walter DOBERENZ
thomas GEWINNUS

Visual Basic 2015

GRUNDLAGEN
PROFIWISSEN
REZEPTE

// Visual Basic-Grundlagen
// LINQ, OOP, ADO.NET
// App-Entwicklung
// Über 150 Praxisbeispiele

EXTRA: 700 Seiten Bonuskapitel zu WPF und Windows Forms

HANSER

Doberenz/Gewinnus

Visual Basic 2015
Grundlagen, Profiwissen und Rezepte

Walter Doberenz
Thomas Gewinnus

Visual Basic 2015

Grundlagen, Profiwissen und Rezepte

HANSER

Die Autoren:

Professor Dr.-Ing. habil. Walter Doberenz, Wintersdorf
Dipl.-Ing. Thomas Gewinnus, Frankfurt/Oder

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

Lektorat: Sieglinde Schärl
Herstellung: Irene Weilhart
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

Print-ISBN: 978-3-446-44380-8
E-Book-ISBN: 978-3-446-44605-2

Inhaltsverzeichnis

Teil II: Technologien

Anhang

Download-Kapitel

LINK: http://doko-buch.de

Teil IV: WPF-Anwendungen

Teil V: Windows Forms

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 und 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 2015 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 2015 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 2015 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 2015 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.

Die Kapitel des Buchs haben wir in fünf Themenkomplexen gruppiert:

1. Grundlagen der Programmierung mit VB.NET
2. Technologien
3. Windows Store Apps
4. WPF-Anwendungen
5. Windows Forms-Anwendungen

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 drei Themenkomplexe, denn bereits hier sind wir an die Grenze des drucktechnisch Machbaren gestoßen. Die übrigen zwei Themenkomplexe mussten wir als PDF auslagern, welche Sie sich kostenlos aus dem Internet herunterladen können.

Zu den Codebeispielen

Alle Beispieldaten dieses Buchs und die Kapitel des vierten und fünften Teils 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 2015 unter Windows 8 bzw. 10 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 1900 Seiten – in diesem Buch nicht alles finden, was Visual Basic 2015 bzw. das .NET Framework 4.6 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 August 2015*

Teil I: Grundlagen

- **Einstieg in Visual Studio 2015**
- **Grundlagen der Sprache Visual Basic**
- **Objektorientiertes Programmieren**
- **Arrays, Strings und Funktionen**
- **Weitere wichtige Sprachfeatures**
- **Einführung in LINQ**

Kapitel 1

Einstieg in Visual Studio 2015

Dieses Kapitel bietet dem VB-Programmierer einen effektiven Schnelleinstieg in die Arbeit mit Visual Studio 2015. 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 2015 gegenüber der Vorgängerversion Visual Studio 2012.

1.1 Die Installation von Visual Studio 2015

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 2015.

1.1.1 Überblick über die Produktpalette

Alle im Handel angebotenen Visual-Studio-Pakete basieren auf dem .NET-Framework 4.6. 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 Community 2015

Bei dieser kostenfreien Version handelt es sich bereits um eine vollständig ausgestattete Entwicklungsumgebung für Windows-Desktop-, Web- und plattformübergreifenden iOS-, Android- und Windows-Applikationen.

Sie kann auch für kommerzielle Projekte eingesetzt werden, wenn es sich dabei um Unternehmen mit weniger als 250 Mitarbeitern handelt.

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

Visual Studio Professional 2015

Wie es der Name bereits suggeriert, handelt es sich bei diesem Standard-Paket bereits um ein professionelles Werkzeug zur Entwicklung beliebiger Anwendungstypen im Team:

- Mit *CodeLens* ist ein leistungsstarkes Features zum Verbessern der Produktivität Ihres Teams enthalten
- Verschiedenen Planungstools (Agile-Projekte, Teamräume, Diagramme, ...) dienen der Verbesserung der Team-Produktivität
- Mit bestimmten MSDN-Abonnementleistungen erhalten Sie Zugang zu nützlicher Software für Entwicklung/Tests, Team Foundation Server, Visual Studio Online Basic ...

Visual Studio Enterprise 2015

Hier handelt es sich um die Vollausstattung für Softwareentwickler, die im Team Anwendungen auf Enterprise-Niveau erstellen wollen. Neben allen Features der Professional-Version sind auch weitere Funktionen enthalten, die eine komplexe Datenbankentwicklung und eine durchgängige Qualitätssicherung ermöglichen sollen.

HINWEIS: Visual Studio Enterprise ersetzt die bisherigen Editionen Premium und Ultimate!

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 2012. Die folgende Auflistung kann lediglich eine Orientierungshilfe sein:

- Betriebssystem: Windows 10, 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-Laufwerk

Die Parameter von Prozessor und RAM sind als untere Grenzwerte zu verstehen.

Ganz wichtig:

HINWEIS: Wollen Sie Apps für Windows 8/10 entwickeln, so ist das Betriebssystem Windows 8 bzw. 10 für das Entwicklungssystem unerlässlich!

Weiterhin ist zu beachten:

- Das .NET Framework ab 4.5 wird von Windows XP nicht mehr unterstützt – motten Sie also Ihren alten Computer ein.
- Das .NET-Framework 3.5 ist standardmäßig nicht mehr in Windows 8 bzw. 10 enthalten, es muss nachinstalliert werden (*Systemsteuerung/Programme und Features/Windows Features aktivieren ...*) oder die Anwendungen müssen auf die Version 4 aktualisiert werden.
- Der SQL Server Express ist nicht mehr im Installationspaket enthalten, sondern muss separat heruntergeladen werden. Alternativ steht nach der Installation von Visual Studio der SQL Server Express 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 Visual Studio, 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

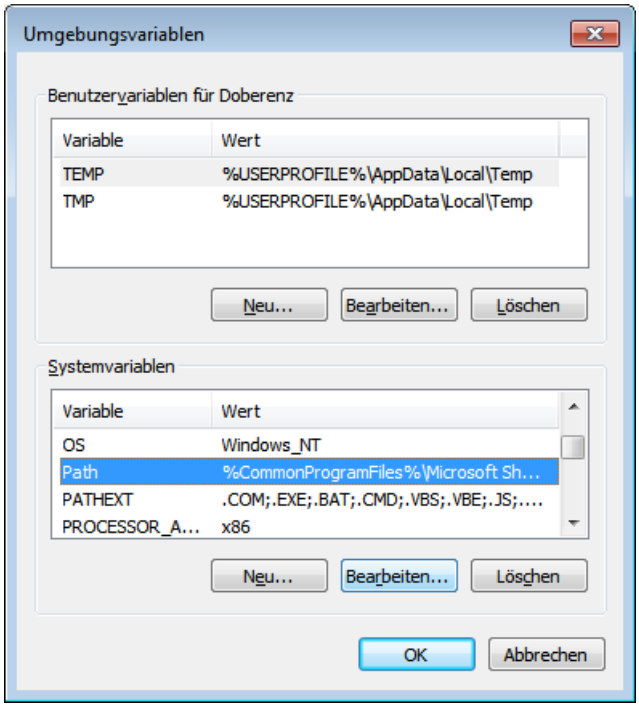
Der VB-Compiler *vbc.exe* befindet sich, ziemlich versteckt, im Verzeichnis

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

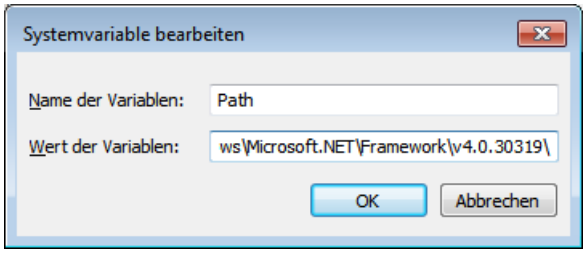
Da das Kompilieren direkt an der Kommandozeile ausgeführt werden soll, werden wir *vbc.exe* in den Windows-Pfad aufnehmen, um so seinen Aufruf von jedem Ordner des Systems aus zu ermöglichen:

- 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.

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

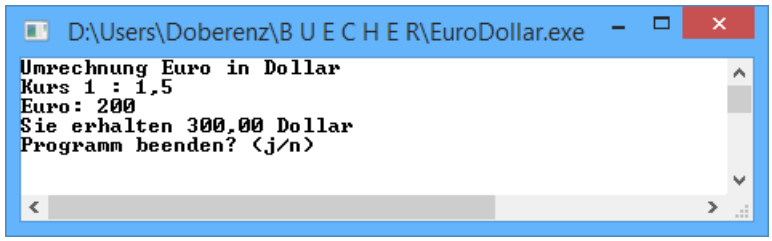
Um bequem an der Kommandozeile arbeiten zu können, kopieren Sie zunächst die Datei *cmd.exe* (Eingabeaufforderung aus *...\Windows\System32*) in dasselbe Verzeichnis, in welchem sich auch die Datei *EuroDollar.vb* befindet.

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 out

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 für verschiedenste Testzwecke, hat dieser einfache Anwendungstyp aber durchaus noch seine Daseinsberechtigung.

Mit Visual Studio 2015 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 2015

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

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

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.

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 2015* zu einer hocheffektiven Entwicklungsumgebung für Windows- und Web-Anwendungen.

1.4 Die Entwicklungsumgebung Visual Studio 2015

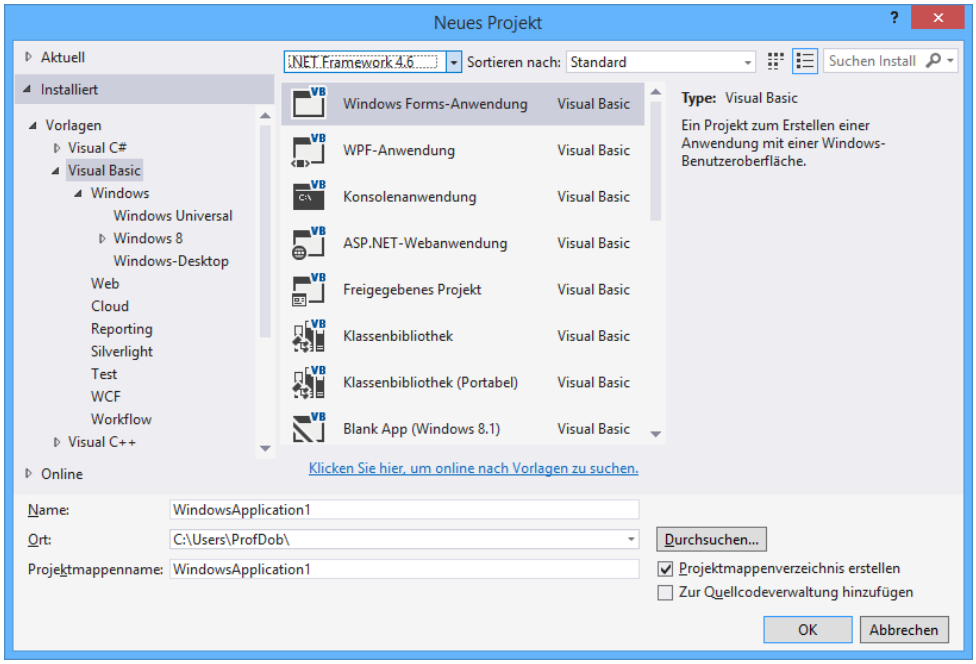
Visual Studio 2015 ist eine universelle Entwicklungsumgebung (IDE[1]) für Windows- und für Web-Anwendungen, die auf Microsofts .NET-Technologie basieren. Alle notwendigen Tools, wie z.B. für den visuellen Oberflächenentwurf, für die Codeprogrammierung und für die Fehlersuche, werden bereitgestellt.

Visual Basic ist nur eine der möglichen objektorientierten Sprachen, die Sie unter Visual Studio 2015 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 Neues Projekt

Wählen Sie auf der Startseite von Visual Studio 2015 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.



[1] *Integrated Developers Environment*

[2] Die Abbildung bezieht sich auf die Community-Edition.

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

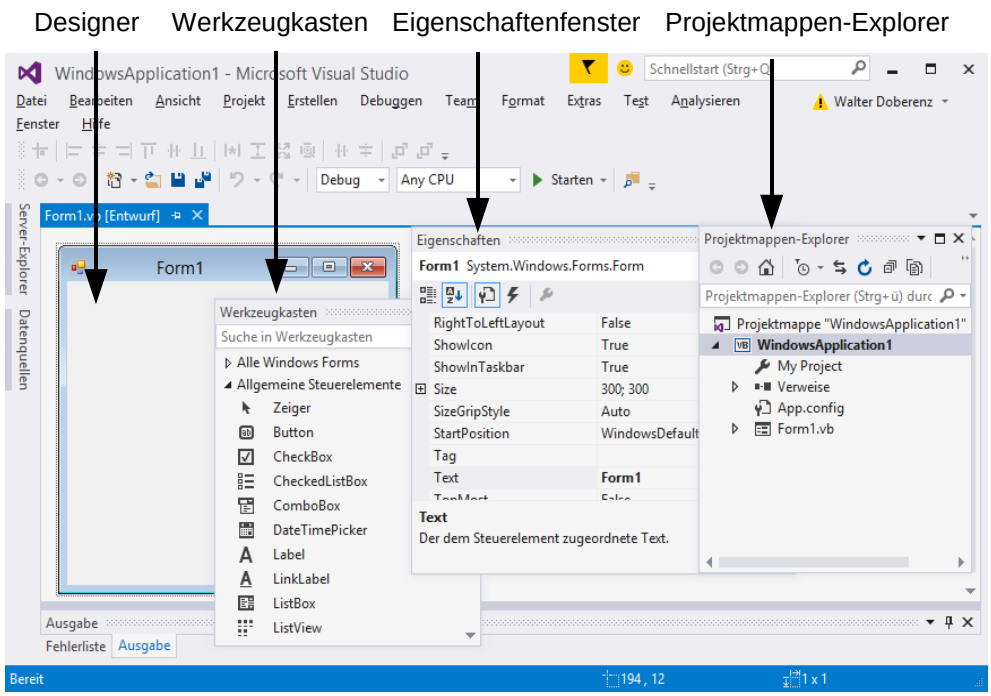
Haben Sie das Häkchen bei *Projektmappenverzeichnis erstellen* gesetzt, so erzeugt Visual Studio automatisch einen Unterordner mit dem Namen des Projekts in dem als Speicherort eingetragenen Hauptverzeichnis.

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, 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-Forms-Anwendungen.

Das Eigenschaften-Fenster

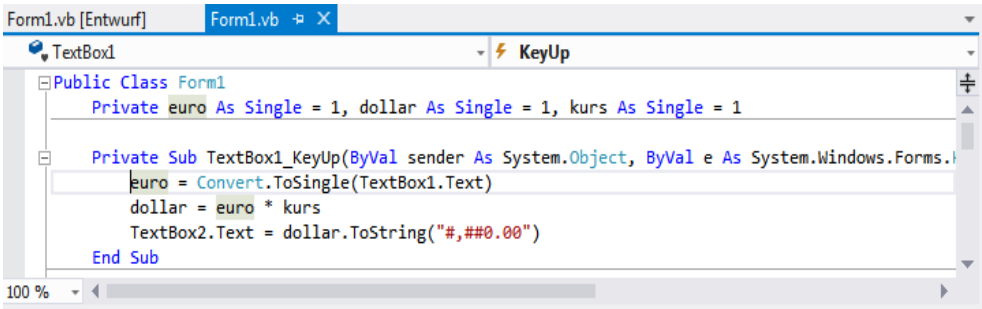
Im Eigenschaften-Fenster (Menü *Ansicht/Eigenschaftenfenster* bzw. *F4*) werden die zur Entwurfszeit editierbaren Eigenschaften des gerade aktiven Steuerelements aufgelistet[1]. Normalerweise hat jede Eigenschaft bereits einen Standardwert, den Sie in vielen Fällen übernehmen können.

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

[1] 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.

Das Codefenster

Für die eigentliche Programmierung ist das Codefenster zuständig. Logischerweise wird dies damit auch zu Ihrem Hauptbetätigungsfeld als VB-Programmierer. Um zum Beispiel das zu *Form1* (siehe obige Abbildung) gehörige Codefenster zu öffnen, klicken Sie auf den Designer von *Form1* mit der rechten Maustaste und wählen im Kontextmenü *Code anzeigen* (F7). Die folgende Abbildung zeigt das Codefenster für *Form1* des zweiten Praxisbeispiels am Kapitelende.



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 Verteilbar-

keit 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.

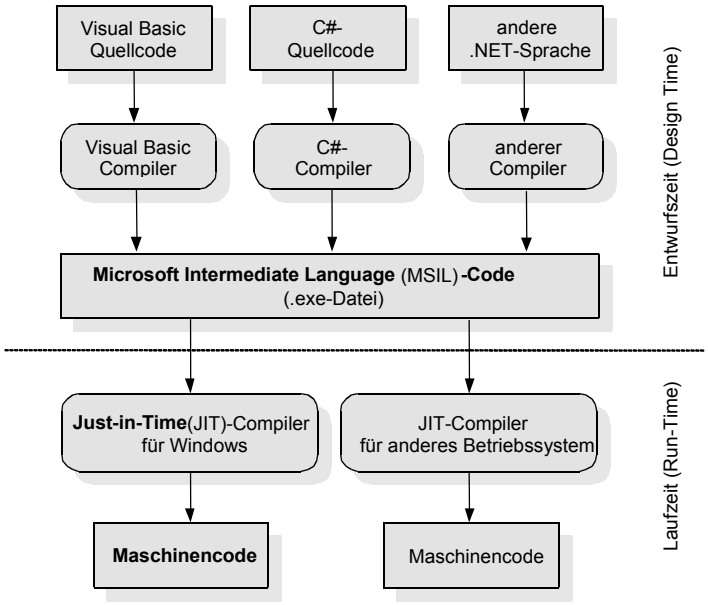
Wie funktioniert eine .NET-Sprache?

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.

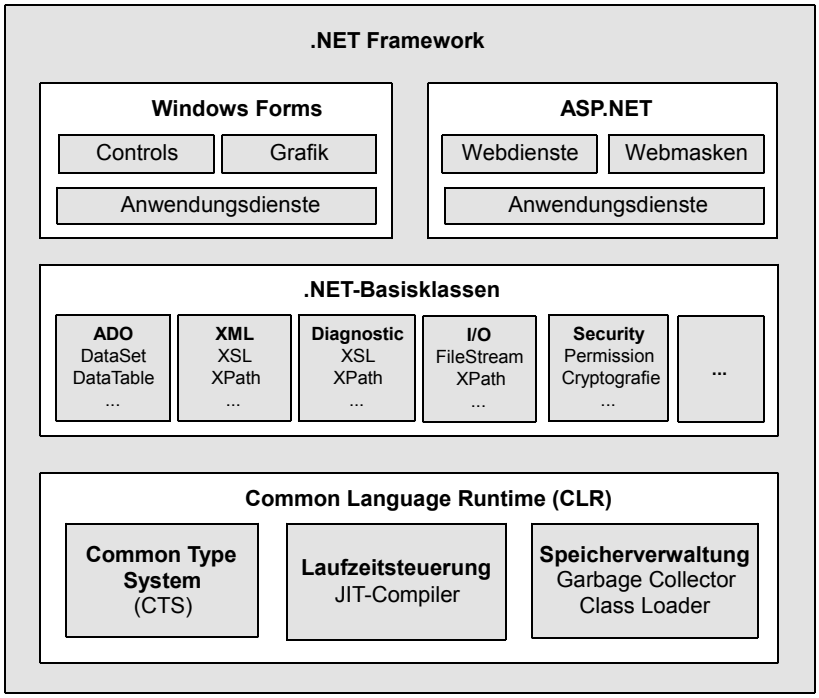
1.5.2 .NET-Features und Begriffe

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.



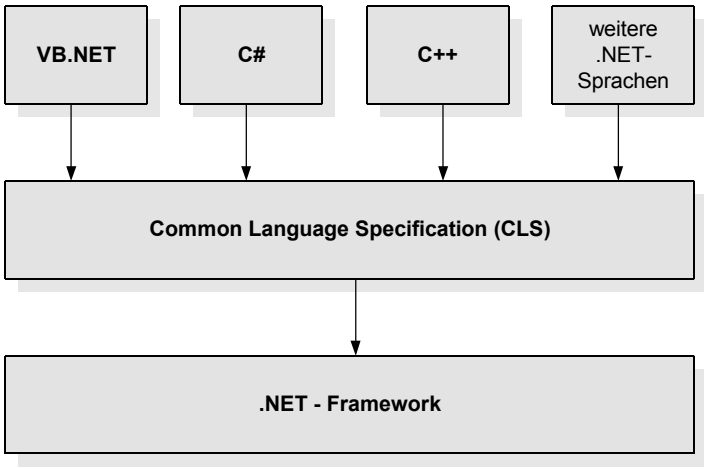
Zu den wichtigsten Komponenten des .NET-Frameworks und den damit zusammenhängenden Begriffen zählen:

- 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.

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

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 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 verzichten kann. Metadaten können daher nicht versehentlich verloren gehen oder mit einer falschen Dateiversion kombiniert werden.

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

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

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

Auch Entwicklungswerkzeuge wie Microsoft Visual Studio verwenden die Reflexion, um z.B. den Mechanismus der IntelliSense zu implementieren. Sobald Sie einen Methodennamen eintippen, zeigt die IntelliSense eine Liste mit den Parametern der Methode an oder 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 2015

Für alle Leser, die bereits mit einer Vorgängerversion gearbeitet haben, dürften die folgenden Ausführungen von Interesse sein. Neueinsteiger können diesen Abschnitt überspringen.

1.6.1 Entwicklungsumgebung

Hier möchten wir vor allem auf zwei Neuerungen verweisen:

- Zum Erstellen von leeren, freigegebenen Projekten gibt es neue Vorlagen. Auf diese freigegebenen Projekte kann jetzt durch verschiedene andere Projekttypen verwiesen werden (Konsolenanwendungen, Klassenbibliotheken, Windows Forms-Apps, Windows Universal Apps, Windows Store 8.1, Windows Phone 8.1, ...).
- Benutzerdefinierte Fensterlayouts lassen sich jetzt speichern, indem Sie im Menü *Fenster* auf *Fensterlayout speichern* klicken. Auch das Löschen, Umbenennen und Neuanordnen von Layouts ist möglich (Menü *Fenster / Fensterlayout verwalten*).
- Der Debugger hat ein neues Design und außerdem einige zusätzliche Funktionalitäten (Lambda-Support, Watch-Fenster).

1.6.2 Neue VB-Sprachfeatures

Die Liste der Neuerungen fällt relativ bescheiden aus und beschränkt sich auf einige Spezialfälle:

- Das Beschreiben von Formatstrings lässt sich mittels Stringinterpolation vereinfachen
- Nullbedingte Operatoren wurden eingeführt
- Kommentare sind jetzt auch in Fortsetzungszeilen möglich
- Zeichenkettenliterale sind auch über mehrere Zeilen erlaubt
- Um den Namen eines Parameters, Members oder Typs als Zeichenfolge zu gewinnen, gibt es mit der *NameOf*-Funktion jetzt einen typsicheren Weg
- Es gibt auch selbst implementierende Read-Only Eigenschaften
- Nicht nur Formulare und Klassen, sondern jetzt auch Module und Interfaces können *Partial* deklariert werden

1.6.3 Code-Editor

Erst auf den zweiten Blick sieht man, dass die Benutzeroberfläche des Code-Editors etwas aufpoliert wurde:

- Das Glühbirnen-Symbol vereinfacht einige Aktionen, wie die Beseitigung von Tippfehlern und das Umgestalten von Code. Es werden Vorschläge zur Problemlösung angezeigt.

- Beim Umbenennen werden jetzt alle Instanzen des umzubenennenden Bezeichners hervorgehoben, der neue Namen des Bezeichners braucht nur einmal im Editor eingegeben zu werden
- Die Auswertung von Ausdrücken wurde verbessert. Auch LINQ- und Lambda-Ausdrücke werden jetzt im Überwachungs- und im Direktfenster unterstützt.

1.6.4 NET Framework 4.6

Zu dieser neuesten Version von .NET Framework wurden einige neue APIs hinzugefügt, um vor allem plattformübergreifende Szenarien zu ermöglichen:

- Neue Kryptografie-APIs, wie etwa *AsymmetricAlgorithm.KeyExchangeAlgorithm*, *AsymmetricAlgorithm.SignatureAlgorithm*
- Das Feature[1] "Ändern der Größe von Windows Forms-Steuerelementen" umfasst jetzt auch die *System.Windows.Forms*-Typen *DomainUpDown*, *NumericUpDown*, *DataGridViewComboBoxColumn*, *DataGridViewColumn* und *ToolStripSplitButton*
- Ein *EventSource*-Objekt kann jetzt direkt konstruiert werden, Sie können eine der *Write()*-Methoden aufrufen, um ein sich selbst beschreibendes Ereignis abzugeben

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 2015 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 2015* erfolgt entweder über das Windows-Startmenü oder schneller über eine vorher eingerichtete Desktop- bzw. Taskleisten-Verknüpfung.

[1] Zur Aktivierung müssen Sie *EnableWindowsFormsHighDpiAutoResizing* in der *app.config* auf *True* setzen.

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).

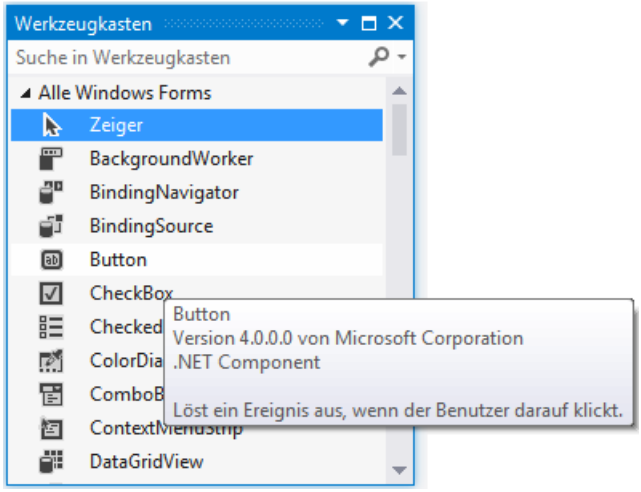
Im Mittelteil klicken Sie auf *Windows Forms-Anwendung* (ganz oben in der Liste). Nehmen Sie im unteren Teil die folgenden Einträge vor bzw. belassen es bei den Standardvorgaben:

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

HINWEIS: Im Dialogfenster rechts oben sehen Sie, dass Sie mit Visual Studio 2015 sowohl Projekte für das .NET Framework 4.6 als auch für die Vorgängerversionen 4.5.2, 4.5.1. 4.5, 3.0 und sogar 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.

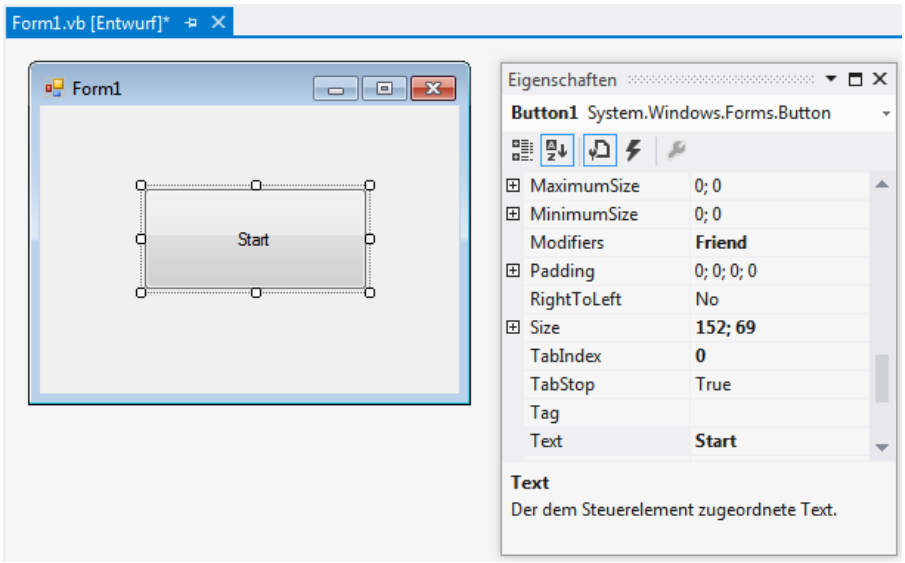


Ein schneller Doppelklick befördert das Steuerelement direkt auf das Formular. Sie können aber auch den gewünschten *Button* erst mit einem einfachen Mausklick im Werkzeugkasten aktivieren, um ihn anschließend auf dem Formular abzusetzen und auf die gewünschte Größe zu zoomen.

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".

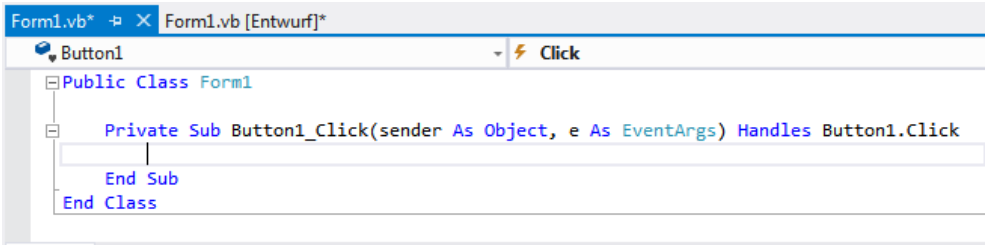
HINWEIS: Verwechseln Sie die *Text*-Eigenschaft nicht mit der *Name*-Eigenschaft. Wenn Sie ein neues Steuerelement platzieren, setzt Visual Studio standardmäßig den Wert von *Text* zunächst auf den von *Name*.



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, 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 mehr sein, sondern ein durchaus nützliches Prográmmchen – 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.

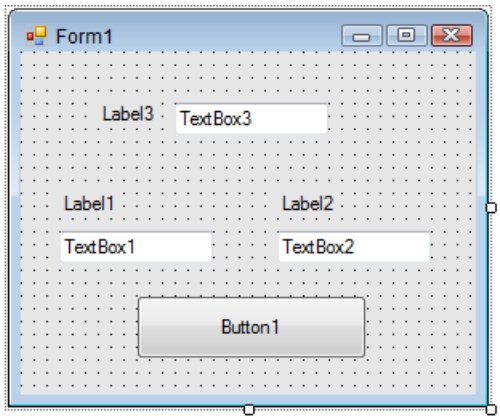
1. Etappe: Visueller Entwurf der Bedienoberfläche

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)[1].

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

HINWEIS: Konzentrieren Sie sich in der ersten Etappe nur auf Lage und Abmessung der 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.

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

2. Etappe: Zuweisen der Objekteigenschaften

Unser Progrämmchen 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:

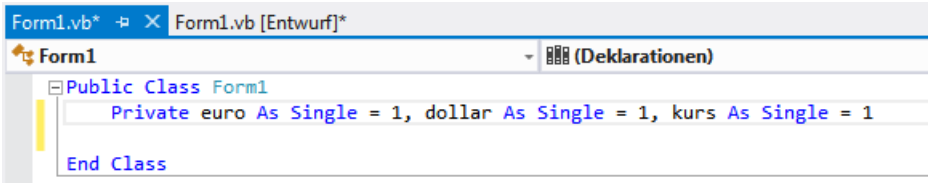
Name des Objekts	Eigenschaft	Neuer Wert
Form1	*Text* *Font.Size*	Umrechnung Euro-Dollar *10*
Label1	*Text*	Euro
Label2	*Text*	Dollar
Label3	*Text*	Kurs 1:
TextBox1	*TextAlign*	*Right*
TextBox2	*TextAlign*	*Right*
TextBox3	*TextAlign*	*Center*
Button1	*Text*	Beenden

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!

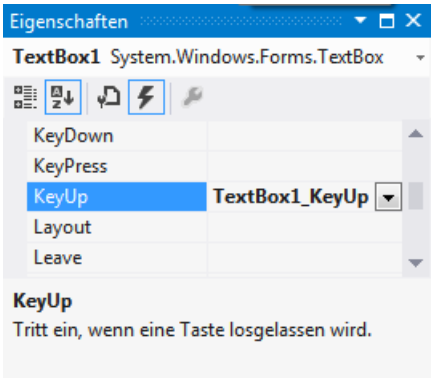
Dabei ist eine fast schon rituelle Erstellungsreihenfolge zu beachten, wie Sie sie mit fortschreitender Programmierpraxis sehr bald auch im Schlaf ausführen können:

- **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:

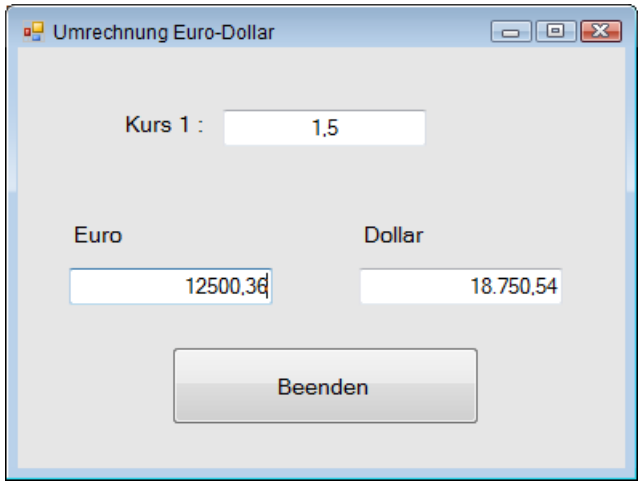
```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Me.Close()
End Sub
```

4. Etappe: Programm kompilieren und testen

Klicken Sie auf den Starten-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.

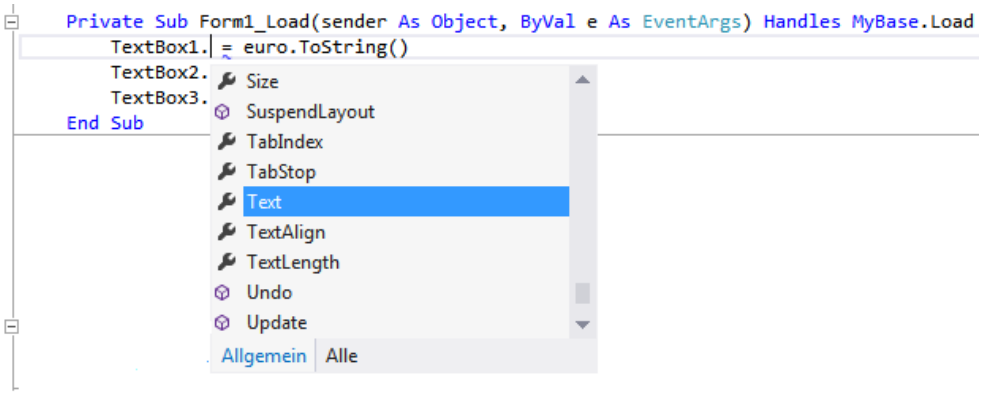


Da wir aus Gründen der Einfachheit in diesem Programm auf eine Überprüfung der Eingabewerte (Eingabevalidierung) verzichtet haben, sollten Sie auf das komplette Löschen des Inhalts eines Textfelds verzichten und nur gültige Zahlenwerte eingeben (also z.B. auch keine Buchstaben). Andernfalls ist ein Konvertieren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

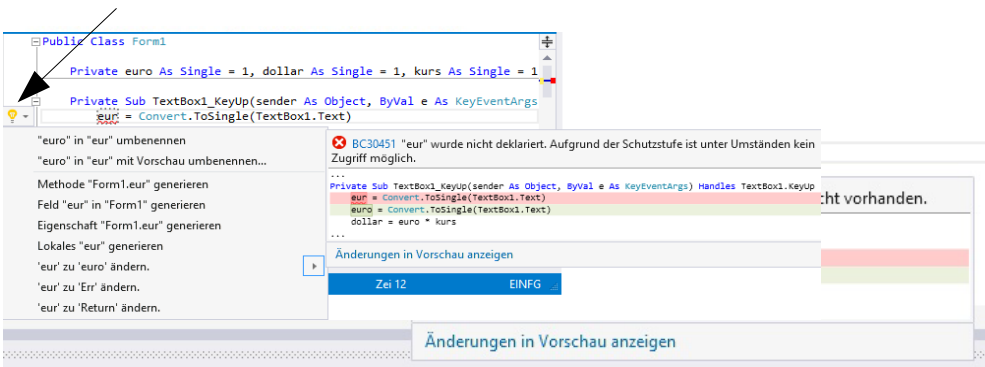


HINWEIS: Wenn Sie das markierte Element übernehmen wollen, brauchen Sie den Namen nicht zu Ende zu schreiben, da die IntelliSense den kompletten Text automatisch ergänzt.

Die Glühbirne sorgt für Erleuchtung

Bereits beim Schreiben des Quellcodes werden Sie von Visual Studio auf grundsätzliche Syntaxfehler (z.B. ein vergessenes Komma) hingewiesen, im Allgemeinen geschieht dies durch Unterstreichen mit einer wellenförmigen (roten) Linie. Wenn Sie mit der Maus auf diese Linie zeigen, erscheint links unterhalb das Symbol einer gelben Glühbirne[1] mit Hinweisen zur Behebung des Fehlers.

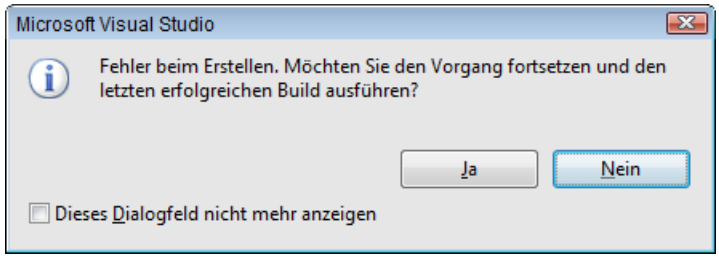
In der folgenden Abbildung wurde z.B. die Variable *euro* falsch eingetippt:



Fehlersuche

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

[1] Dieses Feature gehört zu den Neuigkeiten des Code-Editors von Visual Studio 2015.



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*).

HINWEIS: 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.

Seit VB 2015 sind Kommentare auch bei impliziter Zeilenfortführung erlaubt.

BEISPIEL 2.5: Kommentare bei impliziter Zeilenfortführung

VB
```
Dim mitarbeiter =  {"Müller",    ' Gruppenleiter
                    "Schulze",   ' Stellvertreter
                    "Lehman"}
```

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 2.6: 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 2.7: 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

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 2.8: 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 2.9: Eine weitere alternative Schreibweise für das Vorgängerbeispiel

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

BEISPIEL 2.10: 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 2.11: 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 302). 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 2.12: 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 2.13: 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 2.14: 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 2.15: 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 2.16: 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 2.16: Mehrfachdeklaration

dürfen Sie auch schreiben

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

Anfangswerte

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

BEISPIEL 2.17: 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 2.18: 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
```

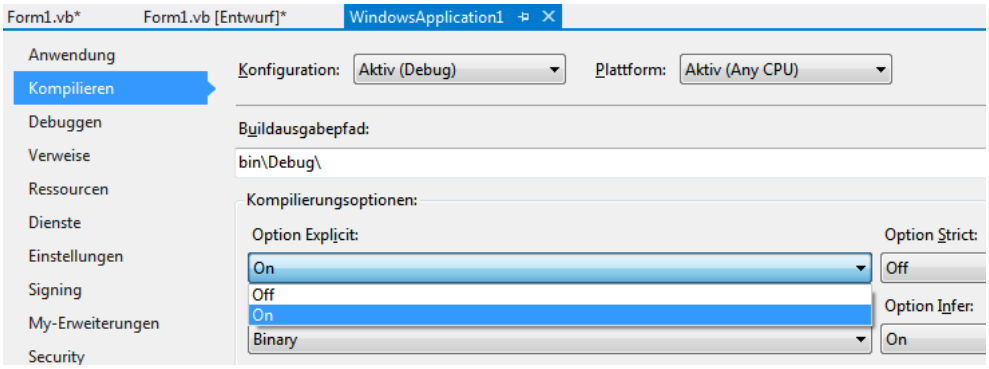
... aber so bemerken Sie Ihren Fehler nicht:

```
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 2015 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 2.19: 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 2.20: 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 2.21: 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 2.22: 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 2.23: 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 2.24: 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 2.25: 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 2.26: 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 2.27: 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 92) 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 2.28: 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 2.29: 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 2.30: 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 2.31: 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 2.32: 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 2.33: Addition von zwei Zeichenketten

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

VB 2015 ermöglicht nun auch Stringliterate die sich über mehrere Zeilen erstrecken.

BEISPIEL 2.34: Stringliterate über mehrere Zeilen

VB
```
Dim s = "Viele Grüße,
         Euer Hans"
```

Das sieht doch besser aus als die alte Schreibweise:

```
Dim s = "Viele Grüße," & vbCrLf & "Euer Hans"
```

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.

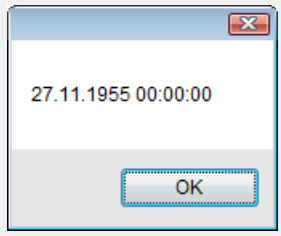
Ein *Date*-Literale muss durch das Rautenzeichen (#) eingeschlossen sein und im Format *m/d/yyyy* (US-amerikanische Schreibweise) angegeben werden.

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 2.35: 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.

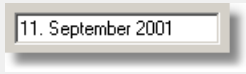
BEISPIEL 2.36: Der 17. August 1987, 19.00 Uhr

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 2.37: 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 2.38: 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 2.39: 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 2.40: 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 106).
- 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 2.41: 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
```

BEISPIEL 2.41: Drei explizite Typkonvertierungen mit dem gleichen Ergebnis.

```
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.

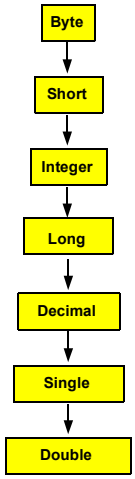
2.4.2 Welcher Datentyp passt zu welchem?

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

BEISPIEL 2.42: 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.

[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!

BEISPIEL 2.43: Umwandlung eines Zeichens in einen String

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

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 2.44: 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 2.45: 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 2.46: 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 2.47: 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 2.48: Wird ein *String* mit *CChar* in eine *Char*-Variable kopiert, so erhält diese das erstemodulglobal Zeichen des Strings.

VB
```
Dim s As String = "Hallo"
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 2.49: 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 2.50: 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 2.51: 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 2.52: 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 2.53: 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 2.54: 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 2.55: 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 2.56: 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 2.57: 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 2.58: *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 2.59: Der Inhalt einer *TextBox* wird in eine Gleitkommazahl konvertiert.

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

BEISPIEL 2.60: 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 2.61: 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 2.62: 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 2.63: 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 2.64: 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 2.65: 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 2.66: Einige Deklarationen von Nullable-Typen

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

BEISPIEL 2.67: 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 2.68: 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 2.69: 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 2.70: 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 2.71: 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 2.72: 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 2.73: 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 2.74: 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 2.75: Arithmetische Operationen

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

BEISPIEL 2.76: 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 2.77: 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 2.78: 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 2.79: 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 2.79: 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 2.80: 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 2.81: 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 2.82: 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 2.83: Zehnmal untereinander den laufenden Index und einen Text in einer *ListBox* ausgeben

VB

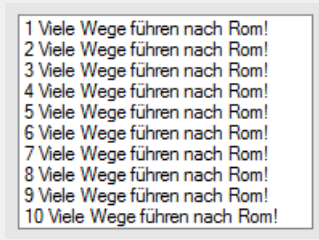
```
Dim i As Integer
For i = 1 To 10
       ListBox1.Items.Add(i.ToString & " Viele Wege führen nach Rom!")
Next i
```

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 2.83: 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 2.84: 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.