walter DOBERENZ
thomas GEWINNUS

Visual C# 2015

GRUNDLAGEN
PROFIWISSEN
REZEPTE

// C#-Grundlagen
// LINQ, OOP, ADO.NET
// App-Entwicklung
// Über 150 Praxisbeispiele

EXTRA: 700 Seiten Bonuskapitel zu WPF und Windows Forms

HANSER

Doberenz/Gewinnus

Visual C# 2015
Grundlagen, Profiwissen und Rezepte

Walter Doberenz
Thomas Gewinnus

Visual C# 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: 978-3-446-44381-5
E-Book-ISBN: 978-3-446-44606-9

Inhaltsverzeichnis

Teil II: Technologien

Teil III: Windows Apps

Anhang

Download-Kapitel

LINK: http://doko-buch.de

Teil IV: WPF-Anwendungen

Teil V: Windows Forms

Vorwort

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

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

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

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

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

Zum Buchinhalt

Wie Sie bereits dem Buchtitel entnehmen können, wagt das vorliegende Werk den Spagat zwischen einem Grundlagen- und einem Profibuch. Sinn eines solchen Buches kann es nicht sein, eine umfassende Schritt-für-Schritt-Einführung in Visual C# 2015 zu liefern oder all die Informationen noch einmal zur Verfügung zu stellen, die Sie in der Produktdokumentation (MSDN) ohnehin schon finden und von denen Sie in der Regel nur ein Mausklick oder die F1-Taste trennt.

- Für den *Einsteiger* wollen wir den einzig vernünftigen und gangbaren Weg beschreiten, nämlich nach dem Prinzip "so viel wie nötig" eine schmale Schneise durch den Urwald der .NET-

Programmierung mit Visual C# 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 C#
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 der folgenden 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.
- Um mit den WinRT-Projekten arbeiten zu können, müssen Sie Visual Studio unter Windows 8 bzw. 10 ausführen und das Windows SDK installiert haben.
- 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 C# 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 Autoren-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 C#**
- **Objektorientiertes Programmieren**
- **Arrays, Strings und Funktionen**
- **Weitere wichtige Sprachfeatures**
- **Einführung in LINQ**

Kapitel 1

Einstieg in Visual Studio 2015

Dieses Kapitel bietet Ihnen einen effektiven Schnelleinstieg in die Arbeit mit Visual Studio 2015. Gleich nachdem Sie die Hürden der Installation gemeistert haben, erstellen Sie mit C# 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 Nachvollzug 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.

1.1 Die Installation von Visual Studio 2015

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

HINWEIS: Voraussetzung für ein erfolgreiches Studium dieses Buchs ist ein Rechner mit einer lauffähigen Installation von Visual Studio 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 WinRT-Anwendungen für Windows 8 bzw. 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 8/10 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 C#-Programm

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

Auch allein mit dem .NET Framework SDK, also ohne Visual Studio 2015, könnte man (zumindest rein theoretisch) vollwertige Programme entwickeln. Das wollen wir jetzt unter Beweis stellen, indem wir eine kleine Euro-Dollar-Applikation als so genannte *Konsolenanwendung* – dem einfachsten Anwendungstyp – schreiben.

1.2.1 Vorbereitungen

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

Compilerpfad eintragen

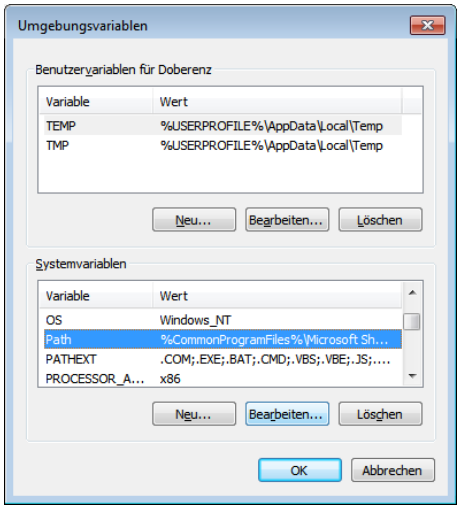
Der C#-Compiler *csc.exe* befindet sich, ziemlich versteckt, im Verzeichnis

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

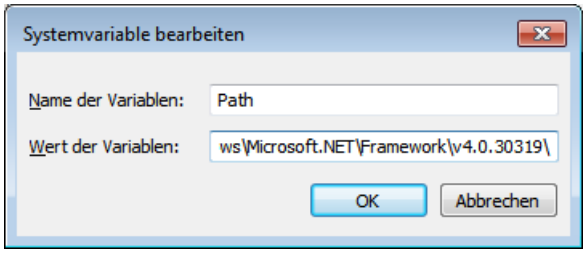
Da das Kompilieren direkt an der Kommandozeile ausgeführt werden soll, werden wir *csc.exe* in den Windows-Pfad aufnehmen, um so seinen Aufruf von jedem Ordner des Systems aus zu ermöglichen:

- Sie finden den Dialog zur Einstellung der *Path*-Umgebungsvariablen in der Windows-Systemsteuerung unter dem Eintrag *System* im Aufgabenbereich "Erweiterte Systemeinstellungen".

- Im Dialog *Systemeigenschaften* klicken Sie auf der Registerkarte *Erweitert* auf die Schaltfläche *Umgebungsvariablen...*.



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



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

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

```
csc /?
```

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

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

1.2.2 Quellcode schreiben

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

```
using System;
class KonsolenDemo
{
  static void Main()
  {
    int i;
    Console.WriteLine("Umrechnung Euro in Dollar");
    do
    {
       float kurs, euro, dollar;
       Console.Write("Kurs 1 : ");
       kurs = Convert.ToSingle(Console.ReadLine());
       Console.Write("Euro: ");
       euro = Convert.ToSingle(Console.ReadLine());
       dollar = euro * kurs;
       Console.WriteLine("Sie erhalten " + dollar.ToString("0.00 $"));
       Console.Write("Programm beenden? (j/n)");
       string s = Console.ReadLine();
       i = string.Compare(s, "j");
    } while(i != 0);
  }
}
```

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

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

1.2.3 Programm kompilieren und testen

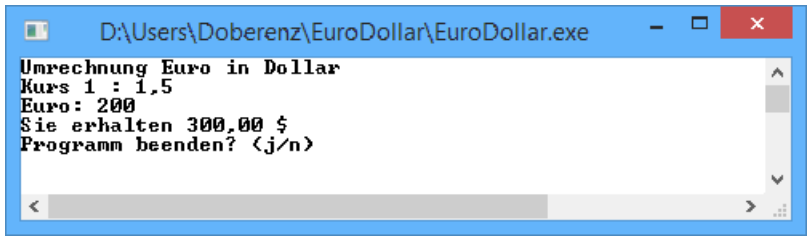
Um bequem an der Kommandozeile arbeiten zu können, kopieren Sie zunächst die Datei *cmd.exe* (Eingabeaufforderung aus *...\Windows\System32*) in dasselbe Verzeichnis, in welchem sich auch die Datei *EuroDollar.cs* befindet.

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

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

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

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



1.2.4 Einige Erläuterungen zum Quellcode

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

Befehlszeilen und Gültigkeitsbereiche

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

using-Anweisung

Mit der ersten Anweisung

```
using System;
```

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

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

nur noch

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

schreiben müssen.

class-Anweisung

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

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

WriteLine- und ReadLine-Methoden

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

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

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

Assemblierung

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

1.2.5 Konsolenanwendungen sind out

Zwar sind mit der Klasse *System.Console* auch verschiedenste farbliche Effekte möglich sind, trotzdem: Bei wem weckt das triste Outfit einer Konsolenanwendung noch positive Emotionen[1]?

Als einfaches Hilfsmittel zum Erlernen von C# und für verschiedene Testzwecke hat dieser einfache Anwendungstyp aber durchaus noch seine Daseinsberechtigung.

Mit Visual Studio 2015 werden wir im Praxisteil dieses Kapitels die Euro-Dollar-Umrechnung nochmals programmieren, dann allerdings mit einer attraktiven Windows-Oberfläche. Bevor es aber damit losgehen kann, sollten Sie sich ein wenig mit der Windows-Philosophie anfreunden.

1.3 Die Windows-Philosophie

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

1.3.1 Mensch-Rechner-Dialog

Die Art und Weise, **wie** die Kommunikation mit dem Benutzer (Mensch-Rechner-Dialog) abläuft, dürfte wohl der gravierendste Unterschied zwischen einer klassischen Konsolenanwendung und einer typischen Windows-Anwendung sein. Wie Sie es bereits im Einführungsbeispiel 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.

---

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

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 zusetzen:

Objekte (Objects)

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

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

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

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

Eigenschaften (Properties)

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

Methoden (Methods)

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

Ereignisse (Events)

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

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

1.3.3 Programmieren mit Visual Studio 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 größeren Projekten sehr lästigen und nervtötenden Herumwursteln und erlaubt (unabhängig von der verwendeten Programmiersprache) eine systematische Vorgehensweise in vier Etappen:

6. Visueller Entwurf der Bedienoberfläche
7. Zuweisen der Objekteigenschaften
8. Verknüpfen der Objekte mit Ereignissen
9. Kompilieren und Testen der Anwendung

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

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

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

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

Die *dritte Etappe* haucht Leben in unsere bislang nur mit statischen Attributen ausgestatteten Objekte. Hier muss in so genannten *Ereignisbehandlungsroutinen* (Eventhandlern) festgelegt werden, **wie** das Formular oder das betreffende Steuerelement auf bestimmte Ereignisse zu reagieren hat. Visual Studio stellt auch hier "vorgefertigten" Rahmencode (erste und letzte Anweisung) für alle zum jeweiligen Objekt passenden Ereignisse zur Verfügung. Der Programmierer füllt diesen Rahmen mit C#-Quellcode aus. Hier können Methoden oder Prozeduren aufgerufen werden, aber auch Eigenschaften anderer Objekte lassen sich während der Laufzeit neu zuweisen.

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

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

Der in diesem Zyklus praktizierte visuelle Oberflächenentwurf, verbunden mit dem ereignisorientierten Entwurfskonzept, macht *Visual Studio 2015* zu einer hoch effektiven 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.

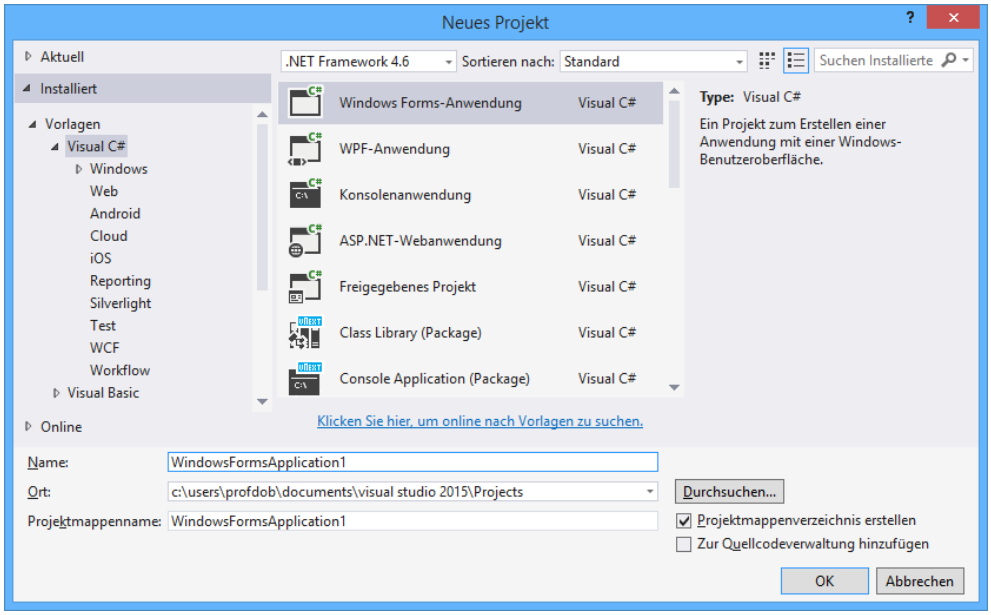
C# ist nur eine der möglichen objektorientierten Sprachen, die Sie unter Visual Studio 2015 einsetzen können. So werden z.B. noch Visual Basic, Visual C++ und Visual F# unterstützt.

[1] *Integrated Developers Environment*

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

1.4.1 Neues Projekt

Wählen Sie auf der Startseite von Visual Studio 2015 den Menüpunkt *Neues Projekt...*, so öffnet sich der Startdialog *Neues Projekt* mit einem umfangreichen und zunächst verwirrenden Angebot an unterschiedlichen Vorlagen[1] für Visual C#-Projekttypen, wobei für den Einsteiger zunächst die klassische *Windows Forms-Anwendung* empfohlen wird.



Visual Studio 2015 erlaubt es, Programme für verschiedene .NET-Framework-Versionen zu entwickeln (Multi-Targeting, siehe obere Klappbox).

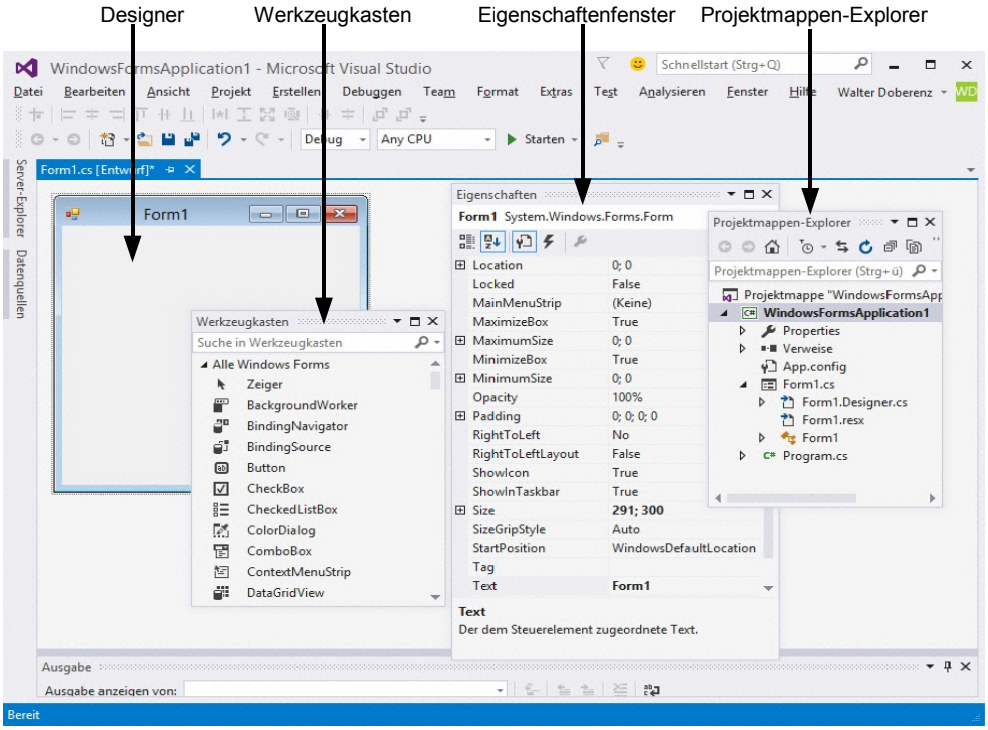
Haben Sie das Häkchen bei *Projektmappenverzeichnis erstellen* gesetzt, so erzeugt Visual Studio automatisch einen Unterordner mit dem Namen des Projekts in dem als Speicherort eingetragenen Hauptverzeichnis.

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

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

1.4.2 Die wichtigsten Fenster

Haben Sie als Projekttyp beispielsweise *Windows Forms-Anwendung* gewählt, könnte Ihnen Visual Studio 2015 etwa den folgenden Anblick bieten, wobei auf die für den Einsteiger zunächst wichtigsten Fenster besonders hingewiesen wird.



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

Der Projektmappen-Explorer

Da es unter Visual Studio 2015 möglich ist, mehrere Projekte gleichzeitig zu bearbeiten, gibt es eine Projektmappendatei mit der Extension *.sln* (Solution), deren Inhalt im Projektmappen-Explorer (*STRG+R*) übersichtlich angezeigt wird. Sie können dieses Fenster deshalb ohne Übertreibung als "Schaltzentrale" Ihres Projekts betrachten.

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

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

Zur Bedeutung der einzelnen Einträge bzw. Dateien:

- *Properties*
  Hier sind verschiedene Dateien zusammengefasst, die die Projekteigenschaften bestimmen. *AssemblyInfo.cs* enthält z.B allgemeine Infos zur Assemblierung des Projekts, wie Titel, Beschreibung, Versionsnummer, Copyright. Weitere Dateien beziehen sich auf die Ressourcen und die Projekteinstellungen.
- *Verweise*
  Hier sind die aktuell für das Projekt gültigen Verweise auf Namensräume bzw. Assemblierungen enthalten. Standardmäßig hat Visual Studio bereits die wichtigsten Verweise eingestellt, weitere können Sie über das Kontextmenü der rechten Maustaste hinzufügen.
- *Form1.cs*
  Diese Datei enthält eine partielle Klasse[1], die den von Ihnen selbst hinzugefügten Code von *Form1* kapselt.
- *Form1.Designer.cs*
  Diese Datei enthält eine partielle Klasse, die den vom Windows Forms Designer automatisch generierten Code von *Form1* kapselt. Der gesamte Code von *Form1* ist also zwischen denpartiellen Klassen in *Form1.cs* und *Form1.Designer.cs* aufgeteilt.
- *Program.cs*
  Eine statische Klasse, welche die *Main*-Methode (den Einsprungpunkt der Anwendung) enthält. In dieser Methode wird durch Aufruf von *Application.Run* eine Nachrichtenschleife gestartet, die ununterbrochen auf Ereignisse wartet, damit die Anwendung darauf reagieren kann.

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

Der Designer

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

Der Werkzeugkasten

Den Werkzeugkasten werden Sie häufig benötigen (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.

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

Das Eigenschaften-Fenster

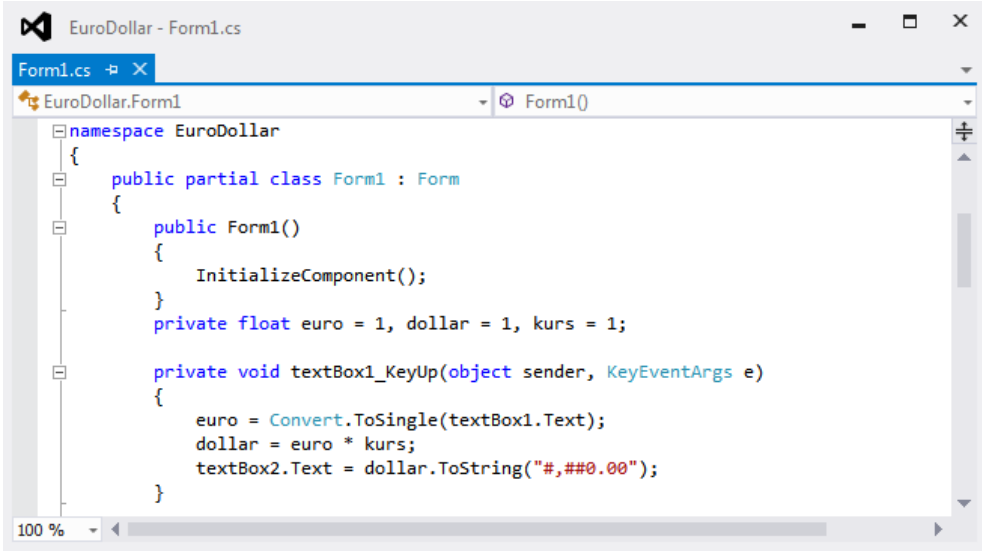
Im Eigenschaften-Fenster (Menü *Ansicht/Eigenschaftenfenster* bzw. *F4*) werden die zur Entwurfszeit editierbaren Eigenschaften des gerade aktiven Steuerelements aufgelistet[1]. Normalerweise hat jede Eigenschaft bereits einen Standardwert, den Sie in vielen Fällen übernehmen können.

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.

Das Codefenster

Für die eigentliche Programmierung ist das Codefenster zuständig. Logischerweise wird dies damit auch zu Ihrem Hauptbetätigungsfeld als C#-Programmierer. Um zum Beispiel das *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* im Praxisbeispiel .

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



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

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

1.5 Microsofts .NET-Technologie

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

1.5.1 Zur Geschichte von .NET

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

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

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

Warum eine extra Programmiersprache?

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

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

Als konsequent objektorientierte Sprache erfüllt C# folgende Kriterien:

- **Abstraktion**
  Die Komplexität eines Geschäftsproblems ist beherrschbar, indem eine Menge von abstrakten Objekten identifiziert werden können, die mit dem Problem verknüpft sind.
- **Kapselung**
  Die interne Implementation einer solchen Abstraktion wird innerhalb des Objekts versteckt.
- **Polymorphie**
  Ein und dieselbe Methode kann auf mehrere Arten implementiert werden.

- **Vererbung**
  Es wird nicht nur die Schnittstelle, sondern auch der Code einer Klasse vererbt (Implementations-Vererbung statt der COM-basierten Schnittstellen-Vererbung).

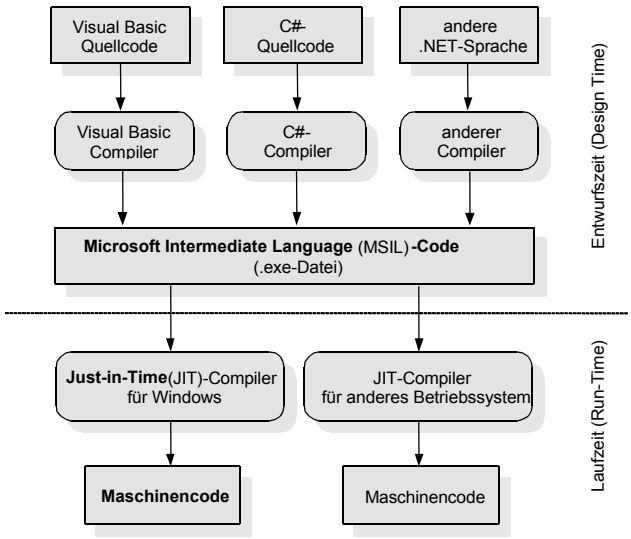
Microsoft kann natürlich nicht über Nacht die COM-Technologie auf die Müllkippe entsorgen, denn zu viele Programmierer würden dadurch auf immer und ewig verprellt werden und sich frustriert einer stabileren Entwicklungsplattform zuwenden. Aus diesem Grund wird COM auch in .NET noch einige Zeit sein Gnadenbrot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

Wie funktioniert eine .NET-Sprache?

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

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

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



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

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.

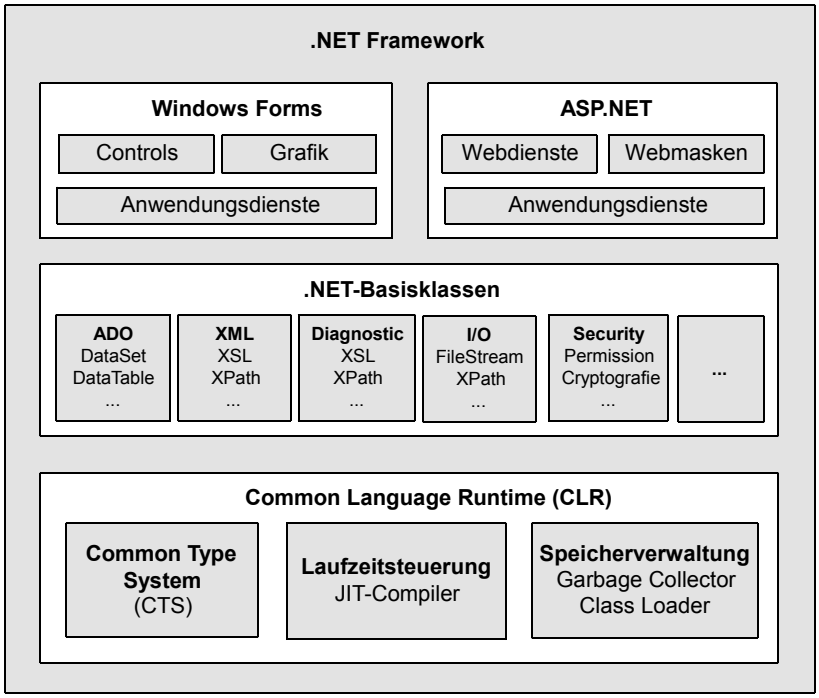
1.5.2 .NET-Features und Begriffe

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:

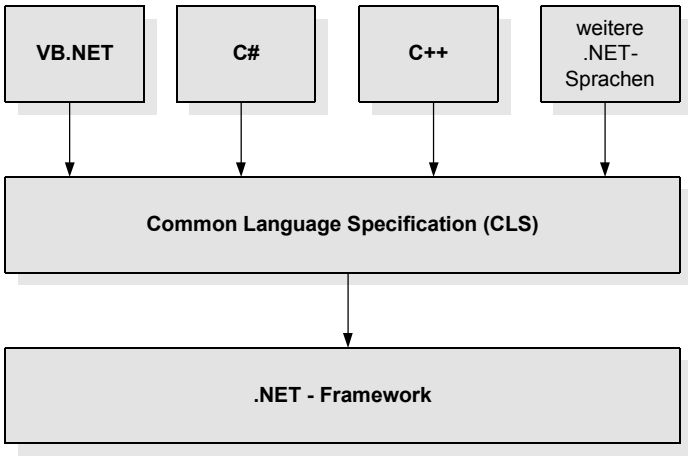
- Common Language Specification (CLS)
- Common Type System (CTS)

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

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

Die Common Language Specification (CLS)

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



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

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

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

Das Common Type System (CTS)

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

Alle diese Typen lassen sich in zwei Kategorien aufteilen:

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

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

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

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

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

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

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

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

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

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

Die Common Language Runtime (CLR)

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

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

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

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

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

Namespaces ersetzen Registry

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

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

Namespace	... enthält Klassen für ...
System.Windows.Forms	... Windows-basierte Anwendungen
System.Collections	... Objekt-Arrays
System.Drawing	... die Grafikprogrammierung

Namespace	... enthält Klassen für ...
System.Data	... den ADO-Datenbankzugriff
System.Web	... die HTTP-Webprogrammierung
System.IO	... Ein- und Ausgabeoperationen

Mit den Namespaces hat auch der Ärger mit der Registrierung von (COM-)Komponenten bei Versionskonflikten sein Ende gefunden, denn eine unter .NET geschriebene Komponente wird von der .NET-Runtime nicht mehr über die ProgID der Klasse mit Hilfe der Registry lokalisiert, sondern über einen in der Runtime enthaltenen Mechanismus, welcher einen Namespace einer angeforderten Komponente sowie deren Version für das Heranziehen der "richtigen" Komponente verwendet.

Assemblierungen

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

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

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

Zugriff auf COM-Komponenten

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

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

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

Metadaten und Reflexion

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

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

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

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

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

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

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

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

Attribute

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

Unter .NET haben Attribute eine grundsätzlich andere Bedeutung:

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

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

Man kann Attribute auch mit "Anmerkungen" vergleichen, die man einzelnen Quellcode-Elementen, wie Klassen oder Methoden, "anheftet". Solche Attribute gibt es eigentlich in jeder Programmiersprache, sie regeln z.B. die Sichtbarkeit eines bestimmten Datentyps. Allerdings waren diese

Fähigkeiten bislang fest in den Compiler integriert, während sie unter .NET nunmehr direkt im Quellcode zugänglich sind. Das heißt, dass .NET-Attribute typsichere, erweiterbare Metadaten sind, die zur Laufzeit von der CLR (oder von beliebigen .NET-Anwendungen) ausgewertet werden können.

Mit Attributen können Sie Design-Informationen definieren (z.B. zur Dokumentation), Laufzeit-Infos (z.B. Namen einer Datenbankspalte für ein Feld) oder sogar Verhaltensvorschriften für die Laufzeit (z.B. ob ein gegebenes Feld an einer Transaktion teilnehmen darf). Die Möglichkeiten sind quasi unbegrenzt.

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

Serialisierung

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

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

Das .NET-Framework unterstützt zwei verschiedene Serialisierungsmechanismen:

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

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

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

Multithreading

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

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

Nicht in jedem Fall ist die Aufnahme zusätzlicher Threads die beste Lösung, da man sich dadurch auch zusätzliche Probleme einhandeln kann. So ist beim Umgang mit mehreren Threads die Threadsicherheit von größter Bedeutung, d.h., aus Sicht der Threads müssen die Objekte stets in einem gültigen Zustand vorliegen und das auch dann, wenn sie von mehreren Threads gleichzeitig benutzt werden.

Objektorientierte Programmierung

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

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

1.6 Wichtige Neuigkeiten in Visual Studio 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*).

1.6.2 Neue C#-Sprachfeatures

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

- Das Beschreiben von Formatstrings lässt sich mittels Stringinterpolierung vereinfachen
- Beim Aufruf von Membern und Indexern kann eine integrierte Nullprüfung durchgeführt werden

- Um den Namen eines Parameters, Members oder Typs als Zeichenfolge zu gewinnen, bietet *nameof* jetzt einen typsicheren Weg
- Auch Auto-Eigenschaften können jetzt Initialisierer haben, ein Setter wird nicht mehr benötigt
- Objektinitialisierer gestatten es jetzt, dass man einen spezifischen Index des neuen Objekts initialisiert
- Mit Hilfe der Klausel *using static* für statische Klassen kann man statische Member direkt aufrufen, der Klassennamen braucht nicht mehr vorangestellt werden
- Ausnahmefilter ermöglichen die Konzentration auf eine ganz bestimmte Ausnahme und die Entscheidung, ob sie mit einem vorhandenen *catch*-Block abgefangen werden soll
- Der *await*-Befehl funktioniert jetzt auch in *catch-finally*-Blöcken und vereinfacht somit die asynchrone Programmierung

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] Zur Aktivierung müssen Sie *EnableWindowsFormsHighDpiAutoResizing* in der *app.config* auf *true* setzen.

1.7 Praxisbeispiele

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

Für diese kleinen Applikationen sind nicht die geringsten Programmierkenntnisse erforderlich, es geht vielmehr darum, ein erstes Gefühl für die Anwendungsentwicklung unter Visual Studio 2015 zu gewinnen.

1.7.1 Unsere erste Windows Forms-Anwendung

Die bescheidene Funktionalität beschränkt sich auf ein Fensterchen mit einer Schaltfläche, über welche per Mausklick die Beschriftung der Titelleiste in "Hallo C#-Freunde" geändert werden kann. Das Beispiel demonstriert, mit welch geringem Aufwand man in Visual Studio eigene Windows-Anwendungen erstellen kann. Der damit ausgelöste Aha-Effekt wird Sie sicher ausreichend motivieren, manche Durststrecken der nächsten Kapitel zu überstehen.

1. Etappe: Visueller Entwurf der Bedienoberfläche

Der Programmstart von *Microsoft Visual Studio 2015* erfolgt entweder über das Windows-Startmenü oder noch schneller über eine Desktop- bzw. Taskleisten-Verknüpfung.

Auf der Startseite klicken Sie den Link *Neues Projekt...*. Im sich daraufhin öffnenden Dialogfenster *Neues Projekt* wählen Sie links in der Baumstruktur unter *Vorlagen* zunächst *Visual C#* und *Windows* aus (siehe Abbildung Seite 62).

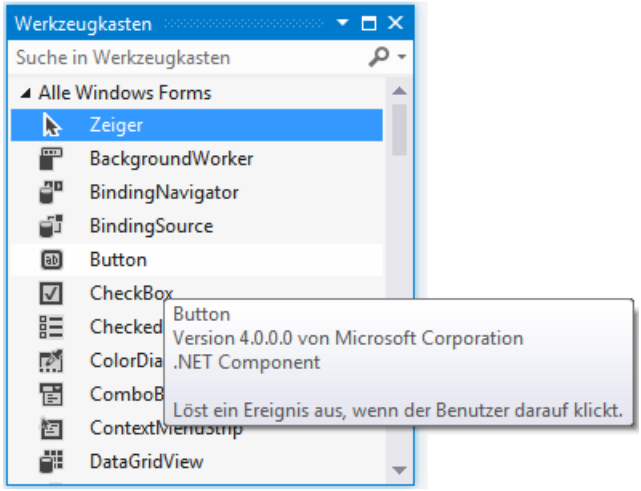
Im Mittelteil klicken Sie auf *Windows Forms-Anwendung* (ganz oben in der Liste). Nehmen Sie im unteren Teil die folgenden Einträge vor bzw. belassen es bei den Standardvorgaben:

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

HINWEIS: Im Dialogfenster rechts oben sehen Sie, dass Sie mit Visual Studio 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 das gewünschte Steuerelement aus:



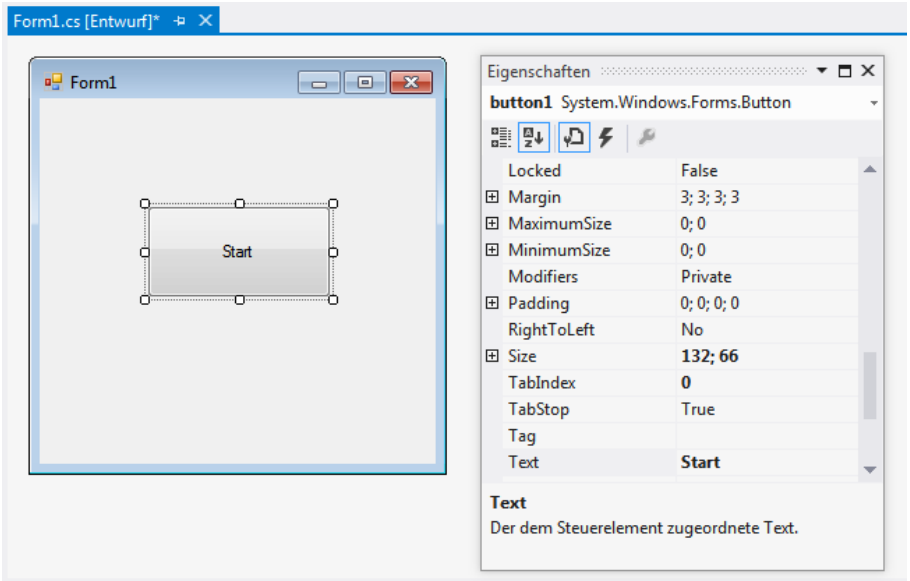
Ein schneller Doppelklick befördert das Steuerelement direkt auf das Formular. Sie können aber auch den gewünschten *Button* in gewohnter Windows-Manier auf das Formular ziehen, dort absetzen und auf die gewünschte Größe zu zoomen.

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 Steuerelement mit der Maus und rufen Sie mit F4 (bzw. über das Menü *Ansicht*) das Eigenschaftenfenster auf. Ändern Sie im Eigenschaften-Fenster die *Text*-Eigenschaft von ihrem Standardwert "button1" in "Start".

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

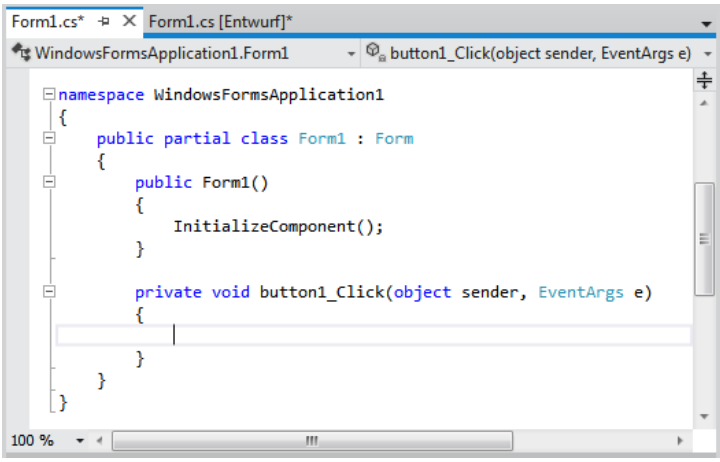
Es dürfte Ihnen nun auch keine Schwierigkeiten bereiten, über die *Font*-Eigenschaft von *button1* auch noch die Schriftgröße etc. zu ändern.



3. Etappe: Verknüpfen der Objekte mit Ereignissen

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

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



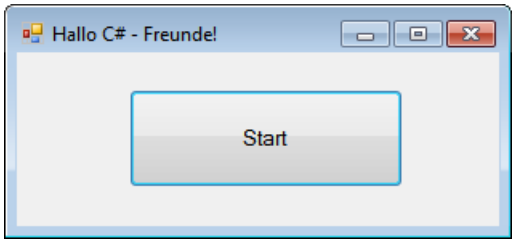
Fügen Sie die fett hervorgehobene Zeile in den bereits vorhandenen Rahmencode ein:

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

4. Etappe: Programm kompilieren und testen

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

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



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

Bemerkungen

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

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

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

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

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

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

- Die als Ergebnis des Kompilierprozesses generierte *.exe*-Datei finden Sie im Unterverzeichnis *...\WindowsFormsApplikation1\bin\Debug* Ihres Projektordners. Es handelt sich hierbei allerdings **nicht** um eine klassische Exe-Datei, sondern um eine so genannte *Assemblierung* (siehe Abschnitt 1.5). Da die EXE im MSIL-Code vorliegt, ist sie nur in Zusammenarbeit mit dem .NET-Framework lauffähig. Haben Sie die Programmentwicklung erfolgreich abgeschlossen und Visual Studio beendet, so können Sie später jederzeit in dieses Verzeichnis wechseln, um durch Doppelklick auf die Datei *WindowsFormsApplication1.exe* das Programm zu starten.
- Direkt im Projektverzeichnis befindet sich die Projektmappendatei *WindowsFormsApplication1.sln*. Wenn Sie auf diese Datei klicken[1], so wird standardmäßig Visual Studio geöffnet und das Programm wird in die Entwicklungsumgebung geladen.

1.7.2 Umrechnung Euro-Dollar

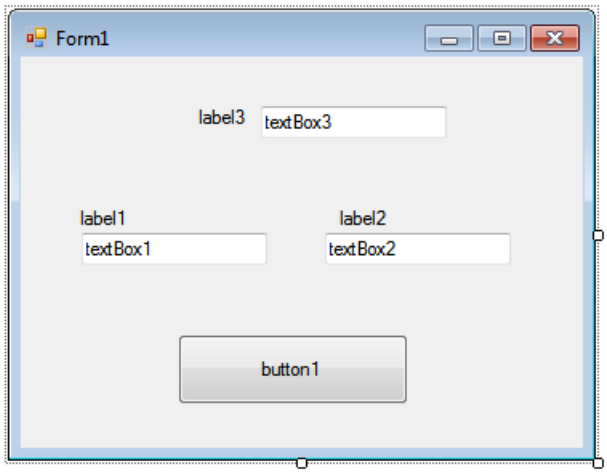
Diesmal soll es keine Spielerei, sondern ein durchaus nützliches Prograämmchen sein – die Umrechnung von Euro in Dollar, ein simpler Währungsrechner also. Durch Vergleichen mit der im Abschnitt 1.2 beschriebenen ersten C#-Anwendung dürften auch die Unterschiede der klassischen Konsolentechnik zur visualisierten,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

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

1. Etappe: Visueller Entwurf der Bedienoberfläche

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

Ziel ist die folgende Bedienoberfläche, die Sie jetzt mühelos im Designer-Fenster erstellen (siehe folgende Abbildung)[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 Komponenten, nicht auf deren Beschriftung, da Eigenschaften erst in der nächsten Etappe angepasst werden!

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

- Im Werkzeugkasten klicken Sie auf das Symbol für die *TextBox*-Komponente. Der Mauszeiger wechselt sein Aussehen.
- Danach bewegen Sie den Mauszeiger zu der Stelle von *Form1*, an welcher sich die linke obere Ecke von *textBox1* befinden soll, drücken die Maustaste nieder und zoomen (bei gedrückt gehaltener Maustaste) die *TextBox* auf ihre endgültige Größe. Analog verfahren Sie mit *textBox2* und *textBox3*.

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

- Nun klicken Sie im Werkzeugkasten auf das Symbol für die *Label*-Komponente und erzeugen auf die gleiche Weise *label1*, *label2* und *label3*.
- Schließlich bleibt noch *button1*, den Sie am unteren Rand von *Form1* absetzen.

2. Etappe: Zuweisen der Objekteigenschaften

Unser Programm besteht nun aus insgesamt acht Komponenten: einem Formular und sieben Steuerelementen. Alle Eigenschaften haben bereits ihre Standardwerte. Einige davon müssen wir allerdings noch ändern. Dies geschieht mit Hilfe des Eigenschaften-Fensters. Wenn Sie mit der Maus auf eine Komponente klicken und danach die *F4*-Taste betätigen, erscheint das Eigenschaften-Fenster der Komponente mit der Liste aller zur Entwurfszeit verfügbaren Eigenschaften.

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

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

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

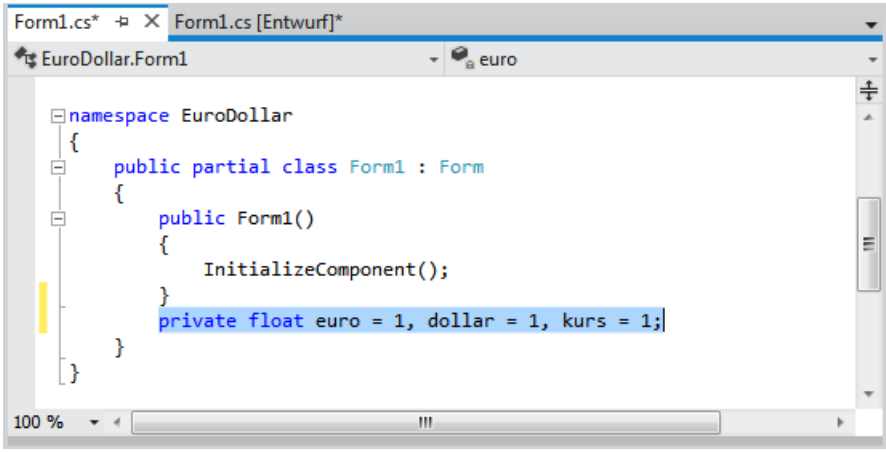
3. Etappe: Verknüpfen der Objekte mit Ereignissen

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

einzelne Bereiche (Regionen) durch das Plus- bzw. Minus-Symbol am linken Rand auf- bzw. zuklappen.

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

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

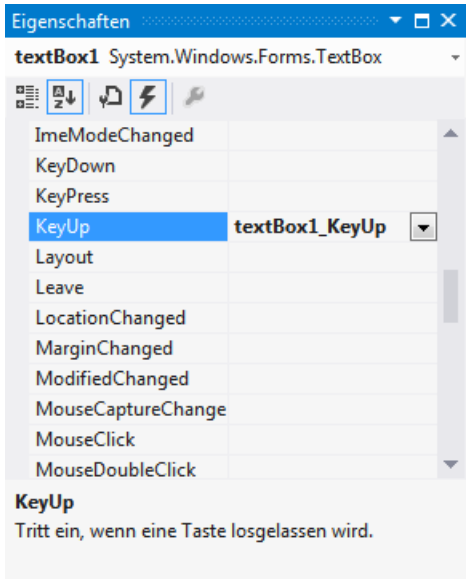


Im Unterschied zur einfachen Konsolenanwendung, bei welcher uns das Programm die Einhaltung einer bestimmten Eingabereihenfolge aufgezwungen hat, soll in unserer Windows Forms-Anwendung die Berechnung immer dann neu gestartet werden, wenn wir bei der Eingabe in eine der drei Textboxen irgendeine Taste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

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

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

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



Füllen Sie nun den Rahmencode wie folgt aus:

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

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

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

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

Ändern des Dollar-Betrags:

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

Ändern des Umrechnungskurses:

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

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

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

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

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

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

4. Etappe: Programm kompilieren und testen

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

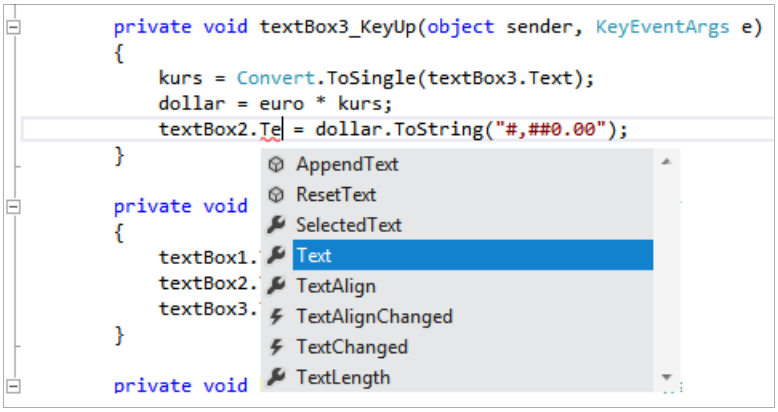
Spielen Sie ruhig ein wenig mit verschiedenen Werten herum, um sich den Unterschied zwischen Konsolen- und Windows-Programmen zu verinnerlichen. Da es keine vorgeschriebene Reihenfolge für die Benutzereingaben mehr gibt, werden die anderen Felder sofort aktualisiert. Eine spezielle "="-Schaltfläche (etwa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

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

IntelliSense – die hilfreiche Fee

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

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

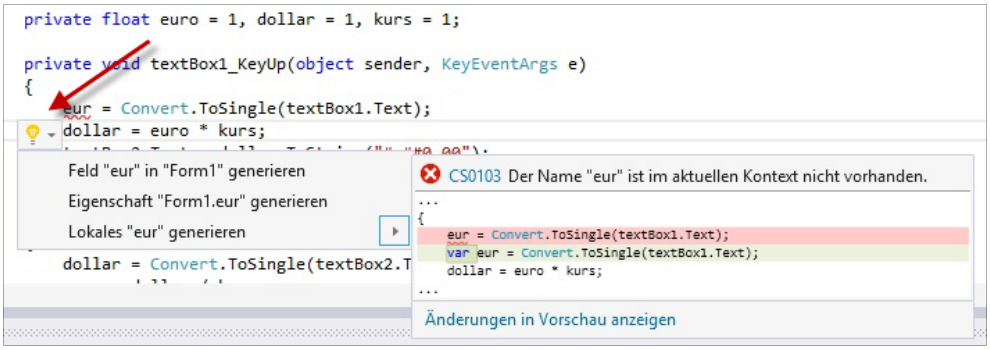


Die Glühbirne sorgt für Erleuchtung

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

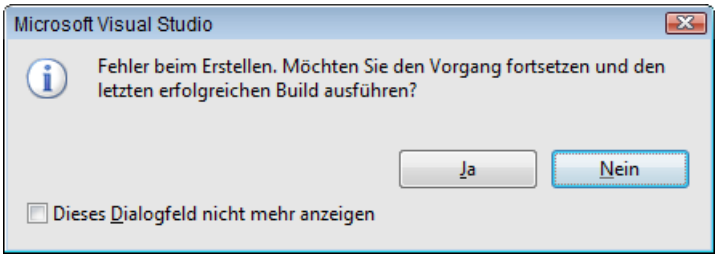
In der folgenden Abbildung wurde z.B. die Variable *euro* falsch eingetippt:

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



Fehler beim Kompilieren

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



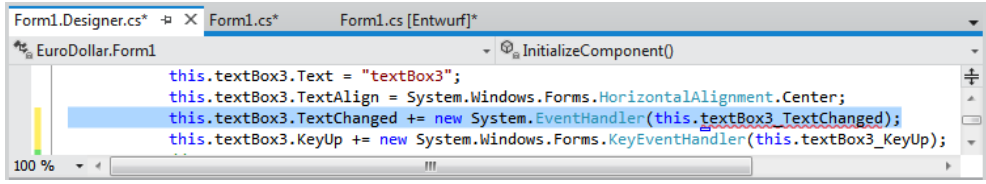
In der Regel sollten Sie *Nein* klicken, um unverzüglich den (oder die) Übeltäter im Quellcode zu suchen und dingfest zu machen. Das Lokalisieren ist meist sehr einfach, da der fehlerhafte Ausdruck durch eine Wellenlinie unterschlängelt ist. Auch hier erhalten Sie Hinweise zur Fehlerursache, wenn Sie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

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

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

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

Erst ein Blick auf die Fehlerliste bringt Sie auf die richtige Fährte, denn Sie haben zwar den Rahmencode des *textBox3_TextChanged*-Eventhandlers wieder gelöscht, nicht aber seine (von Visual Studio automatisch angelegte) Deklaration. Diese ist nach wie vor vorhanden und verweist (für Sie zunächst unsichtbar) auf den nun nicht mehr vorhandenen Code. Klicken Sie deshalb auf den Link "Gehe zu Code" (oder doppelklicken Sie auf den Eintrag in der Fehlerliste), worauf sich das Codefenster der Datei *Form1.Designer.cs* öffnet[1]:



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

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

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

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

Kapitel 2

Grundlagen der Sprache C#

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

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

2.1 Grundbegriffe

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

2.1.1 Anweisungen

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

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

---

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

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

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

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

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

2.1.2 Bezeichner

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

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

BEISPIEL 2.1: Zulässige Bezeichner

C#
```
euro
 _radius
zwerg7
```

BEISPIEL 2.2: Unzulässige Bezeichner

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

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

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

BEISPIEL 2.3: Beide Bezeichner sollten Sie nicht nebeneinander verwenden:

C#
```
MeineAdresse
meineAdresse
```

2.1.3 Schlüsselwörter

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

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

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

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

BEISPIEL 2.4: Schlüsselwörter mit Präfix

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

2.1.4 Kommentare

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

Einzeilige Kommentare

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

BEISPIEL 2.5: Eine Anweisung mit Kommentar

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

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

Mehrzeilige Kommentare

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

BEISPIEL 2.6: Mehrzeiliger Kommentar

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

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

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

2.2 Datentypen, Variablen und Konstanten

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

2.2.1 Fundamentale Typen

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

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

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

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

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

2.2.2 Wertetypen versus Verweistypen

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

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

Wertetypen

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

Verweistypen

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

2.2.3 Benennung von Variablen

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

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

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

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

C#
```
meinHaushaltskassenSaldo
```

2.2.4 Deklaration von Variablen

Variablen werden in C# wie folgt deklariert:

SYNTAX: *Datentyp Variablenname*;

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

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

BEISPIEL 2.8: Drei Variablen unterschiedlichen Datentyps werden deklariert.

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

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

BEISPIEL 2.9: Drei int-Variablen werden deklariert.

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

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

BEISPIEL 2.10: Die folgenden drei Deklarationen sind gleichwertig.

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

Initialisierte Variablen

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

BEISPIEL 2.11: Initialisierte Variablen

C#

Statt

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

können Sie kürzer formulieren:

```
int anzahl = 99;
```

Typinferenz

Dieses in Zusammenhang mit der LINQ-Technologie (siehe Kapitel 6 ab Seite 381) neu eingeführte Sprachmerkmal erlaubt es, dass der Datentyp von Variablen bei der Deklaration vom Compiler automatisch ermittelt wird, ohne dass Sie explizit den Typ angegeben müssen. Als Ersatz für einen konkreten Typ wird das Schlüsselwort *var* verwendet.

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

BEISPIEL 2.12: Typinferenz

C#

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

```
var a = 35;
```

Obige Zeile ist semantisch identisch mit folgendem Ausdruck:

```
int a = 35;
```

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

BEISPIEL 2.13: Keine Datentypänderung möglich

C#

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

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

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

2.2.5 Typsuffixe

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

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

BEISPIEL 2.14: Richtige und falsche Literalzuweisungen.

C#

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

2.2.6 Zeichen und Zeichenketten

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

char

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

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

BEISPIEL 2.15: char

C#

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

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

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

BEISPIEL 2.16: Eine weitere Ergänzung zum Vorgängerbeispiel

C#

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

string

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

BEISPIEL 2.17: Zuweisen einer Stringvariablen

C#

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

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

BEISPIEL 2.18: Das erste Zeichen eines Strings ermitteln

C#

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

HINWEIS: Ausführliches zur Stringverarbeitung finden Sie in Kapitel 4.

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

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

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

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

BEISPIEL 2.19: Korrekte Schreibweise für Dateipfade

C#

Die Anweisung zur Definition eines Dateipfads

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

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

Die folgende Anweisung wäre korrekt:

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

oder in diesem Fall auch die Kurzform:

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

BEISPIEL 2.20: Hinzufügen eines Zeilenvorschubs mit Signalton

C#

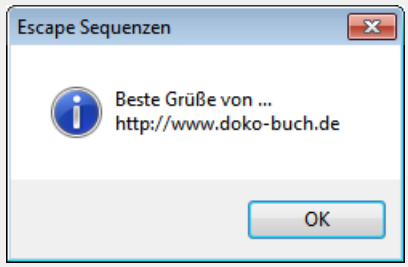
```
MessageBox.Show("Beste Grüße von ...\r\n\ahttp://www.doko-buch.de", "Escape Sequenzen",
                MessageBoxButtons.OK, MessageBoxIcon.Information);
```

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

BEISPIEL 2.20: Hinzufügen eines Zeilenvorschubs mit Signalton

Ergebnis

Ein Signalton ertönt und folgendes Meldungsfenster erscheint:



2.2.7 object-Datentyp

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

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

Variablen des Typs *object* können Sie alles zuweisen.

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

C#

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

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

BEISPIEL 2.22: Die Fortsetzung des Vorgängerbeispiels

C#

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

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

2.2.8 Konstanten deklarieren

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

SYNTAX: `const Datentyp Konstantenname = Wert;`

BEISPIEL 2.23: Verschiedene Konstantendeklarationen

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

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

2.2.9 Nullable Types

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

BEISPIEL 2.24: Falsche und richtige Verwendung von Variablen

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

Initialisieren von Wertetypen mit null

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

BEISPIEL 2.25: Das funktioniert nicht!

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

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

BEISPIEL 2.26: Aber das funktioniert!

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

Oder als explizite Deklaration:

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

Zuweisungen mit dem ??-Operator

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

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

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

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

BEISPIEL 2.28: Das Vorgängerbeispiel wird einfacher realisiert.

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

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

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

2.2.10 Typinferenz

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

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

BEISPIEL 2.30: *var*-Deklaration

C#

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

```
var a = 35;
```

Obige Zeile ist semantisch identisch mit folgendem Ausdruck:

```
int a = 35;
```

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

BEISPIEL 2.31: Keine Änderung möglich ...

C#

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

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

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

2.2.11 Gültigkeitsbereiche und Sichtbarkeit

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

Trotzdem sollten Sie bereits jetzt Folgendes wissen:

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

2.3 Konvertieren von Datentypen

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

2.3.1 Implizite und explizite Konvertierung

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

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

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

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

BEISPIEL 2.33: Das geht nicht

C#

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

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

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

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

BEISPIEL 2.34: Das Vorgängerbeispiel wird fehlerfrei ausgeführt

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

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

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

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

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

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

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

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

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

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

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

BEISPIEL 2.38: Explizite Typkonvertierung des Ergebnisses einer Division

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

as-Konverierungsoperator

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

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

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

2.3.2 Welcher Datentyp passt zu welchem?

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

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

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

2.3.3 Konvertieren von string

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

ToString-Methode

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

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

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

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

BEISPIEL 2.40: Anzeige einer Gleitkommazahl

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

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

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

String in Zahl verwandeln

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

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

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

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

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

BEISPIEL 2.43: Das geht leider nicht.

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

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

BEISPIEL 2.44: Das Vorgängerbeispiel kann wie folgt gelöst werden.

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

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

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

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

BEISPIEL 2.46: Konvertieren eines Stringliterals in eine Ganzzahl.

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

EVA-Prinzip

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

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

BEISPIEL 2.47: Ein Ausschnitt aus dem Einführungsbeispiel 1.7.2

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

2.3.4 Die Convert-Klasse

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

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

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

expr = zu konvertierender Ausdruck

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

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

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

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

2.3.5 Die Parse-Methode

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

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

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

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

2.3.6 Boxing und Unboxing

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

BEISPIEL 2.51: Boxing und Unboxing

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

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

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

Boxing

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

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

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

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

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

Unboxing

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

BEISPIEL 2.53: Das Vorgängerbeispiel wird fortgesetzt.

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

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

BEISPIEL 2.54: Das geht nicht!

C#

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

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

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

BEISPIEL 2.55: Ja, auch das funktioniert!

C#

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

2.4 Operatoren

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

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

Die meisten Operatoren in C# benötigen zwei Operanden.

BEISPIEL 2.56: Operanden

C#

Im Ausdruck

```
i = 12;
```

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

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

2.4.1 Arithmetische Operatoren

Standard-Operatoren

Es gibt zunächst die üblichen Operatoren für die Grundrechenarten:

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

BEISPIEL 2.57: Standard-Operatoren

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

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

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

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

Inkrement- und Dekrement-Operatoren

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

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

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

BEISPIEL 2.59: Postfix-Inkrement und Postfix-Dekrement

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

BEISPIEL 2.60: Äquivalente Version des Vorgängerbeispiels mit Präfixoperationen

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

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

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

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

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

BEISPIEL 2.62: Dasselbe für eine Präfix-Inkrementation.

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

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

2.4.2 Zuweisungsoperatoren

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

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

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

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

BEISPIEL 2.63: Kurz-Operatoren

C#

Statt

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

kann man auch schreiben

```
i += 3;
```

BEISPIEL 2.64: Kurz-Operatoren

C#

Für

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

gibt es diese

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

oder diese Möglichkeit zum Anhängen einer weiteren Zeichenkette:

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

2.4.3 Logische Operatoren

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

Vergleichsoperatoren

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

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

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

BEISPIEL 2.65: Vergleich von zwei Integer-Zahlen

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

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

BEISPIEL 2.66: Vergleichsoperatoren im Zusammenhang mit Verzweigungsbefehlen,

C#

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

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

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

BEISPIEL 2.67: Zwei Strings werden verglichen.

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

Boolesche Operatoren

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

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

BEISPIEL 2.68: Boolesche Operatoren

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

Kurzschlussauswertung

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

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

BEISPIEL 2.69: Kurzschlussauswertung

C#

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

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

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

Bitweise Operationen

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

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

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

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

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

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

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

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

2.4.4 Rangfolge der Operatoren

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

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

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

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

BEISPIEL 2.72: Arithmetische Operationen

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

aber

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

BEISPIEL 2.73: Boolesche Operationen

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

2.5 Kontrollstrukturen

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

2.5.1 Verzweigungsbefehle

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

Klassische Entscheidungsanweisungen

Die folgende Tabelle gibt einen Überblick[1].

Verzweigungsanweisung	Erklärung

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

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

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

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

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

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

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

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

BEISPIEL 2.75: Im Label wird "Befriedigend" angezeigt

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

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

BEISPIEL 2.76: Diese Kontrollstruktur leistet das Gleiche wie das Vorgängerbeispiel

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

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

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

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

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

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

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

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

C#
An den Verzeichnisnamen *myPath* wird ein Slash (/) angehängt, falls keiner vorhanden ist.

```
if (!myPath.EndsWith("/")) myPath += "/";
```

Ergänzung

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

BEISPIEL 2.79: Der Verzweigungsbefehl

C#
```
label1.Text = checkBox1.Checked ? "Ja"  : "Nein";
```

entspricht

```
if (checkBox1.Checked) label1.Text = "Ja";
else label1.Text = "Nein";
```

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

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

Sprungbefehle

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

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

BEISPIEL 2.81: Eine Alternative zum Vorgängerbeispiel (auszugsweise)

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

2.5.2 Schleifenanweisungen

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

Schleifenanweisung	Erklärung

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

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

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

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

for-Schleifen

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

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

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

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

BEISPIEL 2.83: Eine äquivalente Version des Vorgängerbeispiels

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

while-Schleife

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

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

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

do-Schleife

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