *getAdresse* und *addGuthaben* für den Privat- und für den Firmenkunden die richtige ist: Genau darin liegt der springende Punkt zum Verständnis der Polymorphie!

BEISPIEL 3.39: Die alternative Implementierung obigen Codes mittels *For Each*-Schleife bringt das Problem der Polymorphie noch deutlicher auf den Punkt

VB

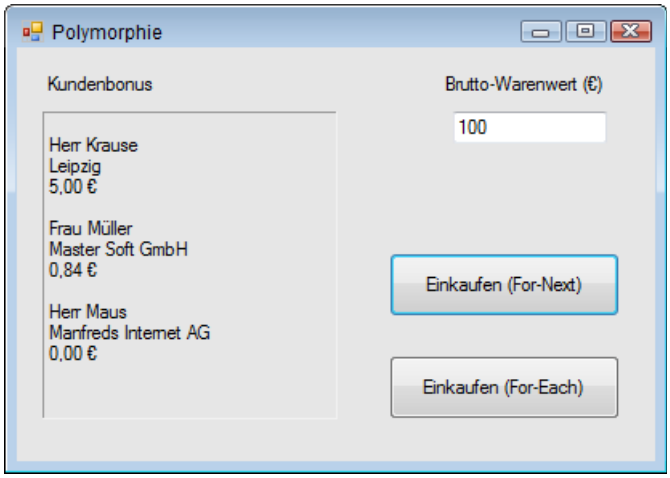
```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        Dim brutto As Decimal = Convert.ToDecimal(TextBox1.Text)
        Label1.Text = String.Empty
```

Die Schleifenvariable *ku* ist eine Referenz auf die Basisklasse *CKunde*:

```
        For Each ku As CKunde In Kunden
            ku.addGuthaben(brutto)
            Label1.Text = Label1.Text & vbCrLf & ku.getAdresse & vbCrLf &
                                        ku.Guthaben.ToString("C") & vbCrLf
        Next
    End Sub
End Class
```

Praxistest

Das Ergebnis anhand der abgebildeten Testoberfläche beweist, dass Vererbung und Polymorphie tatsächlich untrennbar miteinander verknüpft sind. Egal ob Privat- oder Firmenkunde – es werden immer die passenden Methodenimplementierungen aufgerufen:



HINWEIS: 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.

3.10 Besondere Klassen und Features

Zum Schluss wollen wir noch auf einige wichtige Klassen und Features eingehen, die in der OOP eine besondere Rolle spielen.

3.10.1 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, können diese mit dem Schlüsselwort *MustInherit* gekennzeichnet werden.

BEISPIEL 3.40: In unserem Vorgängerbeispiel werden von der Klasse *CKunde* keine Instanzen gebildet, sie kann deshalb als abstrakt deklariert werden.

VB
```
Public MustInherit Class CKunde
...
```

[1] Können Sie sich vielleicht vorstellen, wie eine Instanz der Klasse *Fahrzeug* konkret aussehen soll?

BEISPIEL 3.40: In unserem Vorgängerbeispiel werden von der Klasse *CKunde* keine Instanzen gebildet, sie kann deshalb als abstrakt deklariert werden.

```
End Class
```

Während die Referenzierung nach wie vor möglich ist

```
Dim Kunde As CKunde
```

schlägt der Versuch einer Instanziierung fehl:

```
Kunde = New CKunde("Herr", "Krause")              ' Fehler
```

HINWEIS: Abstrakte Klassen ähneln einem weiteren wichtigen Softwarekonstrukt der OOP, der Schnittstelle (siehe Abschnitt 3.10.1).

3.10.2 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 *Overrides* überschrieben werden. Zur Kennzeichnung abstrakter Methoden verwenden Sie das Schlüsselwort *MustOverride*.

HINWEIS: Die Deklaration einer abstrakten Methode erfolgt in einer Zeile, also ohne Rumpf.

BEISPIEL 3.41: Die Funktion *getAdresse* einer abstrakten *CKunde*-Klasse wird in den Subklassen komplett überschrieben und kann deshalb anstatt mit *Overridable* mit *MustOverride* deklariert werden.

VB

```
Public MustInherit Class CKunde
...
  Public MustOverride Function getAdresse() As String       ' abstrakte virtuelle Methode
...
End Class

Public Class CPrivatKunde
    Inherits CKunde
...
    Public Overrides Function getAdresse() As String  ' überschreibt Methode der Basisklasse
        Return _Anrede & "  " & _Name & "  " & _Wohnort
    End Function
...
End Class
```

3.10.3 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 *NotInheritable* schützen.

BEISPIEL 3.42: Die Klasse *CPrivatKunde* wird versiegelt und darf deshalb keine Nachkommen haben.

VB
```
Public NotInheritable Class CPrivatKunde
    Inherits CKunde
    ...
End Class
```

Beim Versuch, davon eine Subklasse abzuleiten, schlägt Ihnen der Compiler erbarmungslos auf die Pfoten:

```
Public Class CStudent           ' Fehler!!!
    Inherits CPrivatKunde
    ...
End Class
```

HINWEIS: Eine versiegelte Klasse kann niemals eine Basisklasse sein. Vererbungsmodifikatoren wie *MustInherit* und *Overridable* 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.

3.10.4 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 (z.B. *Form1.Designer.vb*) vom Code des Entwicklers (*Form1.vb*) getrennt, was zu einer gesteigerten Übersichtlichkeit beiträgt, wovon man sich nach Öffnen eines neuen Windows Forms-Projekts selbst überzeugen kann).

Die Programmierung ist denkbar einfach, denn alle Teile der Klasse sind lediglich mit dem Modifizierer *Partial* zu kennzeichnen.

BEISPIEL 3.43: Eine einfache Klasse *CKunde*

VB
```
Public Class CKunde
        Private _name As String
        Protected _guthaben As Decimal = 0

        Public Property NachName() As String
            Get
```

BEISPIEL 3.43: Eine einfache Klasse *CKunde*

```
                Return _name
            End Get
            Set(value As String)
                _name = value
            End Set
        End Property

        Public Property Guthaben() As Decimal
            Get
                Return _guthaben
            End Get
            Set(value As Decimal)
                _guthaben = value
            End Set
        End Property
        Public Sub addGuthaben(betrag As Decimal)
            _guthaben += betrag
        End Sub
    End Class
```

Obige Klasse könnte (als eine von mehreren Möglichkeiten) wie folgt in drei partielle Klassen aufgesplittet werden:

```
Partial Public Class CKunde
        Private _name As String
        Protected _guthaben As Decimal = 0
End Class

Public Class CKunde
        Public Property NachName() As String
            Get
                Return _name
            End Get
            Set(value As String)
                _name = value
            End Set
        End Property

        Public Property Guthaben() As Decimal
            Get
                Return _guthaben
            End Get
            Set(value As Decimal)
                _guthaben = value
            End Set
        End Property
End Class
```

BEISPIEL 3.43: Eine einfache Klasse *CKunde*

```
Partial Public Class CKunde
        Public Sub addGuthaben(betrag As Decimal)
            _guthaben += betrag
        End Sub
End Class
```

Wie Sie sehen, kann (muss aber nicht) bei einer der partiellen Definitionen auf *Partial* verzichtet werden, diese Klasse ist dann gewissermaßen die Ausgangsklasse, die durch den Code der anderen partiellen Klassen erweitert wird.

HINWEIS: Auch *Structure*- oder *Interface*-Definitionen können mittels *Partial*-Modifizierer gesplittet werden!

3.10.5 Die Basisklasse System.Object

Jedes Objekt in .NET ist von der Basisklasse *System.Object* abgeleitet. Diese Klasse ist Teil des Microsoft .NET Frameworks und beinhaltet die Basiseigenschaften und -methoden, wie sie für ein .NET-Objekt erforderlich sind.

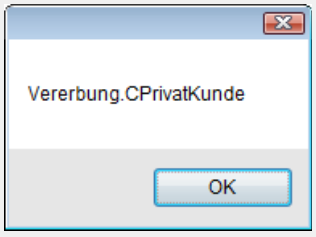
Alle öffentlichen Eigenschaften und Methoden von *System.Object* stehen automatisch auch in jedem Objekt zur Verfügung, welches Sie erzeugt haben. Beispielsweise ist in *System.Object* bereits ein Standardkonstruktor enthalten. Wenn Sie in Ihrem Objekt keinen eigenen Konstruktor definiert haben, wird 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 3.44: Die *ToString*-Methode liefert den Namen der Anwendungskomponente (die Windows-Anwendung heißt hier Vererbung) und die Klassenzugehörigkeit von *kunde1*.

VB
```
MessageBox.Show(kunde1.ToString)
```

Ergebnis

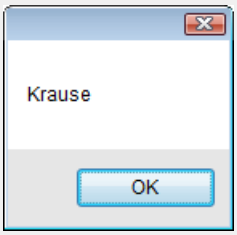


Sie können das standardmäßige Verhalten von *ToString* mittels *Overrides*-Schlüsselwort verändern. Dies erlaubt Ihnen eine individuelle Implementierung einiger Eigenschaften bzw. Methoden von *System.Object*.

BEISPIEL 3.45: Die gleiche *ToString*-Methode des Vorgängerbeispiels liefert nun den Namen des Kunden, wenn Sie die folgende Methode zum Klassenkörper von *CKunde* hinzufügen.

VB
```
Public Overrides Function ToString() As String
   Return _name
End Function
```

Ergebnis



3.10.6 Property-Accessors

Die *Get*- und *Set*- Accessoren von Eigenschaften hatten in den ersten .NET-Versionen von Visual Basic die gleiche Sichtbarkeit wie die Eigenschaft zu der sie gehören. Seit .NET 2.0 ist es möglich, den Zugriff auf einen dieser Accessoren zu beschränken. Meist ist dies für den *Set*-Accessor sinnvoll, während der *Get*-Accessor in der Regel öffentlich bleiben soll.

BEISPIEL 3.46: Eine Eigenschaft mit *Get*- und *Set*-Accessoren. Der *Get*-Accessor besitzt die gleiche Sichtbarkeit wie die *KontoNummer*-Eigenschaft, während der *Set*-Accessor nur einen *Friend*-Zugriff erlaubt.

VB
```
Private _knr As String
Public Property KontoNummer() As String
    Get
        Return _knr
    End Get
    Friend Set(value As String)
        _knr = value
    End Set
End Property
```

3.11 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.

Einen Ausweg bietet die Verwendung von Schnittstellen, diese bieten fast alle Möglichkeiten der Mehrfachvererbung, vermeiden aber deren Nachteile.

HINWEIS: Schnittstellen dienen dazu, um gemeinsame Merkmale ansonsten unabhängiger Klassen beschreiben zu können.

Eine Schnittstelle können Sie sich zunächst wie eine abstrakte Klasse (siehe 3.10.1) vorstellen, in welcher nur abstrakte Methoden definiert werden[1].

3.11.1 Definition einer Schnittstelle

Eine Schnittstelle können Sie zu Ihrem Projekt genauso hinzufügen wie eine neue Klasse. Anstatt des Schlüsselworts *Class* verwenden Sie aber *Interface*.

HINWEIS: Laut Konvention sollte der Namen einer Schnittstelle immer mit "I" beginnen.

BEISPIEL 3.47: Eine Schnittstelle *IPerson*, die zwei Eigenschaften und eine Methode definiert [2]

VB
```
Public Interface IPerson
  Property Nachname As String
  Property Vorname As String
  Function getName() As String
End Interface
```

Vielleicht vermissen Sie im obigen Beispiel die Zugriffsmodifizierer (***Public*** *Property* ...), diese aber haben 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!

3.11.2 Implementieren einer Schnittstelle

Die implementierende Klasse benutzt anstatt des Schlüsselworts *Inherits* das Schlüsselwort *Implements*.

HINWEIS: Die implementierende Klasse geht die Verpflichtung ein, ausnahmslos **alle** Mitglieder der Schnittstelle zu implementieren!

BEISPIEL 3.48: Die Klasse *CKunde* implementiert die Schnittstelle *IPerson*

VB
```
Class CKunde
   Implements IPerson
        ...
```

[1] Dieser Vergleich hinkt natürlich wegen der auch bei abstrakten Klassen nicht möglichen Mehrfachvererbung.

[2] Das Interface verwendet selbstimplementierende Eigenschaften.

BEISPIEL 3.48: Die Klasse *CKunde* implementiert die Schnittstelle *IPerson*

Die von *IPerson* geerbten abstrakten Klassenmitglieder müssen implementiert werden:

```
        Public Property Nachname As String Implements IPerson.Nachname
        Public Property Vorname As String Implements IPerson.Vorname

        Public Function getName() As String Implements IPerson.getName
           Return _Vorname & " " & _Nachname
        End Function
```

Es folgen die normalen Klassenmitglieder:

```
        ...
End Class
```

Den kompletten Code finden Sie im Praxisbeispiel

► 3.12.4 Schnittstellenvererbung verstehen

Dort wird auch gezeigt, wie man eine mit abstrakten Methoden ausgestattete abstrakte Klasse ganz leicht in eine Schnittstelle überführen kann.

3.11.3 Abfragen, ob eine Schnittstelle vorhanden ist

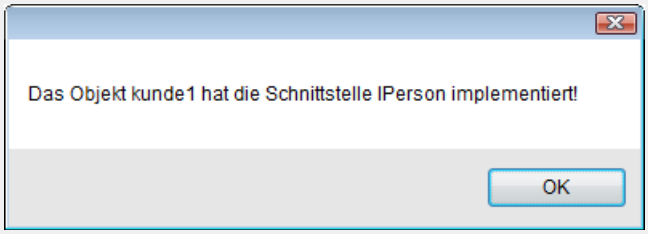
Manchmal möchte man vor der eigentlichen Arbeit mit einem Objekt wissen, ob dieses eine bestimmte Schnittstelle implementiert hat. Eine Lösung bietet eine Abfrage mit dem *TypeOf*-Operator.

BEISPIEL 3.49: Wir ergänzen die Bedienoberfläche des Vorgängerbeispiels um eine zweite Schaltfläche.

VB

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        If TypeOf kunde1 Is IPerson Then
            MessageBox.Show("Das Objekt kunde1 hat die Schnittstelle IPerson implementiert!")
        End If
    End Sub
```

Ergebnis



3.11.4 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 3.50: Eine Klasse implementiert zwei Schnittstellen

VB
```
Public Class CPrivatkunde
    Implements IPerson, IKunde
    ...
End Class
```

3.11.5 Schnittstellenprogrammierung ist ein weites Feld

... und bis jetzt haben wir nur an der Oberfläche gekratzt. Wichtige Prinzipien hier nochmals in Kürze:

- Anstatt 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).

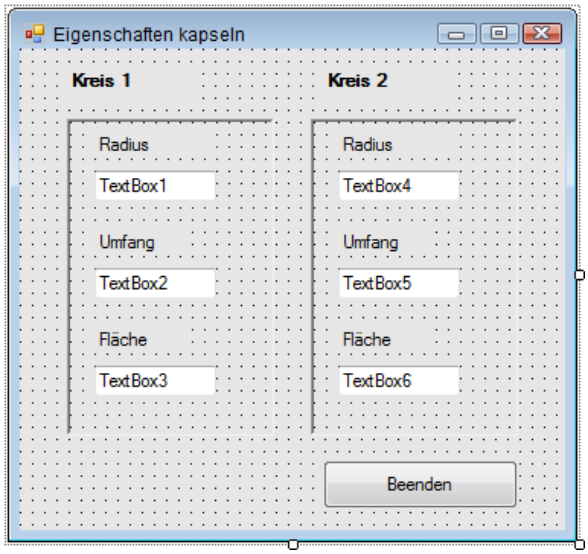
3.12 Praxisbeispiele

3.12.1 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 Beispiel handelt es sich um eine Klasse *CKreis* mit den Eigenschaften *Radius*, *Umfang* und *Fläche*. Diese Klasse speichert intern eine einzige Zustandsvariable *radZ*, aus welcher direkt beim Zugriff alle Eigenschaften berechnet werden.

Oberfläche

Um einen weiteren Vorteil der OOP zu demonstrieren, d.h. ohne viel Mehraufwand beliebig viele Instanzen aus einer Klasse bilden, 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.vb*.

```
Public Class CKreis
```

Die private Zustandsvariable speichert den Wert des Radius:

```
    Private radZ As Double
```

Die Eigenschaft *Radius* (Rahmencode wird automatisch erzeugt!):

```
    Public Property Radius() As String
        Get
            Return radZ.ToString("#,#0.00")
        End Get
        Set(value As String)
            If value <> String.Empty Then
                radZ = CDbl(value)
            Else
                radZ = 0
            End If
        End Set
    End Property
```

Die Eigenschaft *Umfang*:

```
    Public Property Umfang() As String
        Get
            Return (2 * Math.PI * radZ).ToString("#,#0.00")
        End Get
        Set(value As String)
            If value <> String.Empty Then
                radZ = CDbl(value) / 2 / Math.PI
            Else
                radZ = 0
            End If
        End Set
    End Property
```

Die Eigenschaft *Fläche*:

```
    Public Property Fläche() As String
        Get
            Return (Math.PI * Math.Pow(radZ, 2)).ToString("#,#0.00")
        End Get
        Set(value As String)
            If value <> String.Empty Then
                radZ = Math.Sqrt(CDbl(value) / Math.PI)
            Else
                radZ = 0
            End If
        End Set
    End Property
End Class
```

Wechseln Sie nun in den Klassencode von *Form1*.

Quellcode Form1

```
Public Class Form1
```

Ein Objekt wird erzeugt:

```
    Private Kreis1 As New CKreis()
```

Die folgenden Event-Handler sind einfach und übersichtlich, da die Objekte die inneren Funktionalitäten wegkapseln. Den Radius ändern:

```
    Private Sub TextBox1_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox1.KeyUp
        With Kreis1
            .Radius = TextBox1.Text
            TextBox2.Text = .Umfang
            TextBox3.Text = .Fläche
        End With
    End Sub
```

Den Umfang ändern:

```
Private Sub TextBox2_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox2.KeyUp
    With Kreis1
        .Umfang = TextBox2.Text
        TextBox1.Text = .Radius
        TextBox3.Text = .Fläche
    End With
End Sub
```

Die Fläche ändern:

```
Private Sub TextBox3_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox3.KeyUp
    With Kreis1
        .Fläche = TextBox3.Text
        TextBox1.Text = .Radius
        TextBox2.Text = .Umfang
    End With
End Sub
```

Der Code für *Kreis2* ist analog aufgebaut und braucht deshalb hier nicht wiederholt zu werden (siehe Beispieldaten).

```
End Class
```

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. Aus Gründen der Übersichtlichkeit wurde aber auf das Abfangen weiterer Eingaben, die sich nicht in einen numerischen Wert konvertieren lassen, verzichtet.

HINWEIS: Geben Sie als Dezimaltrennzeichen immer das Komma (,) ein, als Tausender-Separator dürfen Sie den Punkt (.) verwenden.

Objektinitialisierer

Man kann ein Objekt auch dann erzeugen und initialisieren, wenn es dazu keinen Konstruktor gibt. Sie könnten also im Code von *Form1* die Instanziierung der Klasse durch direktes Zuweisen ihrer Eigenschaften wie folgt vornehmen:

```
Private Kreis1 As New CKreis With {.Radius = "1.0"}
```

HINWEIS: Mehr zu Objektinitialisierern erfahren Sie im Abschnitt 3.8.2!

3.12.2 Eine statische Klasse anwenden

Als "statisch" wollen wir hier solche Klassen bezeichnen, die lediglich statische (*Shared*) Mitglieder haben. Solche Klassen eignen sich beispielsweise ideal für Formelsammlungen (siehe *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).

$V = 4/3 * Pi * r^3$

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 *Form*ular mit zwei *TextBox*en zur Eingabe von Kugeldurchmesser und Kugelvolumen ist erforderlich (siehe Laufzeitansicht).

Quellcode CKugel

Statische Funktionen werden mit dem Schlüsselwort *Shared* gekennzeichnet.

```
Public Class CKugel
    Public Shared Function Durchmesser_Volumen(durchmesser As String) As Double
        Dim dur As Double = System.Double.Parse(durchmesser)
        Dim vol As Double = Math.Pow(dur, 3) * Math.PI / 6.0
        Return vol
    End Function

    Public Shared Function Volumen_Durchmesser(volumen As String) As Double
        Dim vol As Double = System.Double.Parse(volumen)
        Dim dur As Double = Math.Pow(6 / Math.PI * vol, 1 / 3.0)
        Return dur
    End Function
End Class
```

Quellcode Form1

Die Verwendung der Klasse im Formularcode:

```
Public Class Form1
```

Die Berechnung startet nach Betätigen der *Enter*-Taste:

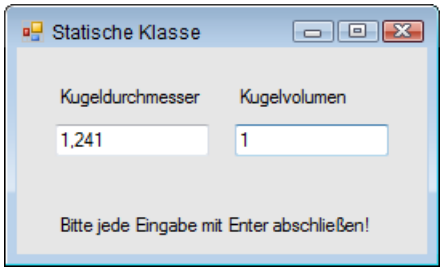
```
 Private Sub TextBox1_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox1.KeyUp
        If e.KeyCode = Keys.Enter And TextBox1.Text <> String.Empty Then
            TextBox2.Text = CKugel.Durchmesser_Volumen(TextBox1.Text).ToString("#,##0.000")
        End If
    End Sub
```

```
    Private Sub TextBox2_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox2.KeyUp
        If e.KeyCode = Keys.Enter And TextBox2.Text <> String.Empty Then
            TextBox1.Text = CKugel.Volumen_Durchmesser(TextBox2.Text).ToString("#,##0.000")
        End If
    End Sub
End Class
```

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:



3.12.3 Vom fetten zum dünnen Client

Lassen Sie sich durch den martialischen Titel nicht irritieren, wir wollen damit lediglich Ihr Interesse für eine Methodik wecken, die Ihnen hilft, den Horizont herkömmlicher Programmiertechniken zu überschreiten und damit einen leichteren Zugang zur OOP ermöglicht. Dabei gehen wir von folgender Erfahrung aus:

Typisch für den OOP-Ignoranten ist, dass er getreu der Devise "Hauptsache es funktioniert" mit Ausdauer und Beharrlichkeit immer wieder so genannte "fette" Clients (Fat Clients) programmiert. In einem solchen *Fat Client* ist in der Regel die gesamte Intelligenz (Geschäfts- bzw. Serverlogik) der Anwendung konzentriert, d.h., eine Aufteilung in Klassen bzw. Schichten hat nie stattgefunden.

Ein qualifizierter objektorientierter Entwurf zeichnet sich aber dadurch aus, dass der Client möglichst "dumm" bzw. "dünn" ist. Ein *Thin Client* verwaltet ausschließlich das User-Interface, die Aufgaben beschränken sich auf die Entgegennahme der Benutzereingaben und deren Weiterleitung an die Geschäftslogik bzw. umgekehrt auf die Ausgabe und Anzeige der von der Geschäftslogik ermittelten Ergebnisse.

Der Server hingegen umfasst die Geschäftslogik und kapselt damit die gesamte Intelligenz der Anwendung.

Die Vorteile einer solchen mehrschichtigen "Thin Client"-Strategie sind:

- gesteigerte Übersichtlichkeit und leichte Wiederverwendbarkeit der Software,
- Realisierung als verteilte Anwendung im Netzwerk ist möglich,
- Wartbarkeit und Erweiterbarkeit der Geschäftslogik sind möglich, ohne dass die Clients geändert werden müssten.

In unserem zweiteiligen Beispiel geht es um einen einfachen "Taschenrechner", den wir in zwei Versionen realisieren wollen.

In unserer ersten Windows Forms-Anwendung haben wir es mit einem Musterbeispiel für einen "fetten" Client zu tun. Im zweiten Teil verwandeln wir das Programm in eine mehrschichtige Anwendung mit einem "dünnen" Client. Neugierig geworden?

Oberfläche

So oder ähnlich sollte unser "Rechenkünstler" in der Entwurfsansicht aussehen:

Quellcode (Fat Client)

```
Public Class Form1
```

Über die Bedeutung der folgenden drei globalen Zustandsvariablen brauchen wir wohl keine weiteren Worte zu verlieren:

```
    Private op As Char                  ' aktueller Operator   (+, - , *, /)
    Private reg1 As String = Nothing    ' erstes Register   (Operand)
    Private reg2 As String = Nothing    ' zweites Register  (Operand)
```

Wir wollen zur Steuerung des Programmablaufs eine spezielle Variable *state* verwenden, die den aktuellen Zustand speichert:

```
Private state As Byte = 1           ' aktuelles Register (1 oder 2)
```

Typisch für die nun folgenden Ereignisbehandlungen ist, dass die durchgeführten Aktionen vom Wert der Zustandsvariablen *state* abhängig sind.

Zur Eingabe einer Ziffer (0...9) benutzt der gesamte Ziffernblock eine gemeinsame Ereignisbehandlung:

```
Private Sub ButtonZ_Click(sender As Object, e As EventArgs)  Handles Button1.Click,
     Button2.Click, Button3.Click, Button4.Click, Button5.Click, Button6.Click,
        Button7.Click, Button8.Click, Button9.Click, Button10.Click, Button11.Click
    Dim cmd As Button = CType(sender, Button)
    Select Case state
        Case 1                ' zum ersten Operanden hinzufügen:
            reg1 &= cmd.Text.Chars(0)
            Label1.Text = reg1
        Case 2                ' zum zweiten Operanden hinzufügen:
            reg2 &= cmd.Text.Chars(0)
            Label1.Text = reg1 & " " & op & " " & reg2
    End Select
End Sub
```

Für die Eingabe der Operation (+, -, *, /) wird ähnlich verfahren:

```
Private Sub ButtonOp_Click(sender As Object, e As EventArgs) _
              Handles ButtonAdd.Click, ButtonSub.Click, ButtonMult.Click, ButtonDiv.Click
    Dim cmd As Button = CType(sender, Button)
    Select Case state
        Case 1
            op = cmd.Text.Chars(0)      ' neuer Operand ...
            state = 2                   ' ... und Zustandswechsel
        Case 2
            ergebnis()        ' erst Zwischenergebnis mit altem Operand ermitteln ...
            op = cmd.Text.Chars(0)               ' ... dann neuer Operand
     End Select
     Label1.Text = reg1.ToString & " " & op
     reg2 = Nothing                              ' Reg2 löschen
 End Sub
```

Die folgende Hilfsprozedur führt die Rechenoperation aus und speichert deren Ergebnis in *reg1*:

```
 Private Sub ergebnis()
    Dim r1 As Double = Convert.ToDouble(reg1)
    Dim r2 As Double = Convert.ToDouble(reg2)
    Select Case op
        Case "+"c
            reg1 = (r1 + r2).ToString
        Case "-"c
            reg1 = (r1 - r2).ToString
        Case "*"c
            reg1 = (r1 * r2).ToString
        Case "/"c
```