

C# mit .NET und MS Visual Studio

Eine Einführung von Prof. Rasso Steinmann

Skript zur Vorlesung Bauinformatik I

Version 3.0

C# mit .NET und MS Visual Studio

Eine Einführung von Prof. Rasso Steinmann

Version 3.0

| | |
|---|----|
| C# mit .NET und MS Visual Studio | 1 |
| 1 Ziel | 6 |
| 2 Hintergrund-Informationen | 6 |
| 2.1 Einleitung | 6 |
| 2.2 Was ist .NET? | 6 |
| 2.3 C# und anderen Programmiersprachen | 8 |
| 2.3.1 C-Syntax, Objekte, Internet..... | 8 |
| 2.3.2 C# und Java | 8 |
| 2.3.3 Programmiersprachen und Standards..... | 9 |
| 2.3.4 Visual Basic 6.0 und Visual Basic.NET | 9 |
| 2.4 Warum ist Programmieren heute komplizierter? | 10 |
| 3 Begleit-Literatur | 11 |
| 4 Schritt für Schritt zum ersten C#-Programm..... | 12 |
| 4.1 Visual Studio.NET | 12 |
| 4.2 Schritt 0: Persönlichen Ordner anlegen..... | 12 |
| 4.3 Schritt 1: Anlegen eines Projektes | 13 |
| 4.4 Schritt 2: Quell-Code verstehen | 14 |
| 4.5 Schritt 3: Quell-Code eingeben | 15 |
| 4.6 Schritt 4: Anwendung erstellen und ausführen | 17 |
| 4.7 Architektur eines C#-Programmes | 18 |
| 4.7.1 Namespaces..... | 18 |
| 4.8 Gratulation..... | 19 |
| 4.9 Wo liegt das fertige Programm?..... | 20 |
| 5 Ein erstes Windows-Programm..... | 21 |
| 5.1 Schritt 1: Anlegen eines Projektes | 21 |
| 5.2 Schritt 2: Arbeiten mit Forms und Steuerelementen..... | 23 |
| 5.3 Schritt 3: Automatisch erzeugten Quell-Code für Forms verstehen | 24 |
| 5.4 Schritt 4: Eigenschaften von Steuerelementen verändern..... | 30 |
| 5.5 Schritt 5: Größe des Formulars ändern | 31 |
| 5.6 Schritt 6: Code für OK schreiben..... | 32 |
| 5.7 Schritt 7: Programm übersetzten und starten | 33 |
| 5.8 Gratulation..... | 33 |
| 6 C#-Bausteine | 34 |
| 6.1 C#-Syntax..... | 34 |
| 6.2 Reservierte Schlüsselwörter | 34 |
| 6.3 Variablen | 34 |
| 6.3.1 Variablen deklarieren | 35 |
| Gültigkeit von Variablen..... | 35 |
| 6.3.2 Einfache Datentypen | 36 |
| 6.3.3 Komplexe Datentypen..... | 36 |
| 6.3.4 Felder..... | 37 |
| 6.3.5 Geltungsbereich von Variablen..... | 38 |
| 6.3.6 Typ-Umwandlungen..... | 38 |

| | | |
|--------|--|----|
| 6.4 | Arithmetische Operatoren | 39 |
| 6.4.1 | Vorrang..... | 39 |
| 6.4.2 | Orientierung/Assoziativität | 40 |
| 6.4.3 | Verbundzuweisungs-Operatoren..... | 40 |
| 6.4.4 | Inkrement-/Dekrement-Operatoren..... | 41 |
| | Präfix- Postfix-Schreibweise..... | 41 |
| 6.5 | Übung 1 | 42 |
| 6.5.1 | Variablen auf Konsole anzeigen | 42 |
| 6.5.2 | Variablen unter Windows anzeigen | 42 |
| 6.5.3 | Rechner unter Windows programmieren | 42 |
| 7 | Programm-Steuerungen..... | 43 |
| 7.1 | Grundlagen..... | 43 |
| 7.2 | if | 43 |
| 7.3 | switch | 44 |
| 7.4 | break..... | 45 |
| 7.5 | continue | 45 |
| 7.6 | Programm-Schleifen..... | 45 |
| 7.6.1 | Schleifen für Wiederholungen | 45 |
| 7.6.2 | while-Schleife..... | 46 |
| 7.6.3 | do-Schleifen | 46 |
| 7.6.4 | for-Schleifen..... | 47 |
| | Geltungsbereich des Schleifenzählers | 48 |
| 7.6.5 | break, continue | 49 |
| 7.7 | Übung 4..... | 49 |
| 8 | Methoden..... | 50 |
| 8.1 | Grundlagen..... | 50 |
| 8.2 | Syntax..... | 50 |
| 8.3 | Methoden mit und ohne Rückgabewerte..... | 51 |
| 8.4 | Call by value / Call by reference | 52 |
| 8.4.1 | Der Stack und Heap..... | 53 |
| 8.5 | Bezeichner überladen | 53 |
| 8.6 | Merke | 54 |
| 8.7 | Übung 2..... | 54 |
| 8.7.1 | Rechner unter Windows mit Methoden | 54 |
| 9 | Auswahl-Anweisungen | 55 |
| 9.1 | Steuerung von Alternativen..... | 55 |
| 9.2 | Boolesche Variablen | 55 |
| 9.3 | Boolesche Operatoren | 55 |
| 9.3.1 | Negation | 55 |
| 9.3.2 | Relationale Operatoren..... | 56 |
| 9.3.3 | Logische Operatoren | 56 |
| 9.3.4 | Vorrang..... | 58 |
| 9.4 | Übung 3..... | 58 |
| 9.4.1 | Ermittlung der Schnittkräfte für Träger mit Kragarm..... | 58 |
| 10 | Arrays | 59 |
| 10.1 | Grundlagen..... | 59 |
| 10.2 | Arrays deklarieren | 59 |
| 10.3 | Zugriff auf Arrays | 60 |
| 10.3.1 | Einzelzugriff und über Schleifen..... | 60 |
| 10.3.2 | foreach-Anweisung | 60 |
| 10.3.3 | Flexibilität mit params | 61 |

| | |
|--|-----|
| 10.4 Mehrdimensionale Arrays | 62 |
| 10.5 Übung 5 | 63 |
| 10.6 ArrayList, ein Array-Typ mit dynamischer Obergrenze | 64 |
| 10.7 Übung 10 | 64 |
| 11 Sortieren | 65 |
| 11.1 Übung 6 | 65 |
| 12 Mit Dateien arbeiten | 66 |
| 12.1 Grundlagen | 66 |
| 12.2 Streams | 66 |
| 12.3 Mit Textdateien arbeiten | 67 |
| 12.3.1 Text-Daten in Datei schreiben | 67 |
| 12.3.2 Text-Daten aus Datei lesen | 69 |
| 12.4 Übung 7 | 71 |
| 12.5 Übung 8 | 71 |
| 12.6 Übung 9 | 71 |
| 13 2D-Grafik | 72 |
| 13.1 Einführung | 72 |
| 13.2 Pen | 72 |
| 13.3 Zugriff auf Grafik-Befehle | 72 |
| 13.4 Auswahl aus 2DGrafikbefehlen | 73 |
| 13.5 Funktionsgrafiken zeichnen | 74 |
| 13.6 Beispiel-Programm | 74 |
| 13.7 Übung 11 | 77 |
| 14 Verschiedene Themen | 78 |
| 14.1 Escape-Sequenzen | 78 |
| 14.2 Mathematik-Klasse | 78 |
| 15 Vorrangstufen von Operatoren | 81 |
| 16 Debugger | 82 |
| 17 Reihenfolge des Stoffes | 84 |
| 18 Musterlösungen | 86 |
| 18.1 Übung 4 | 86 |
| 18.2 Übung 5 | 93 |
| 18.3 Übung 6 | 95 |
| 18.4 Übung 7 | 97 |
| 18.5 Übung 9 | 100 |
| 18.6 Übung 8 | 102 |
| 18.6.1 Lösung mit Array | 102 |
| 18.6.2 Übung 10: Lösung mit dem dynamischen ArrayList | 106 |
| 19 Studienarbeit | 111 |
| 19.1 1. Aufgabe | 111 |
| 19.2 2. Aufgabe | 111 |
| 19.3 3. Aufgabe | 111 |

1 Ziel

Diese Einführung zeigt Ihnen grundlegende Dinge im Umgang mit dem Microsoft- Development-Environment Version 8.0 (= Visual Studio 2005) und dem Microsoft-.NET- Framework Version 2.0.

Während Sie ein kleines Programm erstellen, lernen Sie unter Anwendung der Programmiersprache C# drei Werkzeuge des Development-Environments kennen :

- den Source-Code-Editor,
- den Compiler und
- den Debugger.

Neben der Programmiersprache C# können Sie mit dieser Programmierumgebung auch andere Programmiersprachen verwenden, z.B. C++ oder Visual Basic.NET, die Nachfolge-Sprache der inzwischen abgekündigten Visual Basic - Version 6.0.

Alternativ können Sie für die Lösung der Vorlesungsbeispiele und Studienarbeiten auch andere Programmierumgebungen für C# verwenden, z.B. die von Borland. Aus Ressourcen- Gründen können diese allerdings nicht vom Bauinformatik- Labor unterstützt werden.

Sie lernen einen typischen Arbeitsablauf kennen (Quellcodedatei erstellen - Quellcode kompilieren - Syntaxfehler beseitigen - Quellcode erneut kompilieren - C# Programm ausführen und testen) und Sie sehen auch, wie man sich selbst helfen kann, wenn der Compiler einen Syntaxfehler beanstandet.

2 Hintergrund-Informationen

2.1 Einleitung

Wer sich heute das Ziel setzt, Programme für das Internet und für das Betriebssystem Windows zu schreiben, für den ist die noch junge Programmiersprache C# (sprich "C sharp") zu einer interessanten Option geworden.

Wer sich mit C# beschäftigt, bekommt es bald mit Begriffen wie .NET (sprich "Dot Net") oder "Common Language Runtime System" zu tun.

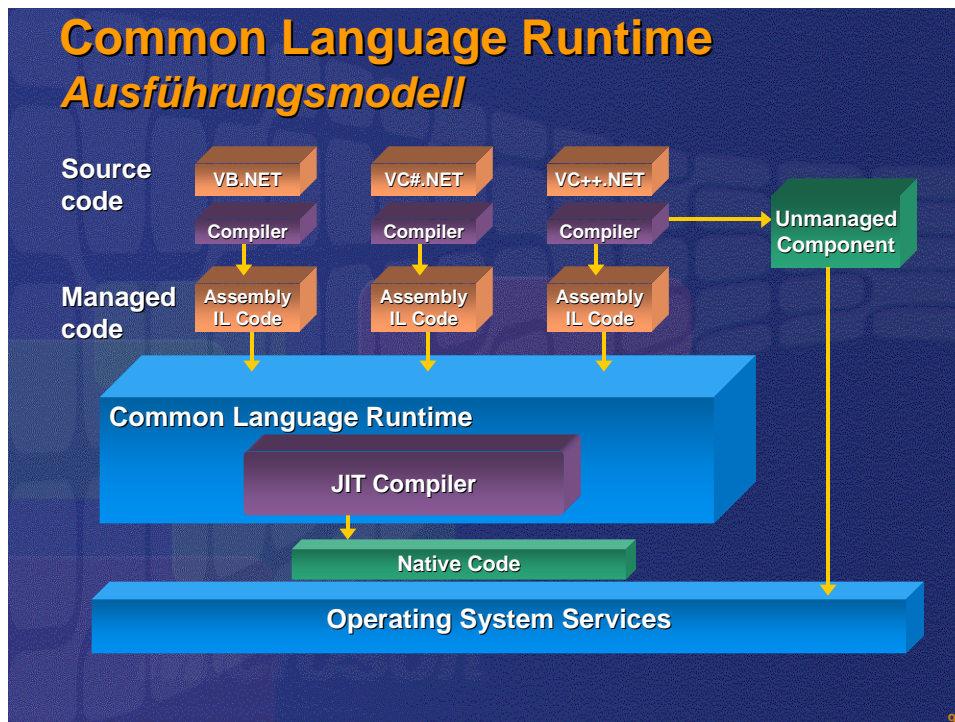
Dieser Abschnitt will Hintergrund-Informationen vermitteln, und damit die Furcht vor einigen dieser "Buzz-Words" nehmen.

Außerdem soll C# vorgestellt und anderen Programmiersprachen gegenübergestellt werden.

2.2 Was ist .NET?

Microsoft hat mit .NET eine neue Technologie eingeführt, mit deren Hilfe unterschiedliche Programme und Programmbausteine mit wesentlich geringerem Aufwand integriert werden können, als das bisher der Fall war. Darüber hinaus stehen eine neue Entwicklungsumgebung und viele neue Werkzeuge zur Verfügung, die das Programmieren wesentlich erleichtern und wirtschaftlicher machen.

Der Kern von .NET ist das sog. "Common Language Runtime System" (CLR). Dieses kann als eine Plattform verstanden werden, auf der sich einzelne Programnteile treffen, die in verschiedenen Sprachen programmiert sein können. Über die CLR können diese Programnteile dann, wenn sie vorher mit einem entsprechendem Compiler übersetzt wurden, zusammengebunden werden.

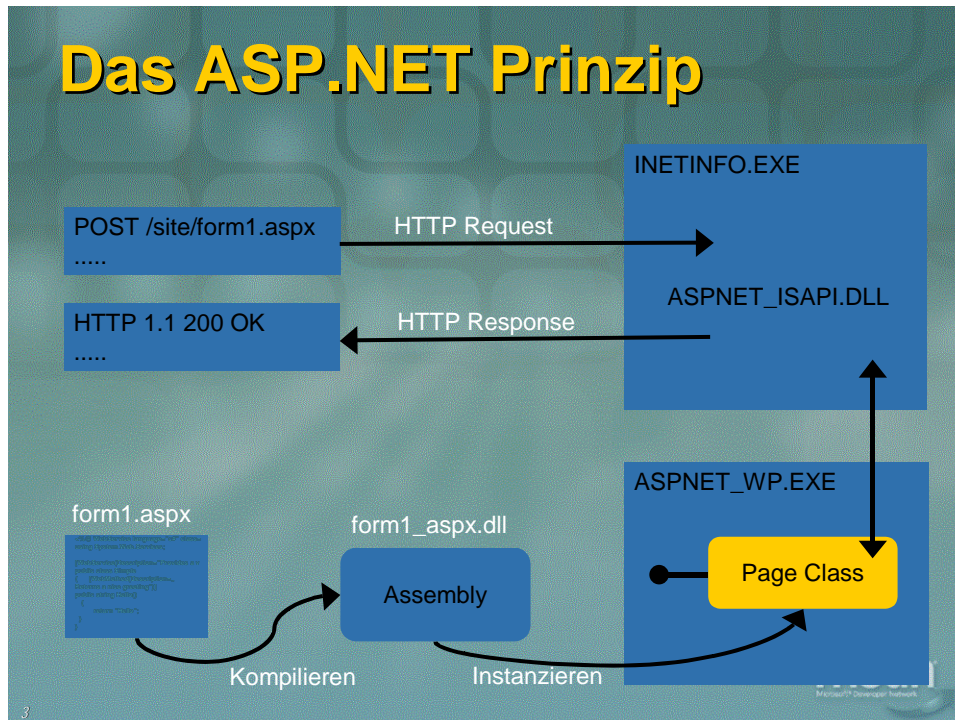


Quelle: Dirk Primbs, .NET Strategy & Developer Group, Microsoft

Auch die Unterstützung verschiedener Rechnertypen, wie Desktops, Tablett-PCs, Mobile Computer und Smartphones wird mit .NET wesentlich vereinfacht.



Nicht zuletzt kann auch die Internet- Programmierung mit dem .NET- Framework sehr wirtschaftlich umgesetzt werden. Dazu steht als Erweiterung ASP.NET (Active Server Pages) zur Verfügung. Häufig wird im Zusammenhang mit dieser Technologie auch der Begriff "Web Matrix" verwendet.



Quelle: Darius Parys, .NET Strategy & Developer Group, Microsoft

2.3 C# und anderen Programmiersprachen

2.3.1 C-Syntax, Objekte, Internet

C# gehört zu den Programmiersprachen, die auf der C- Syntax aufbauen. D.h. alle Programmierer, die Sprachen wie C, C++ oder Java gelernt haben, werden sich auch in der Syntax von C# schnell zurechtfinden. Umgekehrt wird auch ein C# - Programmierer ohne all zu große Mühe auf Java oder C++ umsteigen können.

C# ist eine Sprache, die von früheren Sprachen gelernt hat und sie entspricht als rein objektorientierte Sprache dem Stand der Technik. Wer Sprachen wie C++, Java und Visual Basic kennt, wird in C# etliche Konzepte wiederfinden, und man könnte sagen, dass in C# die Vorteile früherer Sprachen zusammengefasst wurden.

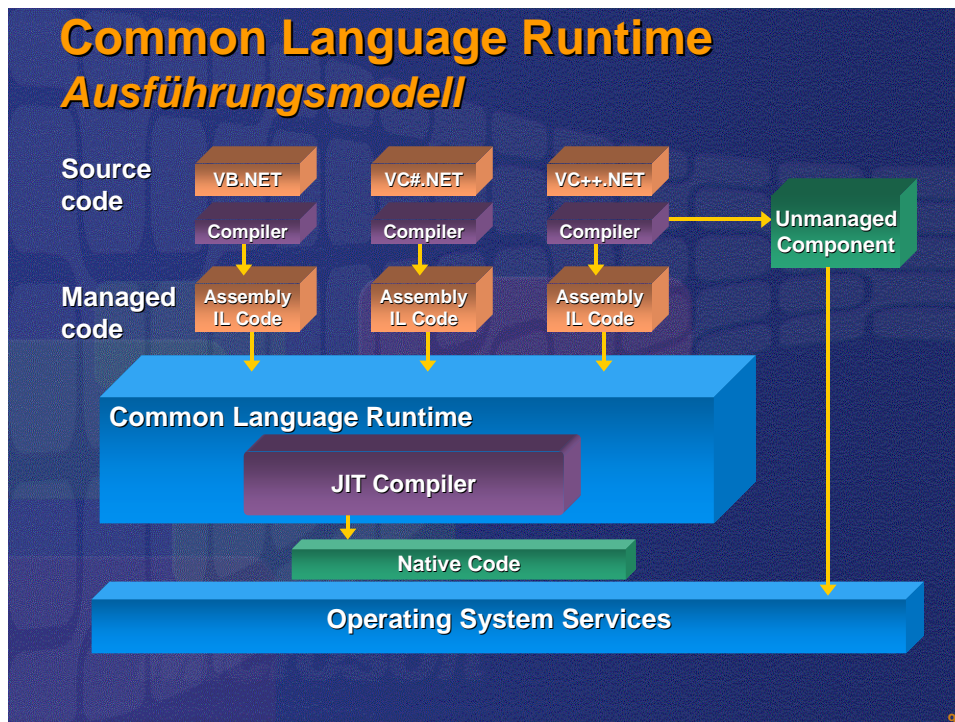
C# ist eine Sprache, die darauf optimiert wurde, möglichst wirtschaftlich Anwendungen für das Internet und für das Betriebssystem Windows zu schreiben.

2.3.2 C# und Java

Die engste Verwandtschaft hat C# mit Java, und viele Konzepte aus Java wurden in C# übernommen. Der Programm- Code von einfacheren Java-Programmen kann praktisch 1:1 übernommen werden. Auch die grundsätzliche Architektur ist in beiden Sprachen sehr ähnlich.

Mit Java wurde das Konzept eingeführt, einen Programm- Quell-Code zunächst in einen plattform-unabhängigen Byte-Code zu übersetzen, der dann vor oder während der Ausführung auf der Zielplattform zum endgültigen, Maschinen-spezifischen Code übersetzt wird.

Ein sehr ähnliches Konzept ist das bereits oben beschriebene "Common Language Runtime System" (CLR). Auch C# und alle anderen .NET- konformen Sprachen werden zuerst in einen allgemeingültigen Bytecode übersetzt, und die Komponenten dann erst auf der Ebene der CLR endgültig verbunden und übersetzt. Dadurch ist es möglich, Komponenten zu integrieren, die in verschiedenen (.NET- konformen) Sprachen programmiert wurden.



Quelle: Dirk Primbs, .NET Strategy & Developer Group, Microsoft

In der obigen Darstellung wird zwischen "Managed" und "Unmanaged" Code unterschieden. Managed Code ist ein Byte-Code, der über die CLR verbunden und übersetzt werden kann. Unmanaged Code ist Maschinenspezifischer Code, der sich nicht den Regeln der CLR unterwirft, und direkt auf das Betriebssystem zu greifen kann.

In der Regel wird man für Anwendungs-Programme immer Managed Code verwenden und Unmanaged Code nur dann, wenn man System-nah entwickeln muss.

2.3.3 Programmiersprachen und Standards

Während frühere Programmiersprachen häufig in Forschungsprojekten und Hochschulen entwickelt und dann zum Standard erhoben wurden (und damit eine gewisse Neutralität ausstrahlen), haben in den letzten Jahren diese Rolle Unternehmen übernommen. So wurde Java von SUN und C# von Microsoft entwickelt, die damit jeweils bestimmte wirtschaftliche Ziele verfolgen.

Entscheidender jedoch, als die Frage, wer eine Sprache entwickelt hat, ist, dass Programmiersprachen, nachdem sie eine ausreichende Reife erreicht haben, von einem unabhängigen und allgemein anerkannten Standardisierungs-Institut zum Standard erklärt werden. Dies ist deswegen wichtig, um die Sprache vor all zu häufigen Versionswechsel zu schützen, und um damit einen Investitionsschutz für die Programmierer und Unternehmen aufzubauen, die sich für eine bestimmte Sprache entschieden haben.

Änderungen oder Erweiterungen sind bei standardisierten Sprachen nur noch über das zuständige Standardisierungs-Institut möglich, nachdem in öffentlich zugänglichen Revisionen die Zustimmung gegeben wurde. Eine standardisierte Sprache gehört also nicht mehr nur einem Unternehmen und kann deshalb nicht mehr von nur einem Unternehmen nach Belieben geändert werden.

Wie schon früher C und C++ sind heute sowohl Java, als auch C# standardisierte Sprachen.

2.3.4 Visual Basic 6.0 und Visual Basic.NET

Anders sieht dies bei Visual Basic aus. Diese Sprache "gehört" einem Unternehmen (Microsoft) und wurde nicht zu einem offiziellen Standard erhoben. Visual Basic Programmierer können deshalb auf eine langjährige und oft leidvolle Erfahrung von häufigen und teilweise gravierenden Änderungen zurückblicken.

Die wohl größte Änderung wurde den Basic- Programmierern mit der Abkündigung von Visual Basic 6.0 und der Einführung von Visual Basic.NET zugemutet. Das neue Visual Basic.Net hat mit dem alten Visual Basic bis zu einem gewissen Grad nur noch die Syntax gemeinsam, sonst aber ist alles anders geworden.

Der Grund für diese radikale Änderung liegt darin, dass in Visual Basic.NET nun Strukturen eingeführt worden, die in anderen Programmiersprachen schon längst zum Alltag gehören. Visual Basic.NET ist jetzt auch eine objektorientierte Sprache geworden und in ihrem inneren Aufbau praktisch identisch zu C#.

Visual Basic.NET musste diesem Umbau deswegen unterzogen werden, damit man Programmkomponenten, die damit geschrieben werden, ohne Aufwand über das .NET- Framework und die CLR mit Komponenten verbinden kann, die mit einer anderen Sprache programmiert werden, z.B. mit C#.

Dieser Schritt sollte von "traditionsbewussten" Programmierern als deutlicher Fingerzeig verstanden werden, dass die Zeiten der reinen strukturierten Programmierung nun endgültig der Vergangenheit angehören, und die objektorientierte Programmierung der Stand der Technik ist.

Während Visual Basic 6.0 also eindeutig ein "Auslaufmodell" ist, spielen bei der Wahl der Nachfolgesprache Geschmacksfragen oft eine entscheidende Rolle. Kann man sich mit einer C- Syntax anfreunden oder will man lieber doch bei einem vertrauten optischen Erscheinungsbild bleiben?

Wenn man also schon so viel neues lernen muss, und es rein technisch gesehen keine Unterschiede zwischen Visual Basic.NET und C# gibt, liegt der Gedanke nahe, bei diesem Schritt auch gleich auf die C-Syntax zu wechseln. Auf alle Fälle gewinnt man dadurch erheblich mehr persönliche Kompatibilität mit anderen Programmiersprachen und letztlich Freiheit durch den Abschied von einer Nicht-Standard- Sprache.

2.4 Warum ist Programmieren heute komplizierter?

Einige werden sich vielleicht fragen, warum das Programmieren so aufwändig geworden ist, warum man selbst für ein einfaches Programm soviel beachten muss, und ob das nicht auch einfacher geht. Besonders, wenn man früher vielleicht mit einem der Basic-Dialekte und - Umgebungen gearbeitet hat, mag einem der Aufwand übertrieben vorkommen, wenn man nur ein kleines Programm schreiben möchte.

In der Tat ist der Einsatz einer Programmierumgebung wie MS Visual Studio wie mit "Kanonen auf Spatzen" geschossen, wenn man damit wirklich nur ein kleines Programm schreiben möchte. Denkt man jedoch einen Schritt weiter, dass man nach den ersten Anfängen doch auch etwas anspruchsvollere Programme erstellen möchte, lernt man die Vorteile einer leistungsfähigen Programmierumgebung sehr schnell schätzen. Je mehr Quell-Code-Dateien sich ansammeln, desto mehr wird man auch die Notwendigkeit einer Projektverwaltung erkennen.

Nicht nur die Programmierumgebung erscheint zunächst aufwändiger, auch in den modernen Programmiersprachen, wie Java oder C#, sind zunächst wesentlich mehr Angaben zu machen, als früher etwa in Basic, bis auch nur ein kleines Programm läuft. Dies mag man zunächst als überflüssig erachten, jedoch sieht man den Sinn bald ein, wenn man erkennt, um wieviel leistungsfähiger moderne Programmiersprachen geworden sind.

Sobald man in die Windows- oder Internet- Programmierung einsteigt, sieht man, dass man z.B. mit C# sogar wesentlich schneller ist als früher in Basic.

Hingegen ist der Aufwand für den eigentlichen funktionalen Programm- Code gleich geblieben. Ein mathematischer Algorithmus ist in C# auch nicht schwerer zu formulieren als in Basic.

Es ist also nicht das eigentliche Programmieren komplizierter geworden, sondern das Umfeld, in dem Programme heute ausgeführt werden, ist leistungsfähiger und damit leider auch komplizierter geworden. Einen Teil des zusätzlichen Aufwandes federn moderne Programmierumgebungen wieder ab.

Man muss also in den sauren Apfel beißen und einen etwas höheren Aufwand betreiben, als früher, dafür bekommt man aber auch wesentlich mehr zurück.

3 Begleit-Literatur

Eine Auswahl möglicher Begleit-Literatur.

Diese Auswahl ist als Vorschlag und nicht als verbindlich zu verstehen. Wer ein C#-Buch entdeckt, das ihn mehr anspricht, als die hier vorgeschlagenen, kann das ebenso gut verwenden. Alle Bücher beschreiben letzt endlich die gleiche Technologie, sie wird nur in jedem Buch etwas anders verpackt.

Für den Anfang, zum Nachschlagen und zum Selbst-Studium:

- Visual C#.NET
- Walter Doberenz, Thomas Kowalski
- Grundlagen und Profiwissen
- Hanser Verlag
- *Schritt-für-Schritt-Anleitungen, zusätzlicher sehr guter "Kochbuch"-Teil, entstanden und erprobt im Studienbetrieb*
- Microsoft Visual C#, Schritt für Schritt
- John Sharp, Jon Jagger
- Microsoft Press
- *Schritt-für-Schritt Anleitung für den Anfänger und Profi*

Wenn jemand tiefer einsteigen möchte:

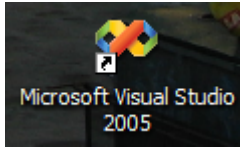
- Windows-Programmierung mit C#
 - Charls Petzold
 - Microsoft Press
 - *Eines der Standard-Bücher für die Windows- Programmierung*

Für erfahrene Programmierer:

- Programmieren mit C#
 - Jesse Liberty
 - O'Reilly
 - *Gut zu lesender kompakter Einstieg*
- C#
 - Eric Gunnerson
 - Galileo Computing
 - *Kompakter Einstieg und Referenz*
- C# in a Nutshell
 - Peter Drayton u.A.
 - O'Reilly
 - *Sehr ausführliche Referenz zum Nachschlagen, weniger als Einstieg gedacht.*

4 Schritt für Schritt zum ersten C#-Programm

4.1 Visual Studio.NET



Microsoft Visual Studio.NET ist eine Programmierumgebung, die folgende Komponenten enthält:

- Common Language Runtime
- Projektverwaltung
- Zur Verwaltung der Dateien für Quell- und Maschinencode verschiedener Programmierprojekte
- Assistenten
- Unterstützt bei der Erstellung von verschiedenen Programmtypen wie Windows- oder Internet-Anwendungen
- Editor
- Zur komfortablen Eingabe des Quell-Codes
- Compiler
- Zum Übersetzen in den Byte- oder Maschinen-Code
- Debugger
- Zur Fehlersuche in Programmen

Achtung Verwechslungsgefahr:

Bitte beachten Sie, dass die Vorgänger-Version Microsoft Developer Studio hieß, die sich evtl. ebenfalls noch auf dem von Ihnen verwendeten Rechner befindet, mit der Sie aber keinen C#-Code erzeugen können:



4.2 Schritt 0: Persönlichen Ordner anlegen

Legen Sie sich einen **persönlichen Ordner** an, unter dem Sie Ihre Programmanwendungen speichern werden. Der Ordner sollte wie folgt benannt werden:

Name.Vorname_StudienGruppe_Semester

Beispiel:

Huber.Josef_1D_WS03-04

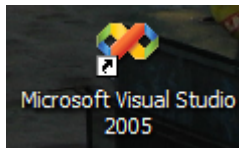
Am besten legen Sie diesen Ordner unter dem Ordner "**Raeume**" an, dann haben sie von allen Räumen des Bauinformatik-Labors Zugriff auf Ihre Daten, egal welcher Rechner gerade frei ist. Außerdem können Sie dann Ihre Daten einfach sichern, indem Sie den kompletten Ordner sichern.

Sie sollten auch die Daten, die Sie mit anderen Programmen erzeugen, unter diesem Ordner ablegen. Deshalb ist es ratsam, dass Sie auch gleich noch einen weiteren Unterordner, z.B. "**Programmieren**" als Sammler für Ihre Programme anlegen, die Sie im Laufe der Zeit erstellen werden. Ihre persönliche Ordernstruktur könnte also wie folgt aussehen:

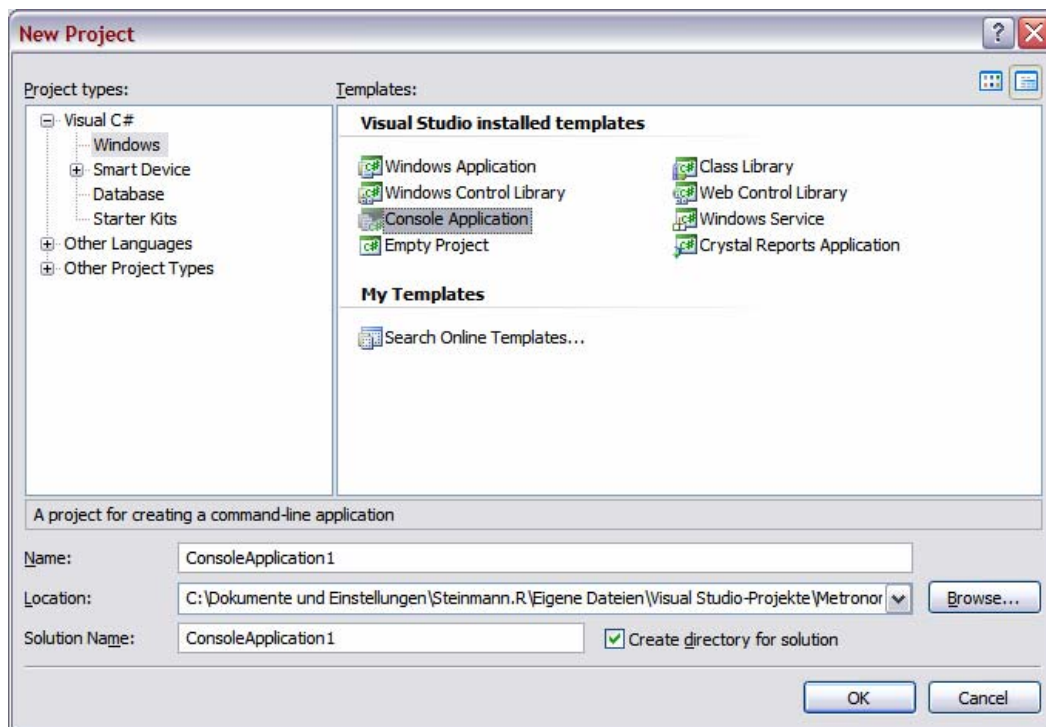
...\Raeume\Huber.Josef_1D_WS03-04\Programmieren

4.3 Schritt 1: Anlegen eines Projektes

Starten Sie die Microsoft Entwicklungsumgebung Visual Studio.NET



und rufen Sie das Menue **Datei - Neu - Projekt** auf, bzw. **File - New -Project**, falls die Englische Version installiert ist:



Wir wollen als erstes ein Programm schreiben, das auf der Konsole, also auf dem schwarzen System-Bildschirm läuft, und dort einen Text ausgibt:

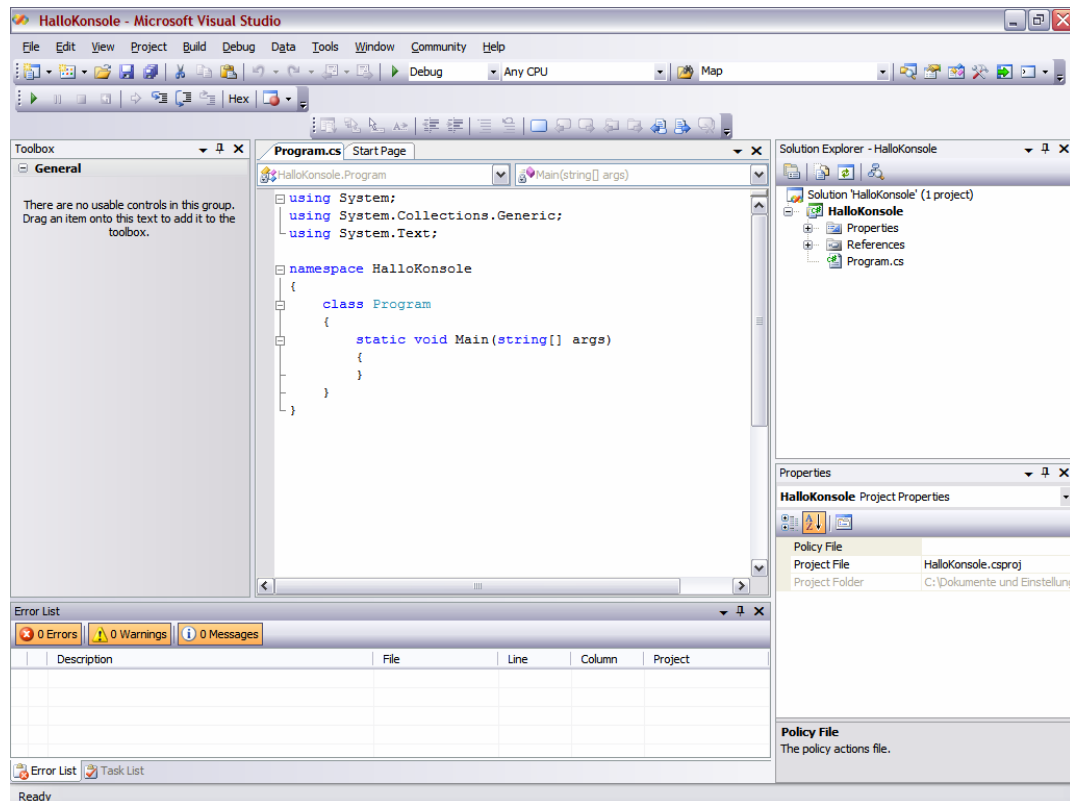
- Klicken Sie in der Liste der Projekttypen auf Visual C#-Projekte
- Klicken Sie im Vorlagen-Feld auf das Symbol "Konsoleanwendung"
- Geben Sie bei Name: einen Namen für Ihr Projekt ein, z.B. **HalloKonsole** . I.d.R. ist dieser Projektname identisch mit dem Namen Ihrer Anwendung. Bitte verwenden Sie keine Zahlen als erstes Zeichen des Namens und keine Sonderzeichen. Eine Folge von Worten unterscheidet man üblicher Weise durch Groß- und Kleinbuchstaben.
- Wählen Sie unter Speicherort: Ihren persönlichen Ordner für Programmieren, den Sie im Schritt 0 angelegt haben. Klicken Sie dazu auf den Knopf "Durchsuchen..."
- Klicken Sie dann auf OK

Es öffnet sich jetzt ein neues Fenster, das im nächsten Schritt erklärt wird.

Außerdem wurde unter Ihrem Ordner für das Programmieren automatisch ein neuer Ordner mit dem Namen "HalloKonsole" angelegt (das können Sie mit dem File- Explorer überprüfen). Dieser neue Ordner enthält neben ihrer Quell-Code-Datei eine Reihe von weiteren Dateien und Unterordnern, die für die Integration in die CLR und die Projektverwaltung erforderlich sind.

4.4 Schritt 2: Quell-Code verstehen

Folgendes Fenster wurde am Ende des vorherigen Schrittes geöffnet, es zeigt das neue Projekt:



Dieses Fenster enthält bereits die wichtigsten C#-Quellcode-Kommandos, die für eine Konsole- Anwendung erforderlich sind. Man muss jetzt diesen Quellcode nur noch auf seine Bedürfnisse anpassen.

Sie sehen, dass der Mehraufwand, der bei einer modernen Programmiersprache wie C# gegenüber früheren Basic-Dialekten erforderlich ist, von einer modernen Programmierumgebung zumindest zum Teil wieder ausgeglichen wird. Es muss also letztendlich kaum mehr editiert werden.

Erklärung des ersten automatisch generierten Quell-Codes:

- Kommandos, die **blau** angezeigt werden, sind reservierte, C#-eigene Kommandos und dürfen nur für ihren bestimmten Zweck, nicht aber z.B. für eigene Variablen-Namen verwendet werden.
- Zeilen in **grün**, die mit **//** eingeleitet werden sind Kommentarzeilen. Es ist ratsam, seinen Quell- Code ausreichend mit Kommentaren zu versehen, die beschreiben, was man sich beim Programmieren gedacht hat. Das hilft einem selber, wenn man den Quell-Code nach einiger Zeit wieder verstehen muss, und hilft einem Kollegen, der evtl. einmal mit dem Quell- Code arbeiten muss.
- Zeilen in **grün**, die mit **///** eingeleitet werden sind spezielle Kommentarzeilen, die zur automatischen Generierung einer Dokumentation von Programmen verwendet werden. Hierzu wird XML verwendet und mit speziellen XML-Tags (= XML-Kommandos), können bestimmte Teile einer Dokumentation gesteuert werden. So beschreibt z.B. der Kommentar zwischen den `<summary>` ... `</summary>` Tags eine kurze, einzeilige Beschreibung einer Klasse oder Methode. Daneben gibt es noch weitere XML-Tags, die die Generierung einer Dokumentation steuern. Später kann der XML-Code aus dem Programmtext herausgefiltert und daraus eine Dokumentation generiert werden, die man mit Internet-fähigen Viewern oder Browsern anzeigen kann.
- **using System;** sagt aus, dass dieses Programm auf Systembefehle zugreifen möchte, z.B. die Ausgabe von Text auf den Bildschirm. Damit nicht jeder Programmierer diese Programmieraufgabe von neuem lösen muss, wird der Code für solche häufig vorkommenden Befehle in einer Bibliothek mit C# mitgeliefert. Neben der System-Bibliothek gibt es noch eine Reihe weiterer Bibliotheken. Der Compiler braucht nun einen Hinweis, wo er mitgelieferten Code suchen soll, um diesen dann beim Übersetzten dem Programm hinzufügen zu können. In diesem Fall wird dem Compiler mitgeteilt, dass er in der System- Bibliothek suchen soll.

- **namespace** **HalloKonsole** bewirkt, dass die in diesem Programm verwendeten Namen im Kontext dieses Programmes eindeutig verstanden werden. Das Konzept des "Namespaces" kann man sich am Beispiel von Vornamen verdeutlichen: Wenn in einer Familie ein bestimmter Vorname gerufen wird, fühlt sich i.d.R. eine bestimmte Person angesprochen. Wird der selbe Vorname aber auf der Straße gerufen, werden sich evtl. mehrere Personen umdrehen. Hier muss man also den Familiennamen hinzufügen, um mit einiger Wahrscheinlichkeit die richtige Person anzusprechen.
- Alle ausführbaren Programmzeilen werden in der C-Syntax immer mit ; angeschlossen.
- C# ist case-sensitiv, d.h. Groß- und Kleinschreibung werden unterschieden. Z.B. kann eine Variable, die mit dem Namen *Test* eingerichtet wurde, im weiteren Programm- Code nicht mit *test* verwendet werden.
- Sämtlicher C#-Quell-Code wird in Klassen strukturiert und gespeichert. Dem vorliegenden Code wurde schon gleich eine Klasse hinzugefügt:
class Class1 sagt aus, dass eine Klasse mit dem Namen Class1 angelegt wurde. Alle Klassen werden jeweils in einer Datei abgelegt, die den Klassennamen enthält und die Datei- Endung **.cs** trägt. Im vorliegenden Fall wurde also die Klasse Class1 in der Datei Class1.cs abgelegt.
 Klassen enthalten Methoden (siehe nächster Punkt) und Eigenschaften, so wie Daten (sog. Felder)
- Die Anweisung für Klassen ist als **Kopf einer Klasse** zu verstehen, der von einem **Rumpf** gefolgt wird, der den eigentlichen Programm-Code enthält. Der Rumpf wird mit { } eingefasst. Auf die Anweisung für den Klassenkopf folgt also kein ; sondern ein Paar von geschweiften Klammern. Man sollte zueinander passende geschweifte Klammern immer in einer Spalte plazieren und den folgenden Programm-Code um einen Tabulator einrücken. Dadurch erhält man gut leserlichen Programm- Code und man sieht mit einem Blick wo ein Rumpf beginnt und wo er endet. Der Editor unterstützt diese leserliche Schreibweise durch automatisches Einfügen von Tabulatoren an geeigneter Stelle.
- Jede Klasse enthält mindestens eine oder mehrere **Methoden**. Synonym zum Begriff "Methode" kann man die Begriffe "Funktion" oder "Unterprogramm" verstehen. Allerdings werden diese Begriffe in der objektorientierten Programmierung nicht mehr verwendet, da hier mit Klassen eine weitere Strukturierungsmöglichkeit eingeführt wurde. Die Begriffe "Funktion" oder "Unterprogramm", die zu Zeiten der strukturierten Programmierung eingeführt wurden, würden nicht eindeutig beschreiben, ob man damit eine Klasse oder eine darin enthaltene Methode meint. Weitere Erklärungen zu Methoden später.

Methoden enthalten den eigentlichen ausführbaren Programmcode. Jede Klasse muss mindesten eine Methode enthalten. Der erste Einstiegspunkt einer Anwendung ist die Methode "**Main**". Jede Anwendung muss deswegen eine Main-Methode enthalten. Besteht eine Anwendung aus nur einer Klasse mit nur einer Methode, muss diese deswegen den Namen **Main** tragen.

static void Main (string[] args) ist der **Kopf** für einer Main- **Methode**. Diesem folgt ein **Rumpf**, der wieder mit { } eingefasst wird. D.h., auch der Kopf einer Methode wird nicht mit einem ; abgeschlossen, sondern auch ihm folgt ein Paar von geschweiften Klammern. Nach dem Namen einer Methode folgt ein Paar runder Klammern (), das die **Liste der Parameter** einfaßt. Auch eine leere Parameter-Liste muss mit () eingefaßt werden.

Weiter Erklärungen zu **static** , **void** , **string[] args** später, wenn Methoden detailliert erklärt werden.

4.5 Schritt 3: Quell-Code eingeben

- Der automatisch generierte Quell-Code zeigt Ihnen mit dem ToDo-Kommentar an, wo Sie Ihren Programm-Code einfügen sollen. Löschen Sie dazu die drei Kommentarzeilen mit dem ToDo- Kommentar. Der Quell-Code sollte dann so aussehen:

```
static void Main( string [] args)
{
}
```

- Setzen Sie den Cursor hinter die erste geschweifte Klammer { und drücken Sie die Enter-Taste.
- Der Editor rückt automatisch um einen Tabulator ein, um Sie beim Schreiben von übersichtlichem Programm-Code zu unterstützen.
- Tippen Sie dann den Befehl **Console** ein. Console ist eine mitgelieferte Klasse, die in der System-Bibliothek enthalten ist. Die Systembibliothek wird ja weiter oben, wie im letzten Schritt erklärt, mit **using** bereits angekündigt.

- Die Klasse "Console" enthält eine Reihe von Methoden. Wir wollen mit unserem Programm einen Text ausgeben und benötigen deswegen eine passende Funktion aus der Klasse "Console". Die Namen der Methoden einer Klasse werden beim Aufruf durch einen "." vom Klassennamen getrennt.

Diese Entwicklungsumgebung enthält sog. IntelliSense-Listen, die einem bei der Auswahl geeigneter Methoden aus einer Klasse helfen:

Sobald Sie hinter **Console** einen Punkt eingeben, öffnet sich eine Liste der möglichen Methoden. Scrollen Sie nach unten und doppelklicken sie auf **WriteLine**.

- Ihr Programm-Code sollte jetzt so aussehen:

```
static void Main( string [] args)
{
    Console.WriteLine
}
```

- Geben Sie jetzt eine Runde Klammer auf ein (
- Wieder meldet sich IntelliSense, denn in der Klasse Console gibt es mehrere Methoden mit dem Namen WriteLine. Diese unterscheiden sich jedoch nur auf Grund ihrer unterschiedlichen Parameterliste. (Man spricht hier von sog. überladenen Methoden, mehr dazu später, wenn Methoden detailliert erklärt werden)
- Schließen die jetzt die Runde Klammer) und schließen Sie diese normale, ausführbare Programmierzeile mit einem ; ab.
- Tipp:** Wann immer sie eine Klammer öffnen, geschweift oder rund, sollten Sie sie immer auch gleich wieder schließen { } (), und dann erst den Inhalt einfügen. Dadurch vermeidet man Klammer- Fehler.
- In die runden Klammern setzten Sie **"Hello World!"**
- Jetzt sollte Ihr Programm wie folgt aussehen:

```
static void Main( string [] args)
{
    Console.WriteLine("Hello World!");
}
```

- Ergänzen Sie noch ein Kommando, mit dem eine Eingabe von der Tastatur abgefragt mit. Wir werden die Eingabe nicht weiter auswerten, halten damit aber das Programm an, so dass man das man sich die Ausgabe ansehen kann. Nach Drücken der Enter-Taste läuft das Programm weiter und wird beendet, das Ausgabefenster wird geschlossen.

```
static void Main( string [] args)
{
    Console.WriteLine("Hello World!");
    Console.ReadLine();
}
```

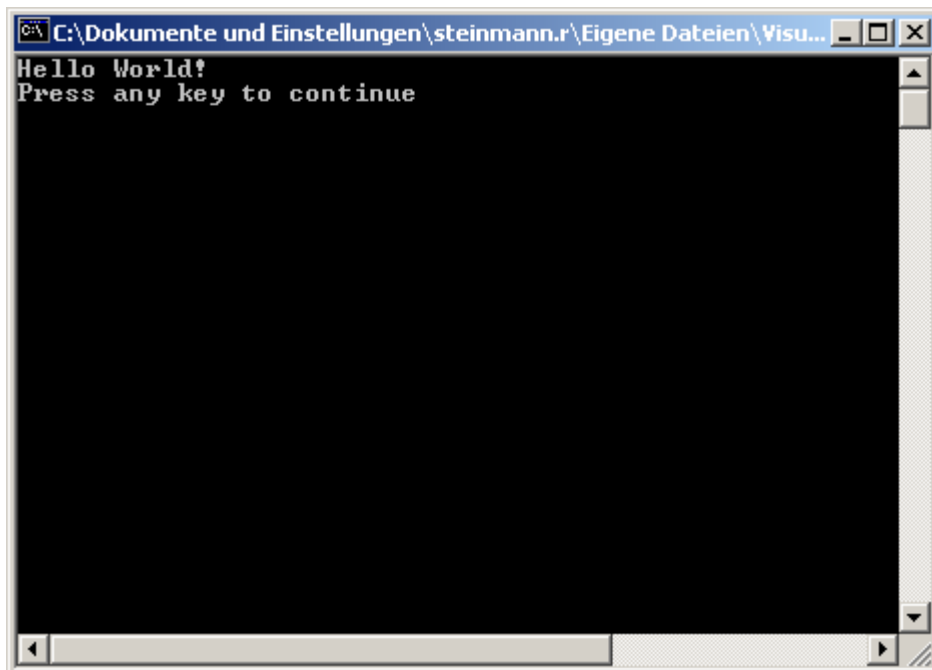

4.6 Schritt 4: Anwendung erstellen und ausführen

Damit das eben editierte Programm ausgeführt werden kann, muss es zunächst compiliert werden.

- Wählen Sie dazu das Menü **Erstellen - Projektmappeerstellen**. Dieser Befehl speichert auch automatisch den Quell-Code ab.
- Wenn Sie sich im letzten Schritt nicht vertippt haben, gibt Ihnen der Compiler die folgende Meldung aus:
Build abgeschlossen -- 0 Fehler, 0 Warnungen
- Sollten Sie sich aber vertippt haben, werden eine oder mehrer Fehler angegeben. Schauen Sie sich die erste Fehlermeldung an und versuchen Sie den Grund zu verstehen. Oft ist es nur eine Kleinigkeit, die man übersehen hat, die aber eine ganze Liste von Fehlern auslöst. Manchmal genügt es schon den Grund für den ersten Fehler zu beheben, um auch alle Folgefehler auszuschalten.
- Falls Sie gleich beim ersten Mal ohne Fehler durchgekommen sind, können Sie jetzt absichtlich einen Fehler einbauen, um zu sehen, wie Fehlermeldungen des Compilers aussehen. Sie können z.B. den ; hinter Ihrer Zeile löschen, oder die " beim Hello World! Text.
- Erzeugen Sie nach diesem Experiment wieder fehlerfreien Programm-Code.

Jetzt können Sie Ihr Programm ausführen:

- Wählen Sie das Menü **Debuggen - Starten ohne Debuggen**.
 1. Wenn Sie nur den Befehl Starten wählen, wird Ihr Programm nur einmal kurz aufblinken.
- Das Programm sollte auf einem Systembildschirm gestartet werden:



- Wenn Sie auf eine beliebige Taste klicken, schließt sich die Konsole wieder.

4.7 Architektur eines C#-Programmes

Die folgende Darstellung soll schematisch noch einmal die wichtigsten Strukturen und Komponenten verdeutlichen, aus denen unser C#-Programm zusammengebaut wurde:

```
// Ankündigung von Bibliotheken, die vorgefertigte Klassen enthalten,
// die im Folgenden verwendet werden
using Bibliothek;

// namespace sorgt dafür,
// dass Namen in einem bestimmten Kontext eindeutig sind.
// In der Entwicklungsumgebung wird der Name des Projektes
// auch als Name des Namespaces übertragen.
namespace NameAnwendung
{
    //Kopf und Rumpf einer Klasse
    class ClassName
    {
        //Kopf und Rumpf der ersten Methode
        static void Main (string[] args)
        {
            Ausführbare Anweisung1;
            Ausführbare Anweisung2;
            ...;
        }

        // Kopf und Rumpf weiterer Methoden
        MethodenName1 (Parameterliste)
        {
            ...;
            ...;
        }

        MethodenName2 (Parameterliste)
        {
            ...;
            ...;
        }
    }
}
```

4.7.1 Namespaces

Wie bereits erwähnt, sorgen Namespaces dafür, dass bestimmte Namen im Kontext des Namespaces eindeutig sind. Sie sind praktisch ein Behälter (in der Fachsprache verwendet man hier den Begriff Container) für andere Bezeichner von Klassen und Methoden.

Der im Beispielprogramm verwendete Aufruf:

```
Console.WirteLine("Hello World");
```

müsste ganz korrekt und ausführlich eigentlich so aussehen:

```
System.Console.WirteLine("Hello World");
```

Durch die Angabe von

```
using System;
```

wird jedoch der Bezug zu einem Namespace "System" hergestellt (die Bibliothek mit diesem Namespace ist bereits im Lieferumfang von C# enthalten). Durch diesen Bezug ist es nicht mehr notwendig, bei jedem Aufruf von `Console.WirteLine("Hello World")` auch noch `System` davorzusetzen, sondern es ist klar, dass es sich nur um diesen bestimmten `Console.WirteLine`-Befehl handeln kann. Die Verwendung des `using`-Kommandos erspart also Tipp- Arbeit.

Kleine Übung:

Sie können die Wirkung von using ausprobieren:

Setzen Sie dazu vor **using System** und **[STAThread]** ein Kommentarzeichen:

```
// using System;
...
//[STAThread]
```

Damit sind diese beiden Programmzeilen jetzt unwirksam und werden vom Compiler übersprungen. Das Programm kennt jetzt also nicht mehr den Kontext zum Namespace "System".

Wenn Sie jetzt die Projektmappe neu erstellen und damit das Programm neu übersetzen, werden Sie eine Meldung bekommen, dass 'Console' nicht gefunden werden konnte. Setzen Sie jetzt vor

Console.WriteLine("Hello World"); noch **System.**, so dass die komplette Zeile so aussieht:

```
System.Console.WriteLine("Hello World");
```

und erstellen die Projektmappe noch einmal. Jetzt kann das Programm wieder fehlerfrei übersetzt werden.

4.8 Gratulation

Gratulation, Sie haben die wichtigsten Schritte der Programmierung mit C# erfolgreich durchlaufen!

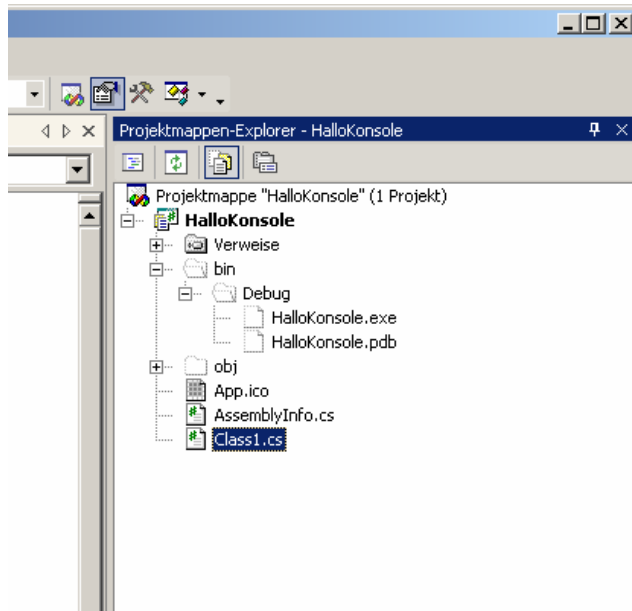
Sie ...

- können jetzt schon einigen C#-Code zumindest im Ansatz verstehen
- haben die wichtigsten Strukturierungsmöglichkeiten (Klassen und Methoden) kennengelernt
- wissen, wie Programmcode für den Kopf und Rumpf einer Klasse und einer Methode aussieht und wie dieser mit { } gekennzeichnet wird
- wissen, dass
- Klassen Methoden enthalten, und immer mindestens eine Methode enthalten sein muss,
- die zuerst aufgerufene Methode Main heißen muss,
- Methoden immer eine Parameterliste haben, die mit () gekennzeichnet wird, selbst wenn die Liste leer ist.
- wissen, dass normale ausführbare Programmzeilen mit ; abgeschlossen werden
- haben das IntelliSense kennegelernt
- können ein Programm übersetzen
- könne ein von Ihnen erstelltes Programm ausführen
- haben gesehen, dass selbst ein sehr einfaches C# Programm zwar einiges an "Overhead" benötigt, dass aber mit einer komfortablen Entwicklungsumgebung der Aufwand dafür minimal ist.

4.9 Wo liegt das fertige Programm?

Vielleicht wird Sie jetzt interessieren, wo Ihr ausführbares Programm abgelegt wurde.

In der Entwicklungsumgebung gibt es dazu einen **Projektmappen-Explorer**. Sollte er nicht sichtbar sein, können Sie ihn unter dem Menu **Ansicht** aktivieren.



Ihre Projektmappe heißt HalloKonsole.

Darunter sehen sie drei Ordner und drei Dateien, u. A. auch die von Ihnen bearbeitet Quell-Datei Class1.cs .

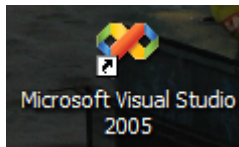
Sie können nun die Ordner bin und Debug, bzw. obj und Debug öffnen und sehen dann die ausführbare Datei HalloKonsole.exe, sowie weitere Dateien, die für das Debuggen benötigt werden.

5 Ein erstes Windows-Programm

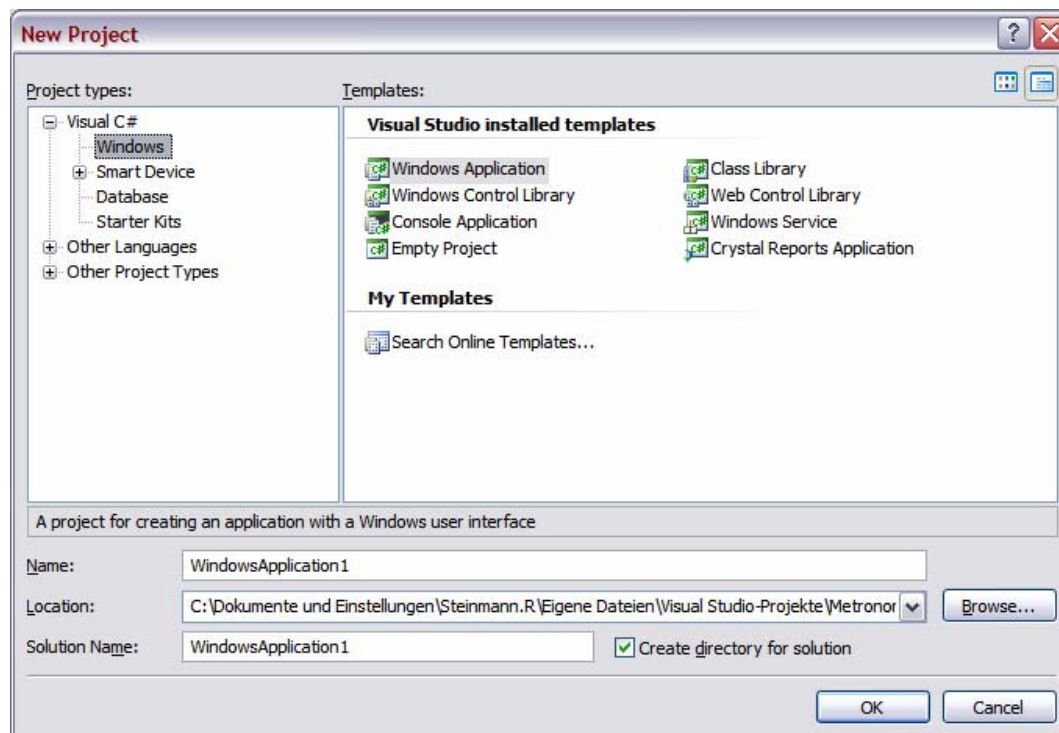
5.1 Schritt 1: Anlegen eines Projektes

Wie schon beim ersten C#-Programm, müssen Sie auch für dieses neue Windows-Programm ein Projekt anlegen.

Starten Sie die Microsoft Entwicklungsumgebung Visual Studio.NET, falls sie noch nicht schon läuft.



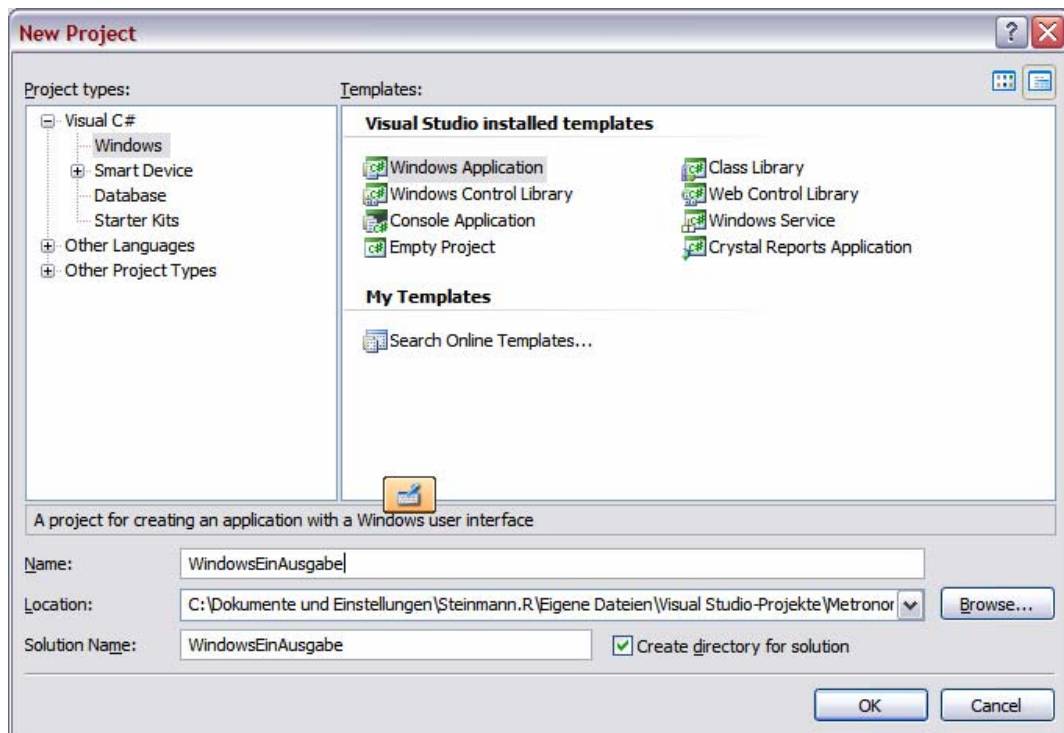
und rufen Sie das Menue **Datei - Neu - Projekt** auf (evtl. ist die Englische Version installiert, hier finden Sie den Aufruf unter **File - New - Project**):



Achten Sie darauf, dass Sie ein C#-Project erstellen. Sollten Sie voreingestellt eine andere Sprache sehen, finden Sie *Visual C#* unter dem Eintrag "*Other Languages*"

Wir wollen jetzt ein Programm schreiben, das unter Windows läuft und in einem Windows- Formular (= sog. Form) einen Text ausgibt und eine Texteingabe erlaubt:

- Klicken Sie in der Liste der Projekttypen auf Visual C#-Projekte
- Klicken Sie im Vorlagen-Feld auf das Symbol "Windows-Anwendung" bzw. "Windows Application"
- Geben Sie bei Name: einen Namen für Ihr Projekt ein, z.B. **WindowsEinAusgabe**. I.d.R. ist dieser Projektname identisch mit dem Namen Ihrer Anwendung und damit Ihres Namespaces. Zur Erinnerung: bitte verwenden Sie keine Zahlen als erstes Zeichen des Namens und keine Sonderzeichen. Eine Folge von Worten unterscheidet man üblicher Weise durch Groß- und Kleinbuchstaben.
- Wählen Sie unter Speicherort: Ihren persönlichen Ordern für Programmieren, den Sie im bereits vor Ihrem ersten C#-Programm angelegt haben. Klicken Sie dazu auf den Knopf "Durchsuchen..."



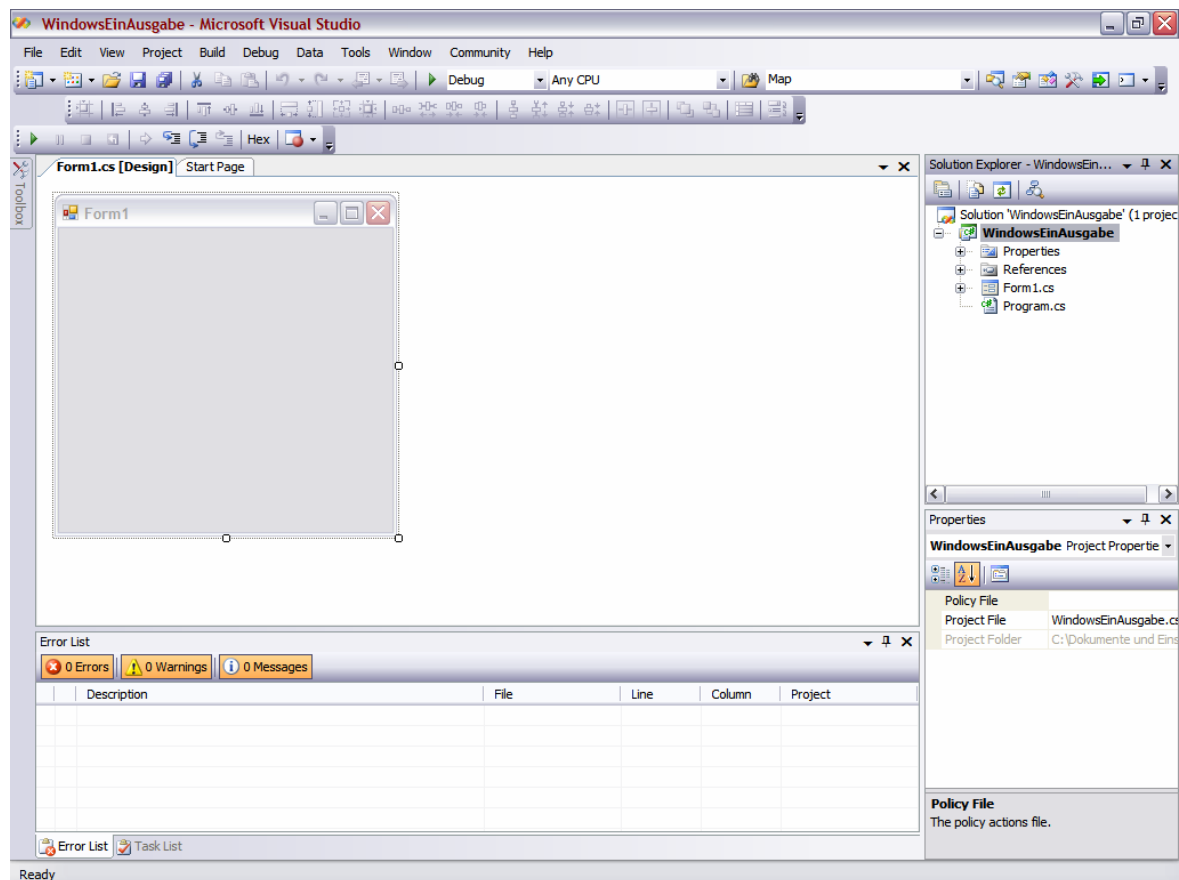
- Klicken Sie dann auf OK

Es öffnet sich jetzt ein neues Fenster, das im nächsten Schritt erklärt wird.

Außerdem wurde unter Ihrem Ordner für das Programmieren automatisch ein neuer Ordner mit dem Namen "WindowsEinAusgabe" angelegt (das können Sie mit dem File- Explorer überprüfen). Dieser neue Ordner enthält neben ihrer Quell-Code-Datei eine Reihe von weiteren Dateien und Unterordnern, die für die Integration in die CLR und die Projektverwaltung erforderlich sind.


5.2 Schritt 2: Arbeiten mit Forms und Steuerelementen

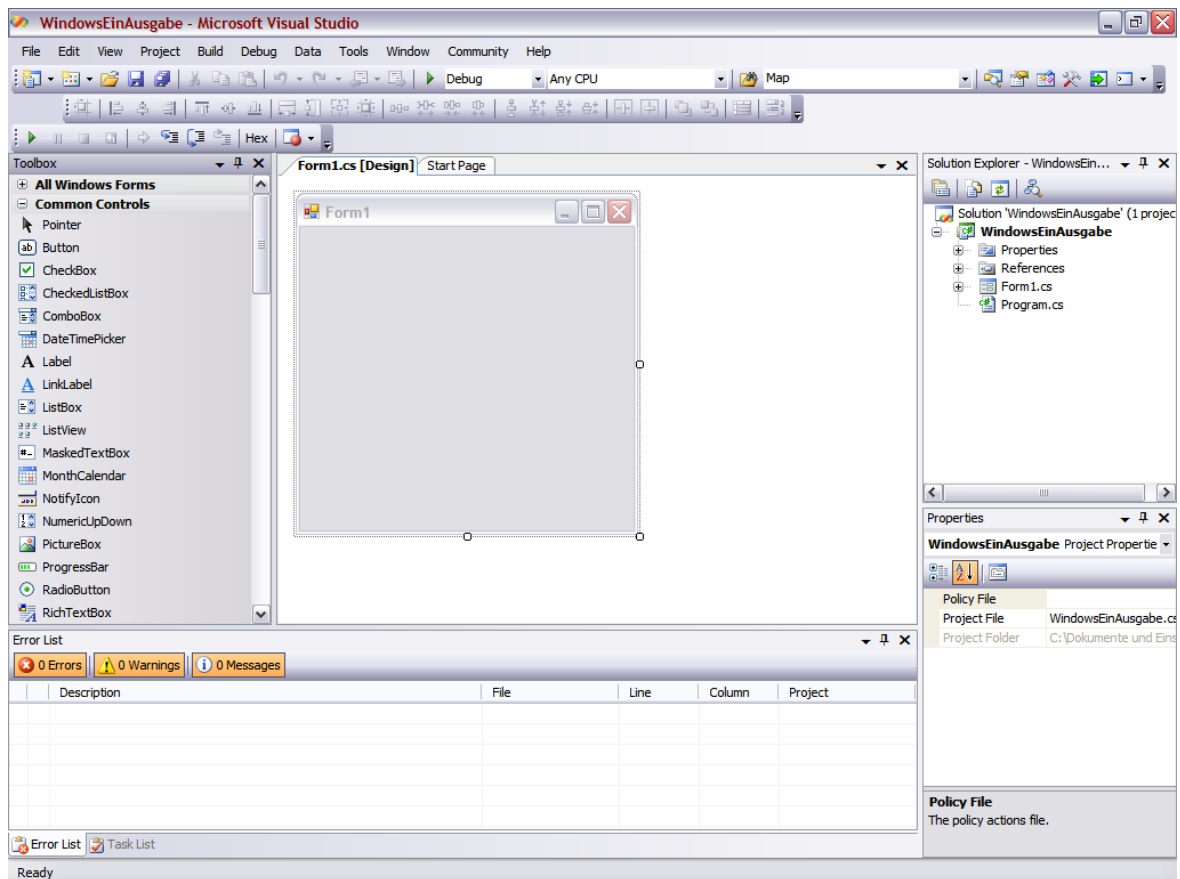
Visual Studio.NET öffnet jetzt die Entwurfsansicht für ein leeres Formular (= sog. Form):



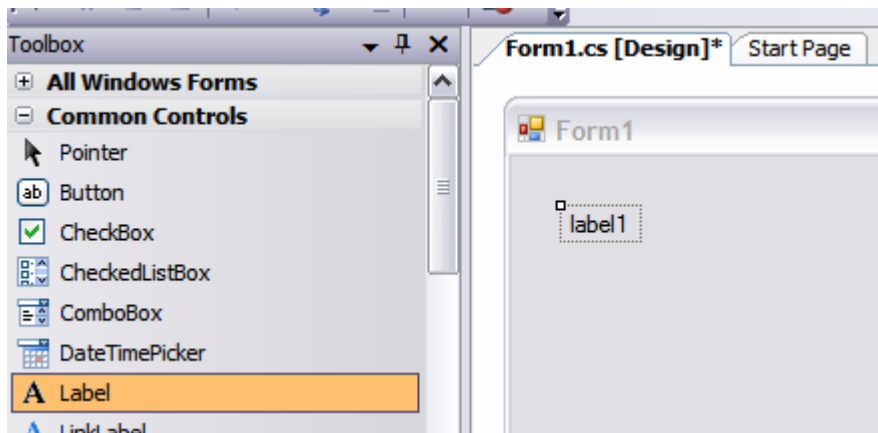
In den nächsten Schritten wird mit Hilfe des Visual Designers eine kleine Windows- Benutznchnittstelle entworfen und erzeugt. Dazu werden in das Formular drei Steuerelemente eingefügt.



- Klicken Sie in der Symbolleiste auf die Schaltfläche *Toolbox*
- Die Toolbox wird auf der linken Seite geöffnet
- Falls die Toolbox das Formular verdeckt, klicken Sie auf die Schaltfläche *Automatisch in den Hintergrund*  in der Titelzeile der Toolbox



- Klicken Sie in der Toolbox auf Label und klicken Sie dann in die linke obere Ecke des Formulars




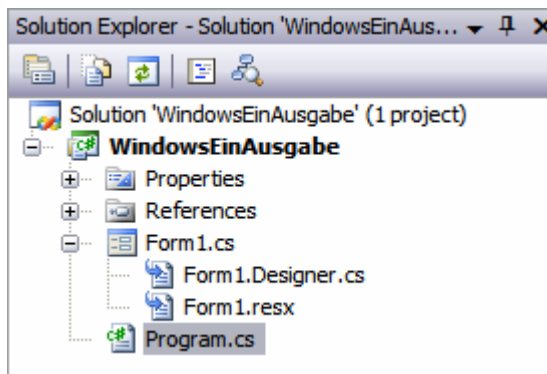
Dem Formular wird ein Label-Steuerelement hinzugefügt


- Klicken Sie nun in der Toolbox auf *TextBox* und klicken dann unterhalb des Label- Steuerelementes in das Formular.
Dem Formular wird eine Textbox-Steuerelement hinzugefügt
- Klicken Sie als nächstes in der Toolbox auf *Button* und dann im Formular rechts neben das Textfeld
Dem Formular wird ein Schaltflächen-Steuerelement hinzugefügt.

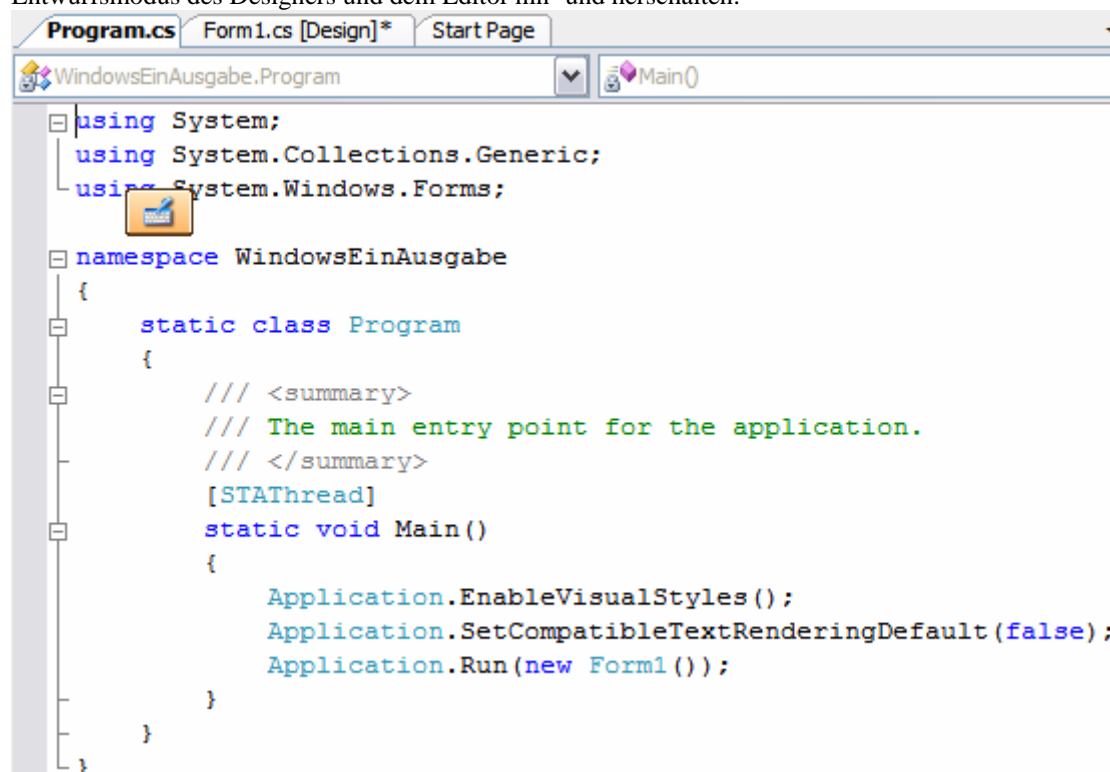
5.3 Schritt 3: Automatisch erzeugten Quell-Code für Forms verstehen

Während Sie die Benutzerschnittstelle grafisch und interaktiv zusammengestellt haben, haben Sie auch unbemerkt programmiert. Parallel hat Visual Studio.NET den C#-Programm-Code generiert, der erforderlich ist, um dieses Benutzer- Interface zu programmieren und später anwenden zu können.

- Sie benötigen jetzt den Projektmappen-Explorer. Falls er noch nicht angezeigt wird, öffnen Sie ihn auf der Symbolleiste mit der Schaltfläche *Solution-Explorer* .
- Der Solution-Explorer sollte bei Ihnen folgendes anzeigen:



- Der für Ihre Windowsanwendung grundsätzlich erforderliche und automatisch generierte Programmcode wurde in den Dateien *Program.cs* und *Form1.cs* abgelegt, die sie auch im Solution-Explorer sehen.
- Klicken Sie im Solution-Explorer auf *Program.cs* und dann auf die Schaltfläche *Code anzeigen* . Die Quelldatei *Program.cs* wird im Text- und Editor-Fenster angezeigt.
- Am oberen Rand des Editor-Fensters sehen Sie Registerkarten. Damit können Sie zwischen dem Entwurfsmodus des Designers und dem Editor hin- und herschalten.



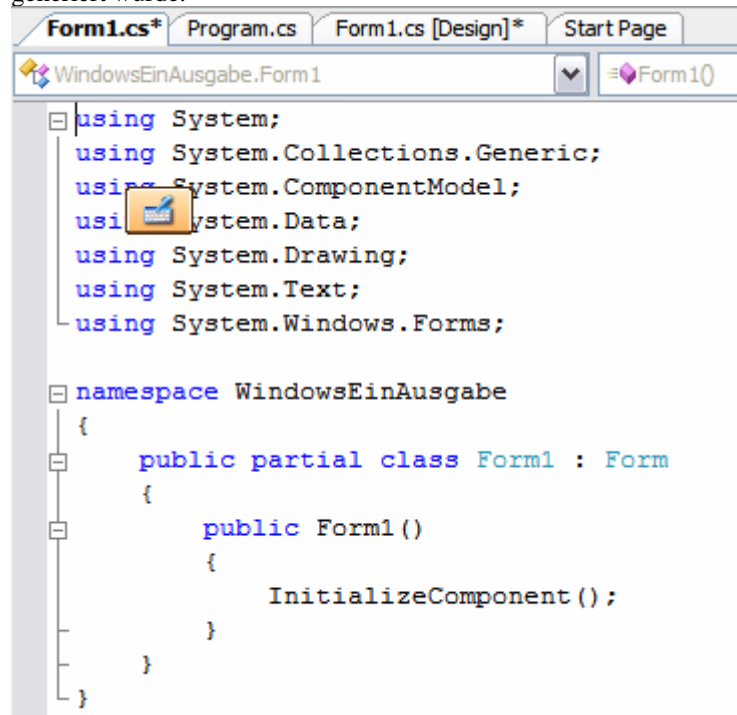
Wie jedes C#-Programm besteht auch Ihre Anwendung aus einem *namespace*, der den Namen trägt, den Sie für das Projekt gewählt haben. Als nächste Unterstruktur folgt die *class* die in diesem Fall automatisch mit *Program* benannt wurde. Klassen enthalten eine oder mehr Methoden, in denen die ausführbaren Programmanweisungen programmiert werden. Der Einstiegspunkt jeder Anwendung ist die Methode *Main*, d.h. in jeder Anwendung muss mindestens die Methode *Main* enthalten sein.

Die Klasse *Program* ist in diesem Fall sehr kurz: sie initialisiert einige grundsätzlichen Dinge, die jedes Windowsprogramm benötigt und startet mit dem Befehl `Application.Run(new Form1());` eine weitere Klasse mit dem Namen *Form1()*.

- Das Starten und Beenden einer Windows-Anwendung erledigt die aufgerufene Methode *Run()* in der Klasse *Application*. Diese Klasse ist Teil des Klassenbaumes *System.Windows.Forms* und wird vom Compiler

gefunden, weil dieser Namespace mit `using` ganz am Anfang angekündigt wurde. Wenn Sie im Quelltext den Maus-Cursor auf *Application* oder auf *Run* legen (ohne zu klicken!), werden entsprechende Erklärungen eingeblendet.

- Den zur Klasse *Form1()* gehörigen Programmcode können Sie sehen, wenn Sie im Solution-Explorer auf *Form1.cs* klicken und dann auf die Schaltfläche für den Quellcode.
- *Forms1.cs* enthält den gesamten C#-Code, der durch Ihren Entwurf der Benutzerschnittstelle automatisch generiert wurde.



- Ab der Version Microsoft Visual Studio 2005 werden sog. partielle Klassen unterstützt. Partielle Klassen wirken wie eine einzige Klasse, können aber auf verschiedene Quelldateien aufgeteilt werden. In unseren einfachen Beispielen, wird der Nutzen noch nicht ersichtlich und scheint eher zu verwirren. Bei Projekten aber, an denen viele Programmierer gleichzeitig arbeiten, ist diese Möglichkeit der zusätzliche Aufteilung sehr hilfreich.
Während in der Vorgängerversion von Visual Studio 2005 in der Datei *Form1.cs* bereits alle für eine kleine Windowsanwendung erforderlichen Kommandos enthalten waren, werden diese ab der Version 2005 auf partielle Klassen aufgeteilt. Der Quellcode von *Form1.cs* dient jetzt also nur noch dazu, die Klasse *Form1()* zu initialisieren.

- Schauen Sie sich den Quelltext an:
 - Am Anfang wurden eine Reihe **using-Direktiven** eingefügt. Dadurch kann Ihr Programm Klassen und Methoden verwenden, die in verschiedenen Bibliotheken mitgeliefert werden. Der erhebliche Programmieraufwand, um Windows-Programme zu starten, zu beenden und um Steuerelemente in einer grafischen Benutzerschnittstelle zu programmieren, ist damit schon vorweggenommen. Der Compiler wird beim Übersetzen aus den Bibliotheken den Code mit dazu laden, der für die Steuerelemente erforderlich ist.
- ```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

```

- Als oberste Organisations-Ebene wurde wieder der **Namespace** angelegt, dessen Name vom Projektnamen übernommen wurde.

```
namespace WindowsEinAusgabe
{
 ...
}
```

- Die nächste Organisationsebene ist die **Klasse**, der hier der Name *Form1* zugewiesen wurde. (Es hätte auch irgend ein anderer Name sein können.). Außerdem wird festgelegt, dass es sich hier um eine partielle Klasse handelt.

```
namespace WindowsEinAusgabe
{
 public partial class Form1 : Form
 {
 ...
 }
}
```

Klassen werden später ausführlich behandelt. Dann wird auch erklärt werden, wozu das Kommando *public* erforderlich ist. Nehmen Sie es momentan einfach hin.

- Die Klasse *Form1* enthält die Methode *Form1* und ist damit ein sog. **Konstruktor**:

```
...
public partial class Form1 : Form
{
 ...
 public Form1()
 {
 ...
 }
 ...
}
```

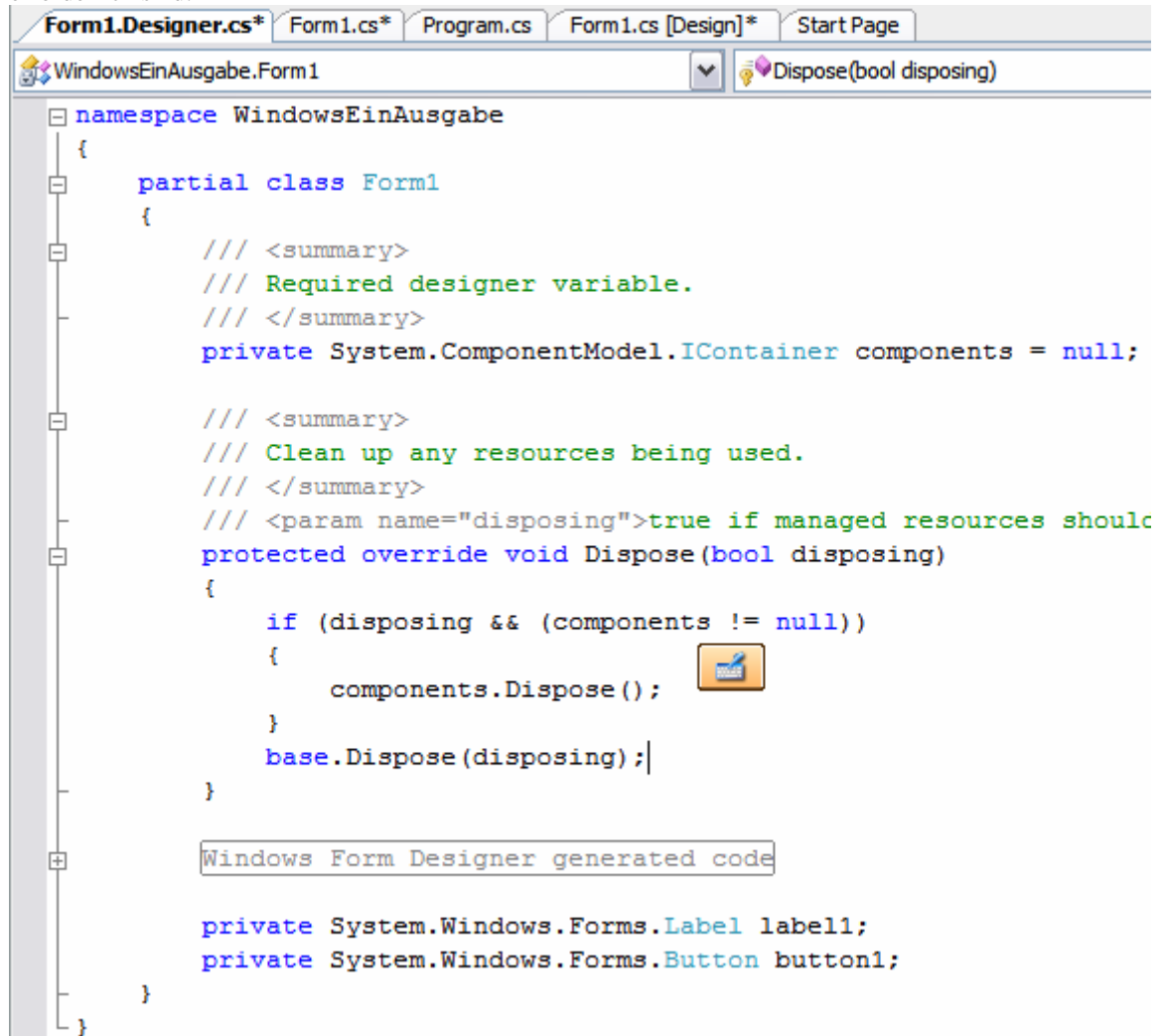
Ein Konstruktor ist eine spezielle Methode, die den gleichen Namen wie die Klasse trägt. Sie wird ausgeführt, wenn das Formular erzeugt werden soll und kann Code enthalten, der das Formular initialisiert. Man sagt in der Fachsprache: Der Konstruktor erzeugt eine Instanz einer Klasse im Speicher (genauer im Heap).

Bildlich kann man sich das wie das Kuchen-Backen vorstellen: Der Konstruktor ist der Bäcker, die Kuchenform entspricht der Klasse und die Instanz entspricht dem einzelnen gebackenen Kuchen. Die Initialisierung entspricht den speziellen Zugaben für einen bestimmten Kuchen, also etwa Rosinen oder Nüsse.

In der Methode *Form1* wird die Methode *InitializeComponent()* aufgerufen, die weiter unten im Programmcode programmiert ist und im Laufe der nächsten Punkte erklärt werden wird.

```
...
public Form1()
{
 ...
 InitializeComponent();
 ...
}
...
```

- Den weiteren Programmcode, der zu dieser Klasse Form1 gehört und für ihre Anwendung automatisch generiert wurde, finden sie in der Datei *Form1.Designer.cs*, deren Inhalte Sie sehen können, wenn sie im Solution-Explorer darauf klicken und den Quellcode öffnen.
- Dieser Programmcode gehört, wie alle bisher vorgestellten Komponenten, zum Namespace WindowsEinAusgabe und ist partieller Teil der Klasse Form1. Er initialisiert und enthält die Strukturen, die für das Windowsformular und die auf ihm abgesetzten Oberflächenelemente erforderlich sind.



```

namespace WindowsEinAusgabe
{
 partial class Form1
 {
 /// <summary>
 /// Required designer variable.
 /// </summary>
 private System.ComponentModel.IContainer components = null;

 /// <summary>
 /// Clean up any resources being used.
 /// </summary>
 /// <param name="disposing">true if managed resources should
 protected override void Dispose(bool disposing)
 {
 if (disposing && (components != null))
 {
 components.Dispose();
 }
 base.Dispose(disposing);
 }

 Windows Form Designer generated code

 private System.Windows.Forms.Label label1;
 private System.Windows.Forms.Button button1;
 }
}

```

- In C# wird der erforderliche Speicherplatz für verwendete Ressourcen automatisch wieder an das Betriebssystem freigegeben, sobald diese nicht mehr benötigt werden. Dies erledigt der sog. **Garbage Collector**. Allerdings weiß man nicht im voraus, wann der Garbage Collector diese Bereinigung durchführen wird. Diese Prozedur, die erhebliche Systemleistung beansprucht, kann also zu einem unerwünschten Zeitpunkt erfolgen und dem Anwender als unangenehmes "Hängen" des Programms vorkommen. Um wertvolle Ressourcen bald wieder freizugeben und um den Zeitpunkt der Freigabe selbst bestimmen zu können, kann man eine **Dispose-Methode** einbauen. Diese erledigt dann die Bereinigung und verhindert, dass der Garbage Collector zu einem späteren, evtl. ungünstigen Zeitpunkt noch einmal versucht, diese Ressourcen wieder freizugeben.

- In der Klasse wurden komplexe Variablen in Form von sog. Feldern erzeugt. **Felder** speichern Daten für eine Klasse. Hier sehen Sie, dass solche Variablen für die vorhin von Ihnen abgesetzten Oberflächenelemente automatisch angelegt wurden.

```
namespace WindowsEinAusgabe
{
 public class Form1 : System.Windows.Forms.Form
 {
 ...
 private System.Windows.Forms.Label label1;
 private System.Windows.Forms.Button button1;
 ...
 }
}
```

Diese Felder enthalten die Datenstrukturen für die Steuerelemente, die Sie dem Formular im Entwurfsmodus hinzugefügt haben. Die Felder tragen die Namen *label1* und *button1* und verweisen über obige Codierung auf entsprechende, bereits mitgelieferte Datenstrukturen.

Die Bedeutung von Feldern, die obige Codierung und die Erklärung, wozu das Kommando *private* dient, werden später genauer erklärt.

- Die nächste Methode ist evtl. im Editor-Fenster ausgeblendet. Dies sieht dann so aus:



Öffnen Sie diesen Code-Bereich, indem Sie auf das kleine "+" klicken.

Sie werden sehen, dass jetzt ein Code-Bereich geöffnet wird, der durch zwei **region**-Kommandos eingerahmt wird. Mit **Region** kann man einen Code-Bereich strukturieren, um ihn im Editor ein- und ausblenden zu können.

In diesem Fall wurde der in diesem Bereich enthaltene Code deshalb in eine Region strukturiert und ausgeblendet, weil man ihn im Editor nicht verändern sollte. Dieser Programmcode wird durch die interaktive Arbeit mit dem Designer im Entwurfsmodus des Formulars komplett generiert, und wird automatisch überschrieben, sobald man im Entwurfsmodus Änderungen vornimmt. Das Zusammenstellen eines Formulars im Entwurfsmodus ersetzt also die wesentlich aufwendigere händische Programmierung.

- Nachdem sie die Region geöffnet haben, können Sie die **Methode InitializeComponent** sehen.

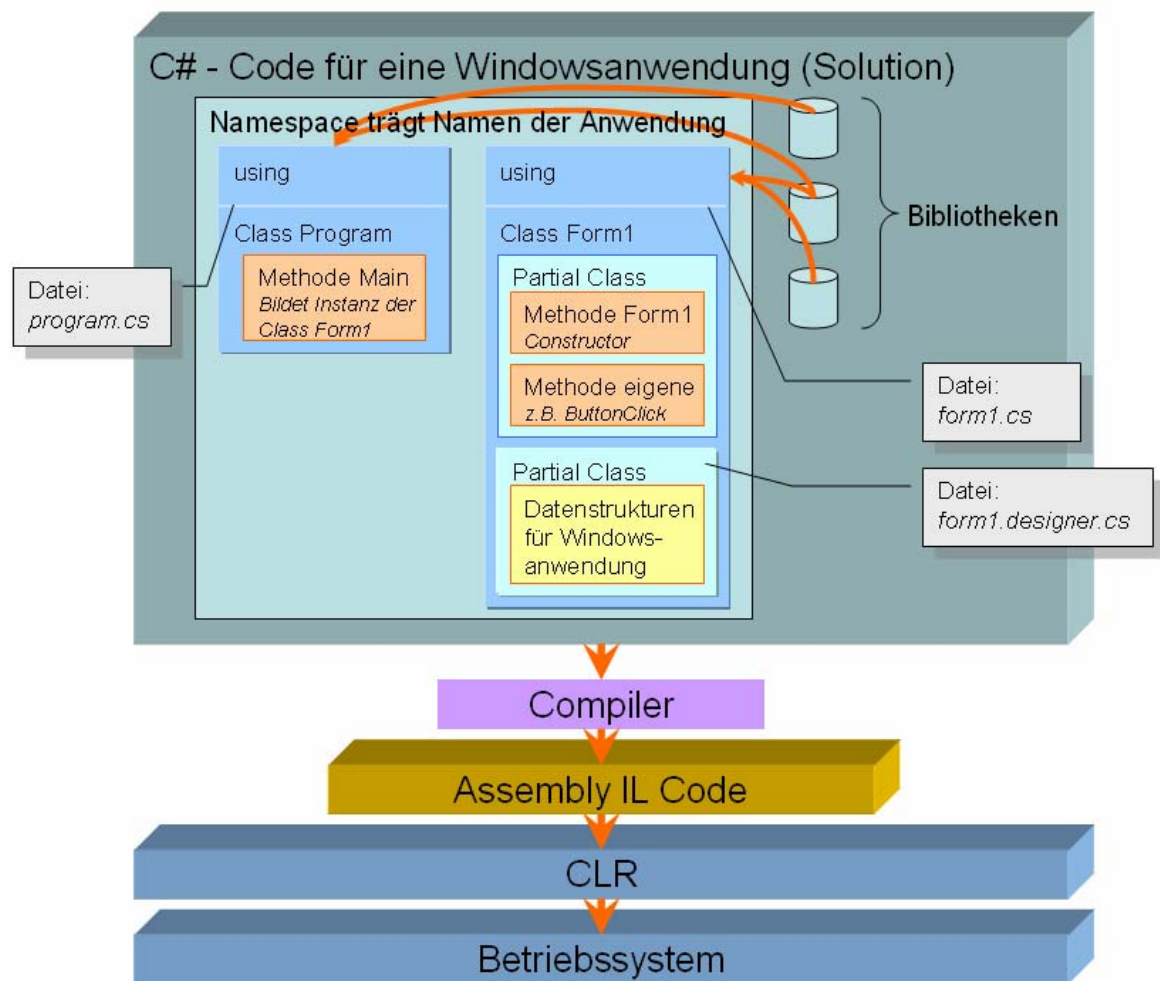
```
...
partial class Form1
{
 ...
 private void InitializeComponent()
 {
 this.Label1 = ...
 ...
 }
 ...
}
```

Die Anweisungen in dieser Methode legen die Eigenschaften der Steuerelemente fest und wurden automatisch auf Grund des Designs aus dem Entwurfsmodus erzeugt. Wie schon gesagt, sollten Sie diesen Programmcode nie direkt im Editor verändern, da er bei Änderungen im Design automatisch neu erzeugt und überschrieben wird und Ihre Änderungen dabei verloren gingen. Falls Sie Änderungen im Design vornehmen wollen, machen Sie das immer im Entwurfsmodus des Designers.


- Abschließende Anmerkung:

In der Regel werden Sie Ihren eigenen Programmcode in die sehr einfach aufgebaute Datei *Form1.cs* eingeben. Alle Änderungen und Ergänzungen am Aussehen der Oberfläche werden automatisch im Quellcode der Datei *Form1.Designer.cs* generiert. Die Veränderung dieser Datei sollte also Visual Studio vorenthalten bleiben. Bis auf seltene Ausnahmen (z.B. bei falscher Anwendung von Visual Studio) ist es nicht erforderlich, in der Datei *Form1.Designer.cs* irgendetwas von Hand etwas zu programmieren. Da der Programmcode als partielle Klasse auf die Dateien *Form1.cs* und *Form1.Designer.cs* aufgeteilt wurde und damit letztlich wie eine einzige Klasse *Form1()* wirkt, haben sie auch im Quellcode der Datei *Form1.cs* Zugriff auf alle Oberflächenstrukturen, die im Quellcode der Datei *Form1.Designer.cs* definiert wurden.

Die folgende Grafik soll die Architektur eines Windowsprogrammes verdeutlichen:



#### 5.4 Schritt 4: Eigenschaften von Steuerelementen verändern

- Klicken Sie oben auf die Registerkarte für das Entwurfswindow (Form1.cs[Design])
- Sie sind wieder im Entwurfsmodus des Designers. Klicken Sie auf die Schaltfläche *Eigenschaften* . Jetzt wird rechts unten das Eigenschaftsfenster angezeigt.
- Klicken Sie im Formular auf das von Ihnen platzierte Steuerelement für die Schaltfläche, auf der automatisch der Name "button1" geschrieben wurde.
- Das dazugehörige Eigenschaftsfenster wird darauf hin eingeblendet. Klicken Sie den Eintrag *Text* an
- Überschreiben Sie den Eintrag **button1** mit **OK** und drücken Sie dann die Enter-Taste
- Im Formular wird jetzt auch der Name button1 auf dem Steuerelement für die Schaltfläche durch OK ersetzt.
- Klicken Sie jetzt das Steuerelement für das Beschriftungsfeld an (label1), und klicken im dazugehörigen Eigenschaftsfenster wieder auf den Eintrag *Text*
- Überschreiben Sie hier **label1** mit **Geben Sie Ihren Namen ein** und drücken die Enter-Taste
- Falls der neue Text nicht ganz in den vorgesehenen Platz für das Beschriftungs-Steuerelement passen sollte, können Sie dessen Rahmen an den weißen Anfassern (=Handels) entsprechend anpassen.
- Erzeugen Sie auf dem Formular auch noch eine Textbox und wiederholen sie die letzten Schritte auch noch für das Steuerelement des Texteingabefeldes. Überschreiben Sie hier den Namen **textBox1** mit **hier** im Eigenschaftsfenster.

- Ihr Formular sollte jetzt in etwa so aussehen:

- Wechseln Sie zurück auf die Register-Karte für den Programmtext-Editor
  - Scrollen Sie nach unten bis zum Code für die Methode *InitializeComponent()*
- Die Zuweisungen der neuen Text-Eigenschaften tauchen nun auch auf der rechten Seite der Zuweisungen im Programmcode auf:

```
private void InitializeComponent()
{
 ...
 this .Label1.Text = "Geben Sie Ihren Namen ein";
 ...
 this .textBox1.Text = "hier";
 ...
 this .button1.Text = "OK";
 ...
}
```

### 5.5 Schritt 5: Größe des Formulars ändern

- Schalten Sie zurück in den Entwurfsmodus des Designers
- Passen Sie das Layout des Formulars mit Hilfe der weißen Anfasser (=Handels) so an, dass es ein gefälliges Aussehen bekommt.
- Ihr Formular könnte jetzt so aussehen:



## 5.6 Schritt 6: Code für OK schreiben

- Falls Sie sich nicht mehr im Designer befinden, schalten Sie ihn wieder ein.
- Bewegen Sie den Mauszeiger auf die Steuerfläche für die OK-Schaltfläche und klicken Sie diese Schaltfläche doppelt an.
- Es wird jetzt das Editor-Fenster für die Quelldatei von *Form1.cs* geöffnet und folgendes wurde automatisch dem Programm- Code hinzugefügt:
- Der Klasse *Form1* wurde automatisch die neue Methode *Button1\_Click* hinzugefügt. Hier werden wir gleich den Programmcode einfügen, der ausgeführt werden soll, wenn der Anwender später auf OK klickt.
- Außerdem enthält die Methode *InitializeComponent* in der Datei *Form1.Designer.cs* eine weitere Anweisung die bewirkt, dass nach Klicken auf die OK- Schaltfläche die Methode *Button1\_Click* aufgerufen wird.

```
private void InitializeComponent()
{
 ...
 this.button1.Click
+=new System.EventHandler(this.button1_Click);
 ...
}
```

Auf die tieferen Hintergründe dieser Anweisung wird an dieser Stelle noch nicht eingegangen.

- Scrollen Sie sich im Programmcode zur Methode *Button1\_Click*
- Innerhalb der Methode geben sie folgende Anweisung ein.  
`MessageBox.Show("GrüßGott " + textBox1.Text);`  
 Sie werden bemerken, dass Sie IntelliSense wieder unterstützt. Sobald Sie hinter dem Klassennamen *MessageBox* einen `.` eingeben, erscheint ein Auswahlfenster aller möglicher Methoden dieser Klasse. Wählen sie hier *Show* aus.  
 Auch nachdem Sie hinter den Objektnamen *textBox1* des Texteingabefeldes einen `.` eingeben, erscheint ein Auswahlfenster mit allen Eigenschafts-Attributen, die dieses Objekt kennt. Wählen Sie hier *Text* aus.  
 Ihr Programmcode sollte jetzt so aussehen:

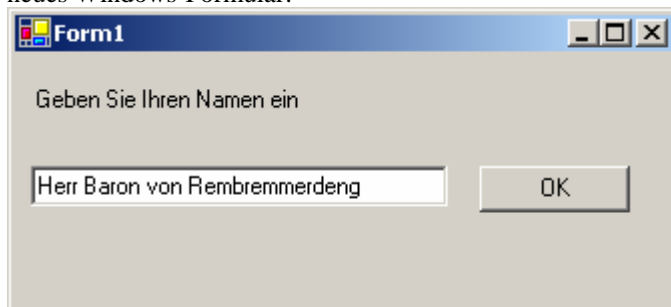
```
...
private void button1_Click(object sender, System.EventArgs e)
{
 MessageBox.Show("GrüßGott " + textBox1.Text);
}
...
```

- Die Methode *Show* erwartet als Parameter einen Text. Dieser wird mit Hilfe des `+`-Operators zusammengesetzt aus dem Text "Grüß Gott " und dem Text, den das Texteingabefeld *textBox1* über seine Eigenschaft *Text* liefert.
- **Achtung:**  
 Wie Sie gerade gelernt haben, lösen Sie durch Doppelklick auf ein Oberflächenelement eine Funktion von Visual Studio aus, mit der Programmcode generiert wird, der ausgeführt wird, wenn später ein Anwender auf das Oberflächenelement klickt. Theoretisch können Sie diese Funktion für jedes Oberflächenelement auslösen, was aber zu einem fehlerhaften Programmverhalten führen würde. Sollten Sie versehentlich einen Doppelklick auf ein Oberflächenelement ausgeführt haben, für das kein Programmcode ausgeführt werden soll (z.B. für einen Label), müssen Sie die dafür generierte Methode aus der Datei *Form1.cs* und die dafür angelegten Strukturen aus der Datei *Form1.Designer.cs* von Hand löschen. Das kann sehr mühsam sein und gerade einen Anfänger noch überfordern. Zügeln Sie also Ihr Temperament, wenn Sie mit den Oberflächenelementen im Designer-Fenster arbeiten! Achten Sie darauf, wann Sie ein Element einmal und wann doppelt anklicken wollen! Vermeiden Sie wildes, ungeduldiges und wahlloses rumklicken, falls der Rechner an dem Sie arbeiten, etwas langsam ist!



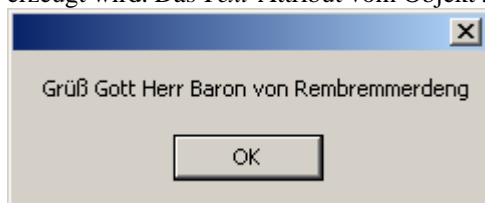
## 5.7 Schritt 7: Programm übersetzten und starten

- Wählen Sie das Menü **Debuggen - Starten ohne Debuggen**
- Der Compiler übersetzt nun Ihren Programmcode und das Programm wird gestartet. Es erscheint Ihr neues Windows-Formular:



Geben Sie Ihren Namen ein und klicken auf OK.

Es erscheint jetzt ein neues Dialogfenster, das Sie mit Ihrem Namen begrüßt und das von der Methode `MessageBox.Show("Grüß Gott "+textBox1.Text)` erzeugt wird. Das `Text`-Attribut vom Objekt `textBox1` liefert Ihren Namen.



## 5.8 Gratulation

Herzlichen Glückwunsch, Sie habe Ihr erstes Windows-Programm geschrieben mit dem man Text ein- und ausgeben kann.

Dabei haben Sie folgendes gelernt:

- Es wurden zwar wesentlich mehr Programmzeilen erzeugt, als beim ersten Beispiel, was vielleicht zunächst verwirrend war, der prinzipielle Aufbau ist jedoch so wie schon beim ersten Programm. Auch dieses Windows-Programm gliedert sich in
  - Namespace
  - Klasse
    - Felder
    - mehrere Methoden
- Neu hinzugekommen ist, dass eine Klasse neben Methoden auch Felder für Daten enthalten kann
- Auch neu war, dass man Bereiche des Codes mit einer Region einfassen kann, um sie im Editor zur besseren Übersicht ein- und aus-blenden zu können.
- Die Benutzerschnittstelle für ein Windows-Programm kann man grafisch und interaktiv zusammenstellen, der dazugehörige Programmcode wird automatisch generiert. Man kann also eine Windows-Benutzerschnittstelle durch Zeichnen programmieren, ohne selbst Programmcode schreiben zu müssen.
- Zwar ist das Programmieren unter Windows im Vergleich zu früher aufwändiger und komplizierter geworden, eine moderne Programmierumgebung kann aber einen Großteil des Aufwandes abnehmen.
- Der größte Teil dieses Programmes wurde automatisch erzeugt, selbst musste man in diesem Beispiel nur eine einzige Programmzeile eingeben.

## 6 C#-Bausteine

### 6.1 C#-Syntax

- C# ist Case-sensitiv, d.h. es wird zwischen Groß- und Kleinschreibung unterschieden. Dies gilt auch für Bezeichner von Klassen, Methoden und Variablen. So beschreiben etwa die Bezeichner *LängeDesBalkens* und *Längedesbalkens* zwei unterschiedliche Dinge
- Ausführbare Anweisungen befinden sich überwiegend in Methoden. Ausführbare Anweisungen müssen mit ; abgeschlossen werden.  
z.B.:  

```
Console.WriteLine("Hello World!");
```
- Bezeichner dürfen nicht mit Zahlen und Sonderzeichen beginnen
- Bezeichner dürfen keine Leerzeichen enthalten
- Setzt sich ein Bezeichner aus mehreren Worten zusammen, so werden diese mit Groß- und Kleinbuchstaben von einander getrennt
- Keine Sonderzeichen in Bezeichner-Namen
- Wie schon früher erläutert, gibt es neben den ausführbaren Anweisungen Kopfzeilen von Strukturen wie Klassen und Methoden. Der den Kopfzeilen folgende Rumpf wird in {} eingefaßt. Nach Kopfzeilen wird deshalb **kein** ; gesetzt.
- C# ist eine sog. formatfreie Sprache, d.h. Leerzeichen und Zeilenumbrüche können fast beliebig und sehr frei verwendet werden. Es ist allerdings sehr ratsam, Zeilenumbrüche und Tabulatoren so zu verwenden, dass der Programmcode optisch gut strukturiert und damit begreifbar wird.

### 6.2 Reservierte Schlüsselwörter

- Reservierte Schlüsselwörter werden in der .NET-Entwicklungsumgebung blau dargestellt, um eine irrtümlich falsche Verwendung zu vermeiden.
- Verwenden Sie keine Bezeichner für Klassennamen, die mit gängigen .NET Framework-Namespaces identisch sind. Verwenden Sie beispielsweise keinen der folgenden Namen als Klassennamen: **System**, **Collections**, **Forms** oder **UI**
- In C# sind Schlüsselwörter für bestimmte Zwecke reserviert, die nicht für Bezeichner von Klassen, Methoden oder Variablen verwendet werden dürfen. Beispiele für diese Schlüsselwörter sind: **using**, **public**, **private**, **byte**, **char**, **string**, **int**, **long**, **float**, **double**, **if**, **for**, **do**, **while**, **return**, **switch**, usw. Eine ausführliche Liste der reservierten Schlüsselwörter finden Sie in der Hilfe der .NET-Entwicklungsumgebung

### 6.3 Variablen

Eine Variable ist ein Platzhalter für eine Adresse, die auf einen Speicherplatz bestimmter Größe verweist, der einen Wert aufnehmen kann. Man kann sich Variablen wie Namen für verschieden große Schubladen in einem großen Schrank vorstellen. In einem Programm muss jede Variable einen eindeutigen Namen tragen. Der Variablenname wird verwendet, um auf den Wert zuzugreifen, der in der Variablen gespeichert ist.

Regeln zur Benennung von Variablen:

- Keine Unterstriche
- Verwenden Sie für verschiedene Variablen keine Namen, die sich nur durch Groß- und Kleinschreibung unterscheiden, etwa *meineVariable* und *MeineVariable*.
- Man sollte Namen von Variablen mit einem Kleinbuchstaben beginnen lassen, z.B. *laengeTraeger*
- Setzt sich der Bezeichner aus mehreren Worten zusammen, setzen Sie die Worte aneinander und beginnen das zweite und die folgenden Worte mit einem Großbuchstaben. Das ist die sog. "Kamel-Schreibweise", wenn man damit den Anblick einer Kamelherde assoziiert, bei der man Höcker an Höcker sieht.

### 6.3.1 Variablen deklarieren

Variablen müssen vor ihrer Verwendung mit einem Datentyp deklariert werden. Dadurch wird ihr Typ bestimmt und die erforderliche Größe des Speicherplatzes festgelegt, den das Betriebssystem zur Verfügung stellen muss. Die Deklaration geschieht unter Verwendung bestimmter Schlüsselwörter in einer sog.

Deklarationsanweisung, z.B.:

```
int anzahl;
double entfernung;
```

Nach der Deklaration muss den Variablen ein Wert zugewiesen werden, andernfalls verweigert der Compiler die Übersetzung und gibt eine entsprechende Fehlermeldung aus. Das unterscheidet C# von anderen Programmiersprachen, die das nicht zwingend vorschreiben. Durch diese Regel wird allerdings der verbreitete Programmierfehler vermieden, dass Variablen unbeabsichtigte und unsinnige Werte enthalten.

z.B.:

```
anzahl = 10;
entfernung = 20.37531;
```

Das "="-Zeichen ist der sog. Zuweisungsoperator, der einer Variablen einen Wert von rechts nach links zuweist.

Deklaration und Wertzuweisung kann auch in einer Anweisung erfolgen, z.B.:

```
int anzahl = 10;
```

Nach der Zuweisung kann die Variable im Programmcode verwendet werden, z.B.:

```
Console.WriteLine(anzahl);
Console.WriteLine("Anzahl =" + anzahl);
Console.WriteLine("Entfernung =" + entfernung);
Console.WriteLine("Anzahl = " + anzahl + " Entfernung = " + entfernung);
```

Folgender Programmcode würde eine Compiler-Fehlermeldung erzeugen, da der Variablen *anzahl* vor ihrer Verwendung in der *WriteLine*-Methode noch kein Wert zugewiesen wurde:

```
int anzahl;
Console.WriteLine(anzahl);
```

Anmerkung:

Zur Funktion des + Operators siehe Kapitel "[Arithmetische Operatoren](#)"

### Gültigkeit von Variablen

Variablen haben grundsätzlich nur innerhalb der Struktur Gültigkeit, in der sie deklariert werden. Dies gilt z.B. für Methoden, d.h. eine Variable hat nur innerhalb der Methode Gültigkeit, in der sie deklariert wurde.

Gleiches gilt sinngemäß für Felder (siehe etwas weiter unten).

Eine Ausnahme ist die Verwendung von Laufvariablen in Schleifen. Darauf wird zu gegebener Zeit hingewiesen werden.

Davon abweichend kann man explizit dafür sorgen, dass sie auch außerhalb der Methode Gültigkeit haben - das wird später erklärt.

### 6.3.2 Einfache Datentypen

C# kennt, wie andere Programmiersprachen auch, eine Reihe von eingebauten einfachen Datentypen. Hier eine Übersicht einiger wichtiger Datentypen, eine komplette Liste enthält die Hilfe der .NET-Entwicklungsumgebung:

| Datentyp             | Beschreibung                            | Größe (Bit)   | Wertebereich                            | Beispiel                                              |
|----------------------|-----------------------------------------|---------------|-----------------------------------------|-------------------------------------------------------|
| <code>int</code>     | Integer<br>Ganzzahl                     | 32            | -2 <sup>31</sup> bis 2 <sup>31</sup> -1 | <code>int anzahl;<br/>anzahl = 12;</code>             |
| <code>long</code>    | Ganzzahl<br>größerer Wertebereich       | 64            | -2 <sup>63</sup> bis 2 <sup>63</sup> -1 | <code>long anzahl;<br/>anzahl = 12L;</code>           |
| <code>float</code>   | Gleitkommazahl<br>7-Stellen Genauigkeit | 32            | ± 3,4 x 10 <sup>308</sup>               | <code>float laenge;<br/>laenge = 1.8F;</code>         |
| <code>double</code>  | Gleitkommazahl<br>doppelte Genauigkeit  | 64            | ± 1,7 x 10 <sup>308</sup>               | <code>double entfernung;<br/>entfernung = 3.2;</code> |
| <code>decimal</code> | Währung                                 | 128           | 28 Stellen Genauigkeit                  | <code>dezimal betrag;<br/>betrag = 5.78M;</code>      |
| <code>char</code>    | einzelnes Zeichen<br>Unicode            | 16            | 0 bis 2 <sup>16</sup> -1                | <code>char buchstabe;<br/>buchstabe = 'a';</code>     |
| <code>string</code>  | Zeichenkette                            | 16 je Zeichen |                                         | <code>string satz;<br/>satz = "Hello World";</code>   |
| <code>bool</code>    | Boolscher wert                          | 8             | true oder false                         | <code>bool flag;<br/>flag = false;</code>             |

2<sup>31</sup> bedeutet 2 hoch 31, entspricht 32.768

### 6.3.3 Komplexe Datentypen

Neben den einfachen Datentypen, ist es auch möglich, Variablen auf Grund von komplexeren Strukturen zu deklarieren. Wenn Sie sich noch einmal den Quellcode des ersten Windows- Programmes ansehen, finden Sie gleich nach dem Klassenkopf, noch bevor eine Methode programmiert wurde, folgende Deklarationen:

```
private System.Windows.Forms.Label label1;
private System.Windows.Forms.TextBox textBox1;
private System.Windows.Forms.Button button1;
```

Die Variable *label1* wird hier auf Grund der komplexeren Struktur *Label* deklariert, die für sich einen komplexen Datentyp darstellt, und die mit der Bibliothek *System.Windows.Forms* mitgeliefert wird. Diese Deklaration wurde automatisch aus der grafischen Gestaltung des Formulars generiert. Entsprechendes trifft für die Variablen *textBox1* und *button1* zu.

Wenn Sie weiter unten in dem Programmcode nachsehen, der in der *region* eingefaßt ist, finden Sie folgende Anweisung:

```
this.label1.Text = "Geben Sie Ihren Namen ein";
```

Während das Schlüsselwort *this* erst später erklärt werden wird, können Sie an dieser Anweisung erkennen, dass sich hinter der Variablen *label1* durch die Deklaration mit einem komplexen Datentyp offensichtlich mehr verbirgt als nur ein einfacher Datentyp. *label1* hat eine Reihe von Eigenschaften erhalten, z.B. die Eigenschaft *Text*. Über *label1.Text* kann man diesem Attribut, das den Typ *string* hat (den Typ muss man wissen oder in der Definition nachlesen), mit dem "="-Operator einen Wert zuweisen.

Gleiches gilt für die Variablen *textBox1* und *button1*.

Noch weiter unten können Sie in der Methode, die durch den Doppel-Klick im Formular-Editor erzeugt wurde, folgendes finden:

```
private void button1_Click(object sender, System.EventArgs e)
{
 MessageBox.Show("Grüß Gott " + textBox1.Text);
}
```

Hier wird eine Methode *Show* der Klasse *MessageBox* aufgerufen, der als Parameter ein String übergeben wird, der sich aus den Teilen "Grüß Gott " und dem Text zusammensetzt, der im Attribut *Text* der Variablen *textBox1* abgespeichert wurde. Ursprünglich stand in diesem Attribut ja der String "*hier*" der dann von dem String des Namens überschrieben wird, den der Anwender eingeben kann.

### 6.3.4 Felder

In dem eben genannten Code-Beispiel:

```
private System.Windows.Forms.Label label1;
private System.Windows.Forms.TextBox textBox1;
private System.Windows.Forms.Button button1;
```

werden die Variablen nicht innerhalb einer Methode, sonder unmittelbar am Beginn der Definition einer Klasse und damit außerhalb aller darin enthaltenen Methoden deklariert. Wie schon früher erwähnt, bezeichnet man Variablen, die an dieser besonderen Stelle deklariert werden, als **Felder**.

Zur Erinnerung: Felder haben in der Klasse Gültigkeit, in der sie deklariert wurden. Damit können ihre Werte in allen Methoden der Klasse gelesen oder beschrieben, und damit geändert werden.

Dies kann man sehr schön an der Variablen *textBox1* beobachten. In der Methode *InitializeComponent()* wird dem Attribut *textBox1.Text* zunächst der Wert *hier* zugewiesen, der dann durch die Eingabe des Namens eines Anwenders überschrieben wird. Später kann die Methode *button1\_Click(...)* den Wert der Variablen mit *textBox1.Text* auslesen.

Anmerkung:

Die Bezeichnung Felder kann bei Programmierern, die andere Sprachen kennen, zu einem Missverständnis führen, denn dort wird dieser Begriff manchmal für mehrdimensionale Variablen verwendet. Die Verwirrung entsteht, weil das englische Wort für mehrdimensionale Variablen, *array*, im deutschen ebenfalls oft mit *Feld* übersetzt wird.

(Es wäre also doch überlegenswert, zumindest in der IT nicht alle englischen Fachbegriffe zwanghaft einzudeutschen)

### 6.3.5 Geltungsbereich von Variablen

Variablen sind nur in der Methode gültig, in der sie deklariert wurden. Innerhalb einer Methode müssen Variablen eindeutige Namen haben. Nach dem eine Methode abgearbeitet wurde, verlieren die darin enthaltenen Variablen ihre Gültigkeit, d.h. der von ihnen belegte Speicherplatz kann vom Garbage-Collector wieder an das Betriebssystem zurückgegeben werden.

#### Ausnahme Felder:

Werden Variablen gleich zu Beginn des Klassen-Rumpfes und damit außerhalb der in der Klasse befindlichen Methoden deklariert, werden sie Felder genannt und sind für die gesamte Klasse und damit auch innerhalb aller Methoden dieser Klasse gültig. Felder verlieren erst ihre Gültigkeit, wenn eine Klasse nicht mehr benötigt wird.

z.B.:

```
class ...
{
 int feld1 = 1;

 methode1 ()
 {
 int a = 2;
 feld1 = 2;
 // a ist nur in methode 1 gültig
 // feld1 ist auch in methode1 gültig
 ...
 }

 methode2 ()
 {
 int b = 2;
 feld1 = 3;
 // b ist nur in methode2 gültig
 // feld1 ist auch in methode2 gültig
 ...
 }
}
```

### 6.3.6 Typ-Umwandlungen

Mitunter ist es notwendig den Typ einer Variablen in einen anderen Typ umzuwandeln, um diese Variable dann konsistent mit anderen Variablen weiterbehandeln zu können, die den anderen Typ haben.

#### Text -> Zahl:

Soll z.B. eine Zahl, die der Anwender in einem Text-Formular eingibt, und die dadurch erst einmal den Typ String hat, weiter als Zahl verarbeitet werden, muß die entsprechende Variable in einen Zahlentyp umgewandelt werden. Folgendes Beispiel zeigt, wie dies mit der Methode **Parse** der Klasse **int** erfolgen kann:

```
int zahl1 = int.Parse(textBox1.Text);
```

Der Wert vom Typ *string*, der im Attribut *Text* der komplexen Variablen *textBox1* abgespeichert ist, wird durch diese Anweisung in einen neuen Wert vom Typ *int* umgewandelt, der in der Variablen *zahl* abgespeichert wird.

#### Zahl -> Text:

Umgekehrt, kann man auch aus einer Variablen mit einem Zahlen-Typ eine Umwandlung in einen String-Typ erreichen. Dazu gibt es die Methode **ToString()**, die in allen Objekten der Basistypen enthalten ist. Im Code könnte das so aussehen:

```
int zahl2 = 30;
textBox1.Text = zahl2.ToString();
```

**Zahl Typ1 -> Zahl Typ2:**

Es können auch Zahlen eines Typs in Zahlen eines anderen umgewandelt werden, z.B. eine Zahl vom Typ **int** in eine vom Typ **long**. Typumwandlungen von einer Zahl die weniger Bits zur Darstellung benötigt in eine Zahl, die mehr Bits benötigt, geschieht verlustfrei, also z.B. **int** -> **long**. Bei einer Umwandlung in die entgegengesetzte Richtung, also z.B. **long** -> **int**, können allerdings Verluste auftreten, wenn der Wertebereich der Zahl, die mit weniger Bits dargestellt wird, überschritten wird.

```
short x = 5;
int y = x; // implizite Konvertierung

int i = 2;
double z = i; // ebenfalls eine implizite Konvertierung

short x = 5;
int y = 500;
x = y; // wird nicht kompiliert

short x = 5;
int y = 500;
x = (short) y; // das ist eine explizite Konvertierung und in Ordnung
```

**6.4 Arithmetische Operatoren**

Folgende arithmetischen Operatoren werden von C# unterstützt:

**Addition/Subtraktion:** +, -

**Multiplikation/Division:** \*, /

**Modulo:** %

Modulo liefert den Rest der Division zweier Zahlen und funktioniert nach der Regel:

$a - ((a / b) * b)$

Wird Modulo auf ganze Zahlen angewendet, so liefert es als Ergebnis ebenfalls eine ganze Zahl. Dies führt zu folgendem überraschenden Ergebnis:

54 % 13 liefert das Ergebnis 2

Erklärung:

54 / 13 ergibt 4,153846 Periode, wird aber als ganze Zahl behandelt, d.h. es wird mit 4 weitergerechnet.

4 \* 13 ergibt 52

54 - 52 ergibt 2

**Zuweisung:** =

**Sonderrolle des + Operators bei Strings:**

Eigentlich dürfen die arithmetischen Operatoren nur auf Variablen mit Zahlen-Typen angewendet werden.

Eine Sonderfunktion hat der + Operator bei Strings, denn darüber können, wie früher schon gezeigt, einzelne Zeichenketten zusammengehängt werden.

z.B.:

```
int anzahl = 10;
double entfernung = 20.37531;
Console.WriteLine("Anzahl = " + anzahl);
Console.WriteLine("Entfernung = " + entfernung);
Console.WriteLine("Anzahl = " + anzahl + " Entfernung = " + entfernung);
```

**6.4.1 Vorrang**

Sämtliche Operatoren in C# sind in **Vorrangstufen** eingeteilt. Dabei haben die Operatoren \*, /, % eine höhere Priorität als + und - .

Die Rechnung

$2 + 3 * 4$

wird also so berechnet:

$2 + (3 * 4)$

[Siehe auch](#)

### 6.4.2 Orientierung/Assoziativität

Neben den Vorrangsstufen besitzen die Operatoren eine **Orientierung** (bzw. **Assoziativität**) und damit eine Richtung in der sie ausgewertet werden.

Der "="-Operator ist rechtsassoziativ und hat damit die Orientierung rechts nach links. In folgender Anweisung wird der Variablen *zahl* der Wert 30 zugeordnet:

```
int zahl = 30;
```

Die arithmetischen Operatoren +, -, \*, /, % sind linksassoziativ und haben damit die Orientierung links nach rechts. Folgende Rechnung:

```
2 / 3 * 4
```

wird also wie folgt ausgewertet:

```
(2 / 3) * 4
```

[Siehe auch](#)

### 6.4.3 Verbundzuweisungs-Operatoren

Neben dem Zuweisungsoperator "=" und den arithmetischen Operatoren gibt es außerdem noch eine Mischform, bei der Zuweisung und arithmetische Operation in einer sehr kompakten Schreibweise zusammengefasst werden. Das sind sog. Verbundzuweisungs-Operatoren, die von Profis aus Gründen der Arbeitersparnis immer anstelle der ausführlichen Schreibweise verwendet werden.

Anstelle der Anweisung:

```
a = a + 4;
```

kann man auch verkürzt schreiben:

```
a += 4;
```

Diese Regel gilt auch für alle anderen arithmetischen Operatoren:

| Ausführlich                              | Verkürzt                       |
|------------------------------------------|--------------------------------|
| <code>variable = variable * zahl;</code> | <code>variable *= zahl;</code> |
| <code>variable = variable / zahl;</code> | <code>variable /= zahl;</code> |
| <code>variable = variable % zahl;</code> | <code>variable %= zahl;</code> |
| <code>variable = variable + zahl;</code> | <code>variable += zahl;</code> |
| <code>variable = variable - zahl;</code> | <code>variable -= zahl;</code> |

Vorrang und Assoziativität von \*=, /=, %=, +=, -= entsprechen dem =, d.h. auch sie sind rechtsassoziativ ([siehe](#)).

Wie schon der +-Operator, kann auch += auf Zeichenketten angewandt werden:

```
string name = "Sepp";
string gruss = "Servus ";
gruss += name; // anstelle von gruss = gruss + name;
Console.WriteLine(gruss); // auf dem Bildschirm erscheint: Servus Sepp
```



## 6.4.4 Inkrement-/Dekrement-Operatoren

Da das In- bzw. Dekrementieren von Variablen um den Wert 1 beim Programmieren sehr häufig vorkommt, wurde auch für diesen Fall eine verkürzte Schreibweise eingeführt, die von Profis immer verwendet wird.

Anstelle von:

```
zaehler = zaehler + 1;
```

bzw.:

```
zaehler += 1;
```

kann man verkürzt schreiben:

```
zaehler++;
```

Das selbe gilt für das dekrementieren um 1:

| Ausführlich              | Verkürzt    |
|--------------------------|-------------|
| variable = variable + 1; | variable++; |
| variable += 1;           | variable++; |
| variable = variable - 1; | variable--; |
| variable -= 1;           | variable--; |

## Präfix- Postfix-Schreibweise

Im Zusammenhang mit Inkrement-/Dekrement-Operatoren muss noch darauf hingewiesen werden, dass es hier zwei Möglichkeiten gibt, die sich unterschiedlich auswirken:

- `zaehler++;` das ist eine **Postfix**inkrementation
- `++zaehler;` das ist eine **Präfix**inkrementation

Dito für `zaehler--;` und `--zaehler;`

Beide Schreibweisen erhöhen die Variable "zaehler" um 1 (bzw. vermindern sie um 1).

Allerdings ist der Wert von ...

- ... `zaehler++` der Wert der Variablen **vor** der Inkrementierung
- ... `++zaehler` der Wert der Variablen **nach** der Inkrementierung

Deshalb liefert folgendes Programm ein zunächst vielleicht überraschendes Ergebnis:

```
int x;
x = 50;
Console.WriteLine(x++);
// x hat den Wert 51, aber x++ den Wert 50,
// d.h. 50 wird ausgegeben
x = 50;
Console.WriteLine(++x);
// x hat den Wert 51 und ++x ebenfalls,
// d.h. 51 wird ausgegeben
```

Dieses Verhalten wird meist in **while**- und **do**-Schleifen verwendet ([siehe](#)).

## **6.5 Übung 1**

### **6.5.1 Variablen auf Konsole anzeigen**

Erstellen Sie ein C#-Programm, das auf Konsole läuft und in dem je eine int, long, float, double, char, string Variable deklariert und mit einem Wert belegt wird.

Anschließend soll das Programm diese Werte auf dem Bildschirm anzeigen.

### **6.5.2 Variablen unter Windows anzeigen**

Erstellen Sie ein C#-Programm, das ein Formular unter Windows anzeigt und in dem je eine int, long, float, double, char, string Variable deklariert und mit einem Wert belegt wird.

Anschließend soll das Programm diese Werte in dem Formular auf dem Bildschirm anzeigen.

### **6.5.3 Rechner unter Windows programmieren**

Erstellen Sie ein C#-Programm für Windows, das über ein Formular zwei Zahlen einliest, diese dann auf Knopfdruck addiert und das Ergebnis in einem Fenster ausgibt.

## 7 Programm-Steuerungen

### 7.1 Grundlagen

Zur Steuerung des Programmflusses des funktionalen Quell-Codes kennt jede Programmiersprache Steueranweisungen. Dazu zählen sog. bedingte Anweisungen (z.B. if, switch), mit denen man logische Entscheidungen programmieren kann, und Programmschleifen (z.B. while, do, for), mit denen man Teile des Programm-Codes wiederholt durchlaufen kann.

### 7.2 if

Häufig muss man auf Grund irgendwelcher Ereignisse alternative Programmzweige vorsehen und eine Verzweigung im Programmfluss einbauen. Dies kann man am einfachsten mit dem **if**-Statement erreichen. Optional kann man nach einem **if** auch noch ein **else** einbauen.

if prüft einen bestimmten logischen Sachverhalt und führt nach positiver Prüfung die nächste im Quell-Code folgende Anweisung aus. Bei negativer Prüfung wird die übernächste Anweisung ausgeführt. Sollen nach einem if-Statement mehrere Anweisungen ausgeführt werden, so müssen diese in einem {}-Block eingefasst werden. Gleiches gilt für else.

Man kann nach einem if-Statement auch grundsätzlich {} setzen, auch wenn nur eine Anweisung ausgeführt werden soll, und die Klammern damit nicht zwingend erforderlich sind. Diese Praxis hat den Vorteil, dass optisch ganz klar zum Ausdruck kommt was gemeint ist, und dass später ggf. leicht noch weitere Zeilen hinzugefügt werden können, ohne dass man dann die {} vergisst.

Syntax:

```
if (Ausdruck)
 Anweisung;
else
 Anweisung;
oder
if (Ausdruck)
{
 Anweisung1;
 Anweisung2;
}
```

Der Typ des Ausdruckes ist bool, d.h. if prüft nur auf TRUE oder FALSE

Logische Prüfungsmöglichkeiten (Operatoren) ([siehe auch](#)):

- == Prüfung auf Gleichheit
- >=
- <=
- >
- <
- != Prüfung auf Ungleichheit

z.B.:

```
...
if (a==0)
 MessageBox.Show("Achtung, a=0");
else
 MessageBox.Show("Der Kehrwert von a = " + 1/a);
...
```

**Achtung:**

```
if (last = 50) // Das Erzeugt einen Compiler-Fehler
if (last == 50) // So ist es richtig
```

## 7.3 switch

Soll eine Fallunterscheidung von mehreren Fällen erfolgen, kann man dies mit einer Reihe von *if* und *else if* Anweisungen durchführen. z.B.:

```
if (tag == 0)
 tagName = "Sonntag";
else if (tag == 1)
 tagName = "Montag";
... usw.
```

Eleganter ist in so einem Fall das **switch**-Statement, dessen Syntax nach folgenden Regeln funktioniert:

```
switch (kontrollAusdruck)
{
 case konstanterAusdruck :
 Anweisungen;
 break;

 case konstanterAusdruck :
 Anweisungen;
 break;

 case konstanterAusdruck :
 Anweisungen;
 break;

 ...

 default :
 Anweisungen;
 break;
}
```

kontrollAusdruck muss vom Typ Integer oder String sein. Er wird einmal ausgewertet und das Programm springt dann in den Fall, dessen konstanterAusdruck mit dem kontrollAusdruck übereinstimmt. Sollte es keine Übereinstimmungen geben, wird der *default*- Fall angesprungen. Die Berücksichtigung eines *default*-Falles ist optional.

Das *break*-Statement bewirkt, dass der switch-Block an dieser Stelle abgebrochen und verlassen wird.

Obiges Beispiel würde mit *switch* so aussehen:

```
switch (tag)
{
 case 0 :
 tagName = "Sonntag";
 break;
 case 1 :
 tagName = "Montag";
 break;
 case 2 :
 tagName = "Dienstag";
 break;
 ...
 default :
 tagName = "unbekannt";
 break;
}
```

**Regeln für switch:**

- switch kann nur für die einfache Datentypen *int* und *string* verwendet werden
- Die case-Klauseln müssen Konstanten sein und dürfen nicht berechnet werden
- Können diese Regeln nicht eingehalten werden, muss nach obigen Muster *if* verwendet werden
- Ein "Durchfallen" von Fällen, indem man *break* weglässt, ist nicht erlaubt (Achtung C, C++ und Java-Programmierer, das ist in diesen Sprachen anders!!!). *break* muss nach jedem case gesetzt werden, auch nach default, sonst verursacht man einen Compiler-Fehler.  
Das wäre falsch:

```
switch (tag)
{
 case 1 : //das ist nicht erlaubt!!!
 case 2 :
 tagName = "Erster Teil der Woche";
 break;
 case 3 : //das ist nicht erlaubt!!!
 case 4 : //das ist nicht erlaubt!!!
 case 5 :
 tagName = "Zweiter Teil der Woche";
 break;
 default :
 tagName = "Wochenende";
 break;
}
```

**7.4 break**

Wie schon beim *switch* ([siehe](#)) erklärt, kann man die **break**-Anweisung einsetzen, um die Ausführung einer bestimmten Programmsteuerung schlagartig zu beenden, ohne sie noch einmal auszuführen

**7.5 continue**

Im Gegensatz zum *break*, wird durch die Anweisung **continue** veranlaßt, dass das Programm weiterläuft und eine übergeordnete Programmsteuerung sofort ausgeführt wird.

*break* und alternativ *continue* werden meist zur Steuerung von Schleifen-Durchläufen eingesetzt. ([siehe](#))

**7.6 Programm-Schleifen****7.6.1 Schleifen für Wiederholungen**

Häufig stellt sich einem beim Programmieren die Aufgabe, einen Teil des Programmcodes wiederholt auszuführen zu müssen. Dazu dienen Programmanweisungen für Schleifen.

Schleifen führen einen bestimmten Programmcode so lange aus, bis eine Abbruchbedingung erreicht wurde. Die Überprüfung, ob die Abbruchbedingung erreicht wurde, kann, je nach Typ, am Anfang oder am Ende der Schleife durchlaufen werden.

Erfolgt die Überprüfung am Schleifenanfang, kann das dazu führen, dass die Schleife evtl. auch überhaupt nicht durchlaufen wird. Falls das Abbruchkriterium gleich zu Beginn erfüllt wird, spricht man von einer sog. abweisenden Schleife.

Erfolgt die Überprüfung erst am Schleifenende, führt das dazu, dass der betreffende Programmcode auf alle Fälle wenigstens einmal durchlaufen wird.

Neben diesem Unterscheidungsmerkmal, gibt es Schleifentypen die auf ein bestimmtes Ereignis warten, bis sie abgebrochen werden. Typischer Weise, weiß man beim Programmieren noch nicht, wie oft diese Schleife durchlaufen wird, da das Abbruchkriterium evtl. durch das Verhalten des Anwenders bestimmt wird.

Weiß man jedoch schon beim Programmieren, wie oft eine Schleife ausgeführt werden soll, wird man einen Typ verwenden, bei dem man die Anzahl der Durchläufe vorherbestimmen kann. Typische Anwendungsfälle dafür sind z.B. Algorithmen oder der Zugriff auf Matrizen.

Folgende Tabelle gibt eine Übersicht der verfügbaren Schleifentypen und ihrer Eigenschaften.

| Schleifentyp    | Prüfung am       | Abbruchkriterium auf Grund von |
|-----------------|------------------|--------------------------------|
| <b>while</b>    | Schleifen-Anfang | Ereignis                       |
| <b>do-while</b> | Schleifen-Ende   | Ereignis                       |
| <b>for</b>      | Schleifen-Anfang | Anzahl der Durchläufe          |

### 7.6.2 while-Schleife

**while**-Schleifen werden verwendet um eine oder mehrere Anweisungen so lange zu wiederholen, solange ein Ausdruck vom typ bool den Wert **true** besitzt.

Syntax:

```
while (boolescherAusdruck)
 Anweisung;
```

Sollen mehrere Anweisungen wiederholt ausgeführt werden, so sind diese in { } einzuklammern:

```
while (boolescherAusdruck)
{
 AnweisungX;
 AnweisungY;
 ...
 AnweisungZ;
}
```

Im folgendem Beispiel werden in einem Programmfragment die Zahlen von 0 bis 9 am Bildschirm dargestellt:

```
int i = 0;

while (i != 10)
{
 Console.WriteLine(i);
 i++;
}
```

Die Überprüfung ( $i \neq 10$ ) ergibt so lange den Wert *true*, solange die Variable *i* von 0 aufwärts zählend kleiner gleich 9 bleibt. Erreicht die Variable *i* den Wert 10, ergibt die Überprüfung ( $i \neq 10$ ) den Wert *false* und die Ausführung der Schleife wird abgebrochen.

### 7.6.3 do-Schleifen

Die **do**, besser die **do-while**-Schleife, ist eine while-Schleife, bei der die Abbruchbedingung erst am Schleifenende getestet wird. D.h., der Programmcode in der Schleife wird auf alle Fälle einmal durchlaufen und erst dann wird mit einer while- Anweisung geprüft, ob weitere Durchläufe erfolgen sollen.

Syntax:

```
do
 Anweisung;
while (boolescherAusdruck);
```

Auch hier müssen mehrere Anweisungen in { } Klammer eingefaßt werden:

```
do
{
 AnweisungX;
 AnweisungY;
 ...
 AnweisungZ;
}
while (boolescherAusdruck);
```

Beispiel:

```
int i = 0;

do
{
 Console.WriteLine(i);
 i++;
}
while (i != 10);
```

### 7.6.4 for-Schleifen

Weiß man schon zum Zeitpunkt der Programmerstellung, wie oft eine Schleife durchlaufen werden soll, verwendet man die **for**- Schleife.

Syntax:

```
for (Initialisierung; boolescherAusdruck; aktualisiereKontrollVariable)
 Anweisung;
```

Auch hier gilt, dass mehrere zu wiederholende Anweisungen in { } einzuklammern sind:

```
for (Initialisierung; boolescherAusdruck; aktualisiereKontrollVariable)
{
 AnweisungX;
 AnweisungY;
 ...
 AnweisungZ;
}
```

In folgender Reihenfolge werden die einzelnen Komponenten ausgewertet und ausgeführt:

- **Initialisierung findet nur einmal zu Beginn statt**
- **Überprüfung des booleschen Ausdrucks**
  - Ausführung der Anweisungen
  - KontrollVariable wird aktualisiert
- **Überprüfung des booleschen Ausdrucks**
  - usw.

Folgendes Beispiel eines Programmfragmentes errechnet in Einer- Schritten das Quadrat einer Zahl zwischen 0 und 10:

```
double x = 0.0;

for(int i=0; i <= 10; i++)
{
 x = Math.Pow(x, 2.0);
 Console.WriteLine("x= " + x);
 x++;
}
```

Man darf jeden der drei Teile einer for-Schleife auch weglassen, aber Achtung! Folgendes Beispiel ergibt eine Endlos- Schleife:

```
for(int i=0; ; i++)
{
 Console.WriteLine("Hilfe! Wer hält mich an?");
}
```

Folgendes wäre eine umständlich und merkwürdig formulierte while-Schleife (vergleiche mit dem Beispiel im Kapitel der while-Schleife):

```
int i = 0;
for(; i != 10 ;)
{
 Console.WriteLine(i);
 i++;
}
```

So würde das Beispiel aus dem Kapitel der while-Schleife aussehen, wenn man es als for- Schleife implementiert:

```
for(int i=0; i != 10 ; i++)
{
 Console.WriteLine(i);
}
```

### Anmerkung:

Eine for-Schleife kann man auch als while- Schleife implementieren:

Statt:

```
for (Initialisierung; boolescherAusdruck; aktualisiereKontrollVariable)
 Anweisung;
```

könnte man auch (umständlich) programmieren:

```
Initialisierung;
while (boolescherAusdruck)
{
 Anweisung;
 aktualisiereKontrollVariable;
}
```

## Geltungsbereich des Schleifenzählers

Wird eine Zähl-Variable im Kopf der for-Schleife initialisiert, so erstreckt sich ihr Geltungsbereich nur auf diese for-Schleife, und nicht etwa auf die gesamte Methode.

Folgendes Beispiel würde einen Compiler-Fehler erzeugen:

```
for(int i=0; i != 10 ; i++)
{
 Anweisungen;
 ...
}
Console.WriteLine(i); //nicht definierte Variable i
```



## 7.6.5 break, continue

Setzt man **break** und **continue** in Programmschleifen ein, bewirkt ...

- **break**, dass die Abarbeitung der Schleife sofort beendet wird. Es werden keine Überprüfungen oder Aktionen der Schleifen- Steuerung mehr ausgeführt
- **continue**, dass sofort wieder in die Schleifensteuerung gesprungen wird, und die dem continue folgenden restlichen Anweisungen der Schleife nicht mehr ausgeführt werden. Nach continue wird also der boolesche Ausdruck sofort wieder ausgewertet.

Beispiel:

```
int i = -10;

while (true)
{
 Console.WriteLine(i);
 i++;

 if (i == 0)
 continue;
 else
 Console.WriteLine("Kehrwert = " + 1/i);

 if (i != 10)
 continue;
 else
 break;
}
```

### Kritische Anmerkung:

Finden Sie den obigen Programmcode nicht etwas unübersichtlich? Der Eindruck täuscht nicht, denn die Steuerung von Schleifendurchläufen mit *continue* führt leider meist zu unübersichtlichen und schwer wartbarem Programmcode. Erfahrene Programmierer versuchen deshalb möglichst ohne *continue* auszukommen. Falls es sich nicht vermeiden lässt, wir der entsprechende Programmcode gut kommentiert.

## 7.7 Übung 4

Ein Einfeldträger sei mit einer Einzellast belastet, die sich vom linken zum rechten Auflager bewegt.

Erstellen Sie ein Programm, das die Auflagerkräfte der beiden Auflager in Abhängigkeit von Last und ihrem Angriffspunkt errechnet. Die Ergebnisse sollen in 1/10-Schritten der wandernden Last ausgegeben werden. Vom Anwender soll die Größe der Last sowie die Länge des Einfeldträgers eingeben können.

Die Ergebnisse sind auf dem selben Form darzustellen, auf dem der Anwender die Eingabewerte eintippen kann. Verwenden Sie dazu eine List-Box. Die Ergebniswerte können in der Listbox nach folgendem Muster aufgelistet werden:

```
meinListBoxName.Items.Add("...." + meineVariable);
```

Nachdem Sie Ihr Programm erfolgreich zum Laufen gebracht haben, setzen Sie den Cursor in den Teil des Programmcodes, den Sie erstellt haben, starten Sie den Debugger und lassen Sie dann den Debugger Schritt für Schritt laufen. Beobachten Sie, wie sich Ihr Programm verhält.

## 8 Methoden

### 8.1 Grundlagen

In den bisherigen Beispielen haben wir Methoden bereits intuitiv kennengelernt. Jetzt sollen sie detailliert erklärt werden:

Methoden gehören (neben Klassen und Namespaces) zu den Sprachelementen in C#, mit denen man Quellcode strukturieren kann. Siehe auch [Architektur eines C#-Programmes](#).

Programmierer, die nicht-objektorientierte Sprachen kennen, liegen mit der Vorstellung, dass Methoden so etwas wie "Unterprogramme" oder "Funktionen" sind, nicht ganz falsch. Dennoch wird für diese Struktur in objektorientierten Sprachen der Begriff "Methode" verwendet, um auszudrücken, dass sie doch nicht ganz dasselbe sind, denn übergeordnet gibt es ja außerdem noch Klassen und Namespaces.

In C# enthalten die Methoden sämtlichen ausführbaren Quellcode, also alle Programmanweisungen, die mit ; abgeschlossen werden.

Wenige Ausnahmen von dieser Regel sind die Deklarationsanweisungen von Feldern, also Variablen, deren Gültigkeit sich über eine ganze Klasse, und damit über mehrere Methoden erstrecken, und die `using`-Anweisungen, über die eine Referenz auf andere Namespaces hergestellt wird.

Methoden sind also eine Sammlung von Anweisungen, die unter einem Namen zusammengefasst werden.

Eine besondere Methode ist die **Main-Methode**: sie ist die erste Methode, die in einer Anwendung ausgeführt wird. Jeder Anwendung muss eine Main-Methode enthalten.

### 8.2 Syntax

Wie schon früher erläutert, bestehen Methoden aus einer Kopfzeile, die den Namen mit einer Parameterliste enthält, und einem Rumpf, der in geschweiften Klammern eingeschlossen wird. Der Rumpf enthält die ausführbaren Programmanweisungen einer Methode.

```
rückgabeTyp methodeName(parameterliste) // Die Parameterliste kann leer
sein
{
 // Methodenkörper, hier folgen die Anweisungen
 ...;
 ...;
}
```

Falls die Parameterliste leer ist, müssen nach dem Namen trotzdem die () Klammern folgen

#### Aufruf von Methoden:

Methoden werden aufgerufen, indem man ihren Namen aufruft. Da Methoden Werte zurückliefern können, können sie auch rechts vom = -Zeichen stehen oder selbst wieder als Parameter in einem Methodenaufruf eingesetzt werden. z.B.:

```
eineMethode(a,b,c);
int a = eineAndereMethode(x, y);
eineMethode(eineAndereMethode(x, y) , b, c);
```

## 8.3 Methoden mit und ohne Rückgabewerte

Es gibt zwei verschiedene Arten von Methoden:

- Solche, die einen Wert zurückliefern
- und solche, die keinen Wert zurückliefern

### Methoden mit Rückgabe eines Wertes

Methoden können als Ergebnis einen Wert zurückliefern. Dieser Rückgabewert wird weiterverarbeitet, indem entweder die Methode rechts vom =-Zeichen aufgerufen und z.B. einer Variablen zugewiesen wird, oder aber, indem die Methode als Parameter in einer Parameterliste eines anderen Methodenaufrufes gesetzt wird. Auf Grund dieser Eigenschaft muss auch die Methode einen Typ besitzen. Die Deklaration des **Types der Methode** muss mit der Deklaration des **Types des Rückgabewertes** übereinstimmen. Die Rückgabe eines Wertes geschieht über die **return**-Anweisung, z.B.:

```
int addiereWerte(int ersterWert, int zweiterWert)
{
 return ersterWert + zweiterWert;
}
```

In einer anderen Methode könnte dann die *addiereWerte*-Methode wie folgt aufgerufen werden:

```
...
...
int a = 2;
int b = 3;
...
...
int ergebnis = addiereWerte(a, b);
...
...
```

### Methoden ohne Rückgabe eines Wertes

Liefern Methoden keinen Wert zurück, so muss Ihr Typ **void** sein. Wir haben schon einige Methoden vom Typ void gesehen, z.B. die **Main-Methode**. Da sie die erste Methode ist, die in einer Anwendung ausgeführt wird, kann sie keinen Wert zurückliefern. Deshalb muss der Typ der Main- Methode void sein. z.B.:

```
void zeigeErgebnis(int ergebnis)
{
 MessageBox.Show("Das Ergebnis ist: " + ergebnis);
}
```

Auch in Methoden vom Typ void kann die **return**-Anweisung verwendet werden, um einen sofortigen Abbruch ihrer Ausführung zu erwirken. Allerdings darf hier hinter der return-Anweisung kein Wert zurückgegeben werden, d.h. die return-Anweisung wird unmittelbar mit einem ; abgeschlossen. z.B.:

```
void zeigeKehrrbruchVonErgebnis(int ergebnis)
{
 //Falls Ergebnis = 0, die Methode abbrechen
 if (ergebnis == 0)
 return;

 MessageBox.Show("Das Ergebnis ist: " + 1/ergebnis);
}
```

## 8.4 Call by value / Call by reference

Parameter können beim Aufruf einer Methode nach zwei Modellen übergeben werden:

- Call by Value
- Call by Reference

Damit ist folgendes gemeint:

### Call by Value:

Im Standard-Fall werden in der Parameterliste einer Methode **Variablen mit einfachen Datentypen** übergeben (z.B. int, double, string, usw) (sog. **Werttypen**). Beim Aufruf einer solchen Methode, wird für diese Variablen in der Parameterliste eine Kopie im Speicher angelegt, die so lange Gültigkeit hat, so lange die Methode ausgeführt wird. Nach Abschluss der Methode kann der Garbage- Collector die für die Variablen in der Parameterliste angelegte Speicher- Kopie wieder an das Betriebssystem freigeben.

Änderungen der Werte der Variablen der Parameterliste, die während der Ausführung der gerufenen Methode erfolgen, haben also nur in der gerufenen Methode und nur so lange Gültigkeit, bis diese Methode abgeschlossen wurde. Änderungen der Werte der Variablen der Parameterliste haben damit keine Rückwirkende Auswirkung auf die übergeordnete Methode, in der gerufene Methode aufgerufen wird.

Die Speicherbereiche von rufender und gerufener Methode sind strikt voneinander getrennt.

z.B.:

```
void Main ()
{
 ...
 ...
 int a = 2;
 int b = 3;
 ...
 ...
 int ergebnis = erhöheUndAddiereWerte(a, b);
 // In der Methode erhöheUndAddiereWerte wurden beim Aufruf für die
 Variablen
 // a und b jeweils eine Speicherkopie angelegt, auf die über die
 Variablen
 // ersterWert und zweiterWert zugegriffe werden kann.
 // Nach Abschluss der Methode erhöheUndAddiereWerte bleiben die Werte
 der
 // Variablen a und b unverändert.
 ...
 ...
}

int erhöheUndAddiereWerte(int ersterWert, int zweiterWert)
{
 ersterWert++;
 zweiterWert++;
 return ersterWert + zweiterWert;
}
```

### Call by Reference:

Oft werden aber auch **Variablen mit komplexen Datentypen** in der Parameterliste einer Methode übergeben. Das sind dann i.d.R. Objekte (Klassen) (sog. **Referenztypen**), so z.B. auch mehrdimensionale Variablen wie Vektoren oder Matrizen (werden später erklärt). In diesem Fall wird keine Kopie des betroffenen Speicherbereiches erzeugt, sondern es wird eine Referenz (C-Programmierer würden sagen, ein "Pointer") auf den Speicherbereich übergeben, in dem das Objekt gespeichert ist.

Ändert die gerufene Methode Werte der in der Parameterliste übergebenen Variablen, so bleibt nach Abschluss der gerufenen Methode diese Änderung auch für die rufende Methode erhalten. Das kann gewünscht sein, kann aber bei Unachtsamkeit auch zu überraschenden und ungewünschten Seiteneffekten führen.

Rufende und gerufene Methode greifen auf den selben Speicherbereich zu.  
Hierzu werden später bei passender Gelegenheit Beispiele gezeigt werden.

### 8.4.1 Der Stack und Heap

Der Speicher teilt sich in der Verwaltung durch den Hauptprozessor in zwei Bereiche auf, den **Stack** und den **Heap**:

#### Stack

Im Stack werden Datenelemente nach dem Prinzip Last-in-First-out verwaltet. D.h. was zuletzt hineingeschrieben wurde, wird als erstes wieder entnommen. Man kann sich das wie bei einem Stapel Teller vorstellen. Lokale Variablen, sog. Werttypen, werden in C# in den Stack geschrieben. Dort ist ein Speicherbereich reserviert, den der/die ProgrammiererIn mit dem Namen (Bezeichner) der Variablen referenziert. Beim Aufruf einer Methode, wird im Stack eine Kopie für die Werttyp- Variablen angelegt, die in der Parameter-Liste übergeben werden). Für Jede Methode wird für die darin enthaltenen lokalen Variablen ein Stack-Frame (Rahmen) aufgebaut, wodurch die Gültigkeit der lokalen Variablen eindeutig bestimmt ist. Dies gilt z.B. auch für Laufvariablen (Schleifenzähler) einer for-Schleife, die solange gelten, bis die for-Schleife abgearbeitet ist. (siehe auch [for-Schleifen](#) und [Geltungsbereich von Variablen](#))

#### Heap

Objekte, sog. Referenztypen, werden im Heap angelegt. Dabei wird eine Adresse zurückgegeben, die der/die ProgrammiererIn mit einem Namen (Bezeichner) für das Objekt referenziert. Beim Aufruf einer Methode wird die Adresse übergeben, d.h. dass sowohl die rufende, als auch die gerufene Methode auf das selbe Objekt referenzieren und es damit auch verändern können. Referenz-Typen sind z.B. auch Arrays.

#### Garbage Collector

Nachdem eine Methode abgearbeitet ist, ist auch der dafür aufgebaute Stack-Frame zu Ende. Alle diesem Frame zugehörigen lokalen Variablen werden für den Garbage Collector markiert. Der Garbage Collector zerstört dann nach einiger Zeit alle markierten Variablen, d.h. er gibt ihre Speicheradressen wieder an die Verwaltung des Prozessors und des Betriebssystems frei.

Sobald eine Methode, die auf ein Objekt referenziert, abgearbeitet ist, wird auch die Referenz auf das Objekt zerstört. Der Garbage Collector räumt Objekte aus dem Heap, sobald die letzte Referenz auf sie zerstört wurde.

## 8.5 Bezeichner überladen

Werden innerhalb eines Geltungsbereiches zwei identische Bezeichner verwendet, quittiert dies der Compiler mit einer Fehlermeldung. Die gilt z.B. für Klassen, Methoden, Variablen, Felder, usw.

Man spricht in diesem Fall davon, dass die Bezeichner überladen sind. Während das ein häufiger Programmierfehler ist, kann man diese Eigenschaft auch ganz gezielt einsetzen.

Werden im selben Geltungsbereich (z.B. innerhalb einer Klasse) zwei Methoden mit identischem Bezeichner definiert, würde der Compiler einen Fehler melden. Trotzdem ist es oft erforderlich und nützlich, wenn dies doch möglich wäre.

Ermöglicht wird dies, indem sich zwei Methoden mit identischem Bezeichner und Typ trotzdem in den Typen der Parameterliste unterscheiden können. Als Beispiel sei hier genannt die im ersten Programm verwendete Methode.

```
Console.WriteLine("Das ist ein schöner Tag");
```

```
Console.WriteLine(int a = 2)
```

Einmal erwartet sich WriteLine einen string als Parameter, im zweiten Fall erwartet sie einen Integer Wert. D.h. die Methode Console.WriteLine liegt in mehrfachen Versionen vor, die sich in ihren Parameterlisten unterscheiden. Der Compiler erkennt das als unterschiedlich an und findet beim Aufruf die passende Methode.

## 8.6 Merke

- Der Typ der Methode muss dem Typ des Rückgabewertes entsprechen
- Methoden geben mit der **return**-Anweisung Werte zurück
- Gibt eine Methode **keinen Wert** zurück, muss ihr Typ **void** sein
- Mit einer **return**-Anweisung ohne Wert, kann eine Methode vom Typ void an jeder Stelle im Programmablauf verlassen werden
- Methoden mit gleichem Namen und gleichem Typ, aber mit unterschiedlichen Parameterlisten, sind jeweils andere Methoden. Parameterlisten können sich in Typ und Anzahl der einzelnen Parameter unterscheiden. In solchen Fällen sagt man, dass ein **Bezeichner überladen** wird.
- Die Methode, die als **erste** in einer Anwendung durchlaufen wird, muss **Main** heißen. Ihr Typ ist **void**
- In Parameterlisten von Methoden werden ...
  - ... Variablen mit einfachen Datentypen als Wert übergeben (= call by Value)
  - ... Variablen mit komplexen Datentypen als Referenz übergeben (= call by Reference)
- siehe auch [Gültigkeit von Variablen](#)

## 8.7 Übung 2

### 8.7.1 Rechner unter Windows mit Methoden

Erstellen Sie ein C#-Programm für Windows, das über ein Formular zwei Zahlen einliest, diese dann auf Knopfdruck addiert und das Ergebnis in einem Fenster ausgibt. Erstellen Sie für die verschiedenen Funktionen des Rechners je eine Methode.

## 9 Auswahl-Anweisungen

### 9.1 Steuerung von Alternativen

Komplexe Problemlösungen bedingen, dass es mehrere Lösungsalternativen gibt und deshalb der Programmfluss auf Grund von Fallunterscheidungen in verschiedene Äste aufgespaltet und dann entsprechend gesteuert werden muss. Dazu dienen sog. Auswahlanweisungen.

In C# gibt es zu diesem Zweck:

- if-Anweisungen ([siehe](#))
- switch-Anweisungen ([siehe](#))

### 9.2 Boolesche Variablen

Zur Steuerung von Auswahlanweisungen werden Variablen mit einem speziellen Datentyp verwendet, die als Wert entweder "*wahr*" oder "*falsch*" annehmen können. In Anlehnung an den Bereich der Aussagen-Algebra in der Mathematik, die maßgeblich von Boole entwickelt worden ist und in der es um die Theorie und Verarbeitung logischer Aussagen geht, wird dieser einfachste Datentyp in C# "*bool*" genannt.

Alle Aussagen, die eindeutig entweder mit "*wahr*" oder "*falsch*" beantwortet werden können, werden mit booleschen Variablen codiert.

z.B.:

```
bool derTrägerIstAusreichendDimensioniert;
derTrägerIstAusreichendDimensioniert = true;
Console.WriteLine(derTrägerIstAusreichendDimensioniert); // gibt true aus
```

### 9.3 Boolesche Operatoren

Neben den booleschen Variablen gibt es zur Verarbeitung logischer Aussagen außerdem boolesche Operatoren, die als Ergebnis entweder "*true*" oder "*false*" liefern.

#### 9.3.1 Negation

Der einfachste boolesche Operator ist die Negation, die eine logische Aussage in ihr Gegenteil umkehrt. Dieser Operator wird in C# mit **!** codiert.

z.B.:

```
!false entspricht true
!true entspricht false
!= entspricht ungleich
```

Anmerkung:

Operatoren mit nur einem Operanden, wie das **!**, nennt man unäre Operatoren. Operatoren, die zwei Operanden besitzen nennt man binäre Operatoren

### 9.3.2 Relationale Operatoren

Relationale Operatoren vergleichen zwei Werte auf Grund einer bestimmten Vorschrift und liefern als Ergebnis, ob die Vorschrift eingehalten wurde, oder nicht.

z.B.:

Die Variable *last* sei mit 50 (kN) vorbelegt

| Operator | Bedeutung          | z.B.        | Ergebnis |
|----------|--------------------|-------------|----------|
| ==       | gleich             | last == 250 | false    |
| !=       | ungleich           | last != 0   | true     |
| <        | kleiner als        | last < 40   | false    |
| >        | größer als         | last > 40   | true     |
| <=       | kleiner gleich als | last <= 50  | true     |
| >=       | größer gleich als  | last >= 60  | false    |

### 9.3.3 Logische Operatoren

Neben der Negation stellt C# noch zwei weitere logische Operatoren zur Verfügung, den UND- Operator, codiert mit **&&**, sowie den ODER-Operator, codiert mit **||**.

Diese Operatoren erlauben, boolesche Ausdrücke zu verbinden, um dadurch auch logisch komplexere Aussagen codieren zu können. Wie die relationalen Operatoren liefern auch die logischen Operatoren als Ergebnis entweder "*true*" oder "*false*" zurück. Folgende Wahrheitstafel gibt eine Übersicht über die Wirkung der logischen Operatoren:

| a     | b     | UND: Ergebnis von a&&b |
|-------|-------|------------------------|
| true  | true  | true                   |
| false | false | false                  |
| false | true  | false                  |
| true  | false | false                  |

| a     | b     | ODER: Ergebnis von a  b |
|-------|-------|-------------------------|
| true  | true  | true                    |
| false | false | false                   |
| false | true  | true                    |
| true  | false | true                    |



## UND-Operator

Im folgenden Beispiel soll die Belastung für einen Träger von einem Textfeld aus einer Eingabeoberfläche geliefert werden. Die Last soll nur für Werte zwischen 0 kN und 50 kN gültig sein. Eine logische Variable soll das Ergebnis dieser Untersuchung speichern:

```
double last = double.Parse(textBox1.Text);
bool gültigeLast;
gültigeLast = (last >= 0) && (last <= 50);
if (gültigeLast)
{
 //Berechne Träger
 ...
 MessageBox.Show("Das Ergebnis ... =" + ...);
}
else
 MessageBox.Show("Berechnung nicht möglich, Last ungültig!");
```

Nur in den Fällen, wo die Variable *last* einen Wert zwischen 0 und 50 trägt, wird die logische Variable *gültigeLast* den Wert *true* zugewiesen bekommen, in allen anderen Fällen bekommt sie den Wert *false* zugewiesen.

**Achtung:** die folgende Zeile würde einen Compiler-Fehler verursachen:

```
gültigeLast = (last >= 0 && <= 50); // erzeugt Compiler-Fehler
```

## ODER-Operator

Im folgenden Beispiel soll eine Lasteingabe für einen Zweifeldträger erfolgen, und nur dann gültig sein, wenn wenigstens eines seiner Felder mit einer Last größer Null belastet ist. Auch das Ergebnis dieser Untersuchung wird in einer logischen Variablen gespeichert:

```
double lastFeld1 = double.Parse(textBox1.Text);
double lastFeld2 = double.Parse(textBox2.Text);
bool gültigeLast;
gültigeLast = (lastFeld1 > 0) || (lastFeld2 > 0);
if (gültigeLast)
{
 //Berechne Zweifeldträger
 ...
 MessageBox.Show("Das Ergebnis ... =" + ...);
}
else
 MessageBox.Show("Berechnung nicht möglich, Last ungültig!");
```

Die Variable *gültigeLast* bekommt den Wert *true* bereits zugewiesen, wenn entweder *lastFeld1* oder *lastFeld2* größer als Null ist. Natürlich ist das Ergebnis auch für den Fall *true*, wenn beide Felder mit einer positiven Last belastet werden.

### Anmerkung:

Alternativ kann man die logische Untersuchung auch gleich direkt in das if- Statement einbauen. Das Programm würde dann so aussehen:

```
double lastFeld1 = double.Parse(textBox1.Text);
double lastFeld2 = double.Parse(textBox2.Text);
if ((lastFeld1 > 0) || (lastFeld2 > 0))
{
 //Berechne Zweifeldträger
 ...
 MessageBox.Show("Das Ergebnis ... =" + ...);
}
else
 MessageBox.Show("Berechnung nicht möglich, Last ungültig!");
```

oder noch kompakter, dafür aber auch unübersichtlicher:

```

if ((double.Parse(textBox1.Text) > 0) || (double.Parse(textBox2.Text) >
0))
{
 //Berechne Zweifeldträger
 ...
 MessageBox.Show("Das Ergebnis ... =" + ...);
}
else
 MessageBox.Show("Berechnung nicht möglich, Last ungültig!");

```

### 9.3.4 Vorrang

Zum Vorrang und Assoziativität von logischen Operatoren [siehe Übersicht](#).

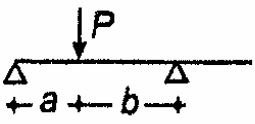
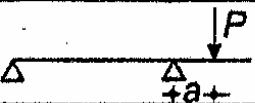
## 9.4 Übung 3

### 9.4.1 Ermittlung der Schnittkräfte für Träger mit Kragarm

Für einen Träger mit Kragarm sollen für die Belastung mit einer Einzellast die Auflagerkräfte A und B, sowie das Biegemoment im Feld (Mf) und über der Stütze (Mk) berechnet werden.

Dabei sollen die beiden Lastfälle

- Lastangriff im Feld und
- Lastangriff auf dem Kragarm unterschieden werden.

| System                                                                              | A               | B                               | Mf              | Mk    |
|-------------------------------------------------------------------------------------|-----------------|---------------------------------|-----------------|-------|
|  | $\frac{Pb}{l}$  | $\frac{Pa}{l}$                  | $\frac{Pab}{l}$ | 0     |
|  | $-\frac{Pa}{l}$ | $\left(1 + \frac{a}{l}\right)P$ | $M_k$           | $-Pa$ |

Erstellen Sie ein Windows-Programm, in dem der Anwender die erforderlichen Eingaben machen kann, und das die Ergebnisse berechnet und ausgibt.

## 10 Arrays

### 10.1 Grundlagen

Arrays sind mehrdimensionale Variablen. Sie werden in C# als Objekte angelegt, d.h. sie liegen im Heap, und ihre Namen (Bezeichner) stehen stellvertretend für Referenzen auf sie. Werden Arrays als Parameter beim Aufruf einer Methode übergeben, werden demnach Referenzen übergeben (siehe [Call by Reference](#)).

Diese Eigenschaft kann man sich zu Nutze machen, wenn eine Methode mehr als einen Wert zurückgeben soll. Über die return-Anweisung kann immer nur ein Wert zurückgegeben werden. Will man aber aus einer Methode mehrere Ergebnisse zurückliefern, kann man ein Array an sie übergeben. Die gerufene Methode verändert dann die Inhalte der Array-Elemente, die dann danach auch der rufenden Methode zur Verfügung stehen.

Die Tatsache, dass Arrays keine einfachen Werttypen, sondern komplexe Objekttypen sind, verleiht ihnen die angenehme Eigenschaft, dass sie über einen ganzen Satz von praktischen Methoden verfügen, die den Zugriff auf sie, und ihre Verwaltung erleichtern.

Es gibt eindimensionale und mehrdimensionale Arrays. In der Mathematik verwendet man eindimensionale Arrays für Vektoren und mehrdimensionale für Matrizen.

#### Achtung ProgrammiererInnen aus andern Sprachen:

Während in der englischen Literatur auch bei anderen Programmiersprachen in diesem Kontext immer von Arrays gesprochen wird, wurde in der deutschen Literatur bei anderen Sprachen (z.B. Java) für diese Datenstrukturen der Begriff "Felder" eingeführt. Die deutsche C#-Literatur verwendet den Begriff "Felder" jedoch für Variablen, deren Geltungsbereich sich über eine ganze Klasse erstreckt. Auf Grund dieser Inkonsistenz können also leicht Irrtümer und Missverständnisse entstehen.

### 10.2 Arrays deklarieren

Arrays werden mit einem Datentyp deklariert und anschließend instantiiert. D.h. aus einer Mutterklasse aller Arrays wird eine bestimmte Instanz eingerichtet, mit der man dann arbeiten kann. Dies geschieht mit Hilfe des **new**- Operators. Bildlich gesprochen, wird mit einer Kuchenform (= Klasse) einmal ein Rosinen- und einmal ein Marmor-Kuchen (= Instanzen) gebacken.

Syntax:

```
Typ[] Array-Name;
```

z.B.:

```
int[] meinIntArray;
```

Die [] teilen dem Compiler mit, dass ein Array angelegt werden soll, in diesem Fall vom Typ *int*, mit dem Bezeichner *meinIntArray*.

Nun muss das Feld instantiiert werden:

```
meinIntArray = new int[5];
```

Damit wird zum Ausdruck gebracht, dass zu dem Array 5 Elemente gehören.

Man kann auch verkürzt schreiben:

```
int [] meinIntArray = new int[5];
```

## 10.3 Zugriff auf Arrays

### 10.3.1 Einzelzugriff und über Schleifen

Jedes Array-Element wird über eine numerische Adresse angesprochen, die Zählung beginnt bei 0.

z.B.:

```
int[] meinIntArray = new int [5];
meinIntArray[0] = 1;
meinIntArray[1] = 3
usw.
meinIntArray[4] = 9;
```

Da die Zählung der Element-Adressen bei 0 beginnt, ist in diesem Fall 4 die größte Adresse, die noch beschrieben werden kann.

Bei Feldern, die nicht zu lang sind, kann man auch folgende Kurzschreibweisen für die Wertzuweisung verwenden:

entweder:

```
int[] meinIntArray = new int[5]{1, 3, 5, 7, 9};
```

oder noch kürzer:

```
int[] meinIntArray = {1, 3, 5, 7, 9};
```

Auf längere Felder greift man in der Regel über Schleifen zu. Dabei lernen wir gleich eines der Attribute von Arrays kennen, das "Length"-Attribut, über das man Abfragen kann, wieviel Elemente ein Feld hat.

**Achtung Falle:** Da die Zählung bei 0 beginnt, muss man immer aufpassen, bis zu welchem Wert man den Schleifenzähler laufen lässt.

```
int[] meinIntArray = new int[5];
// Array füllen
for (int i =0; i<meinIntArray.Length; i++)
{
 meinIntArray[i] = i;
}

// Array auslesen
for (int i =0; i<meinIntArray.Length; i++)
{
 Console.WriteLine(meinIntArray[i].ToString());
}
```

### 10.3.2 foreach-Anweisung

Da sehr häufig über Schleifen auf Arrays zugegriffen wird, hat man in C# die **foreach**-Anweisung aus VB übernommen.

Das Fragment von vorhin sieht damit so aus:

```
int[] meinIntArray = new int[5];
// Array füllen
for (int i =0; i<meinIntArray.Length; i++)
{
 meinIntArray[i] = i;
}

// Array auslesen
foreach (int i in meinIntArray)
{
 Console.WriteLine(i.ToString());
}
```

### 10.3.3 Flexibilität mit params

Beim Methodenaufruf muss man ja peinlich darauf achten, dass man die selbe Anzahl von Parametern im Aufruf verwendet, wie sie auch in der Methodendefinition vorgesehen sind.

Das Schlüsselwort **params** verhilft hier zu einer gewissen Flexibilität, denn dadurch kann man eine Methode so definieren, dass sie mit Arrays beliebiger Länge aufgerufen werden kann.

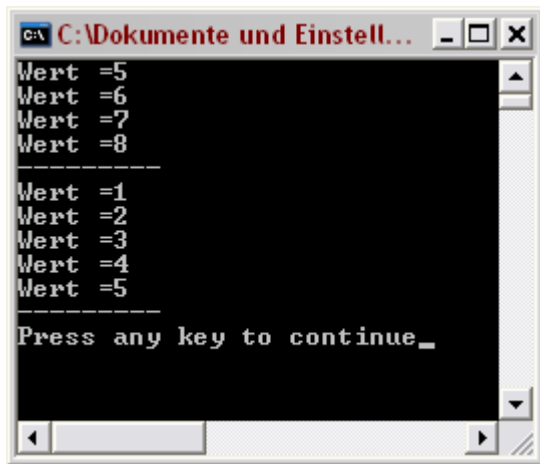
z.B.:

```
using System;
namespace params_Demo_Konsole
{
 class Class1
 {
 [STAThread]
 static void Main(string[] args)
 {
 // Dieser Konstruktor ist erforderlich, damit man Methoden dieser
 // Klasse aufrufen kann. Mit new wird eine Instanz der Class1
 // eingerichtet.
 Class1 c = new Class1();

 // Durch die Zuordnung zur Instanz c und damit zu einem Objekt
 // kann die Methode ZeigeWerte aufgerufen werden
 c.zeigeWerte(5,6,7,8);
 int[] explizitesArray = new int[5] {1,2,3,4,5};
 c.zeigeWerte(explizitesArray);
 }

 public void zeigeWerte(params int[] intWerte)
 {
 foreach (int i in intWerte)
 {
 Console.WriteLine("Wert =" + intWerte[i]);
 }
 Console.WriteLine("-----");
 }
 }
}
```

Ausgabe:



## 10.4 Mehrdimensionale Arrays

Mehrdimensionale Arrays bestehen aus **mehreren Zeilen und mehreren Spalten**. Dabei können entweder alle Zeilen die selbe Länge haben, man spricht dann von **rechteckigen Arrays**, oder jeweils eine andere Länge haben, die man **ungleichförmige Arrays** nennt.

Ein rechteckiges Array ist ein Array mit zwei oder mehr Dimensionen. Im zweidimensionalen Array entspricht

- die erste Dimension der Zeile und
- die zweite Dimension der Spalte

Syntax:

*Typ[, ] Array-Name*

z.B.:

```
int[,] meineMatrix = new int[2,3]
```

Damit ist ein Feld mit dem Namen *meineMatrix* deklariert und eingerichtet worden, das 2 Zeilen und 3 Spalten beschreibt.

Auf mehrdimensionale Felder wird in der Regel über geschachtelte Schleifen zugegriffen.

z.B.:

```
using System;
namespace MehrDimArray
{
 class Class1
 {
 [STAThread]
 static void Main(string[] args)
 {
 const int zeilen = 4;
 const int spalten = 3;

 int[,] rechteckArray = new int[zeilen, spalten];
```



## 10.6 ArrayList, ein Array-Typ mit dynamischer Obergrenze

Arrays haben die manchmal unangenehme Eigenschaft, dass man ihnen beim Einrichten eine fixe Größe zuordnen muss, die nachträglich leider nicht mehr geändert werden kann. Zum Zeitpunkt des Programmierens weiß man jedoch häufig noch nicht, wieviele Werte in einem Array abgelegt werden sollen. So ist das Array entweder unterdimensioniert, was Probleme im Programmablauf verursacht, oder durch Angstzuschläge überdimensioniert, was unnötig Speicherplatz verschwendet.

Als Lösung für dieses Problem gibt es in C# die sog. ArrayList-Klasse. ArrayList ist ein Array-Typ, dessen Länge dynamisch erst zur Laufzeit des Programmes nach oben erweitert werden kann.

ArrayList muß jedoch ganz anders behandelt werden wie ein Array, das wir bisher kennengelernt haben. ArrayList kennt einen eigenen Satz von Methoden zur Verwaltung und eigene Attribute, die die Eigenschaften beschreiben.

Ein wesentlicher Unterschied ist auch, dass ein Array bei der Einrichtung mit einem bestimmten Datentyp deklariert wird, und damit alle seine Elemente den gleichen Datentyp besitzen. Ein ArrayList jedoch hat keinen bestimmten Datentyp und kann Elemente unterschiedlichen Datentyps enthalten. Bei Zuweisungen eines ArrayList-Elementes auf Variablen oder Objekte, die mit einem bestimmten Datentyp deklariert wurden, muss deshalb eine explizite Typkonvertierung erfolgen. Lediglich implizite Typkonvertierungen akzeptiert der Compiler nicht.

Will man ArrayList verwenden, so muss mit Hilfe von `using` auf den Namespace **Systems.Collection** verwiesen werden.

z.B.:

```
using System;
using System.IO;
using System.Collections;
```

So wird ein ArrayList eingerichtet:

```
ArrayList meinDynamischesArray = new ArrayList();
```

So werden Elemente zu einem ArrayList hinzugefügt:

```
for (int i = 0; i<5; i++)
{
 meinDynamischesArray.Add(i*5);
}
```

So werden ArrayList-Elemente ausgegeben. Beachte, dass das Attribut für die höchste Array- Adresse **Count** heißt (und nicht *Length*, wie beim Array)

```
for (int i = 0; i<meinDynamischesArray.Count; i++)
{
 Console.WriteLine(meinDynamischesArray[i].ToString());
}
```

So werden Elemente eines ArrayList explizit auf Variablen mit einem bestimmten Typ zugewiesen:

```
int j = (int) meinDynamischesArray[4];
```

## 10.7 Übung 10

Nehmen Sie das Programm aus Übung 8 und entwickeln Sie es weiter, so dass es nicht mehr mit einem statischen Array arbeitet sondern mit einer dynamischen ArrayList. Dadurch kann das Programm Dateien beliebiger Länge lesen und sortieren, ohne dass die Länge des Arrays angepaßt werden muss.



## 11 Sortieren

### 11.1 Übung 6

Erstellen Sie eine allgemeine Methode, die ein eindimensionales Feld von Integer-Zahlen der Größe nach sortieren kann. Diese Methode soll beliebig lange Felder empfangen können.

Binden Sie diese Sortiermethode in eine Klasse ein, und rufen Sie sie aus der Hauptmethode auf. In der Hauptmethode definieren sie das zu sortierende Feld. Zeigen Sie das Feld vor und nach dem Sortieren am Bildschirm an.

## 12 Mit Dateien arbeiten

### 12.1 Grundlagen

Sollen Daten dauerhaft erhalten bleiben, also über das Ende eines Programmes hinaus, so müssen sie in Dateien auf einem dauerhaften Speicher (z.B. Festplatte, Flash-Memory, ...) abgespeichert werden. C# stellt eine Vielzahl von Funktionen zur Verfügung, mit denen Dateien und Ordner-Strukturen gelesen, verwaltet und geändert werden können.

Auf eine Datei kann lesend oder schreibend zugegriffen werden. Dabei gibt es zwei wesentliche Unterschiede, binäre und Text-Dateien:

- **Binäre Dateien** halten nicht nur reinen Text sondern alle mögliche Information binär abgespeichert. Viele Dateien, die von Anwendungsprogrammen erzeugt werden, sind binäre Dateien. Beispiele sind doc-Dateien aus WinWord, xls-Dateien aus Excel usw.  
Wenn man nicht genau weiß, ob es sich um eine binäre oder Text-Datei handelt, die man öffnen und evtl. verändern möchte, sollte man sicherheitshalber die Funktionen für die binäre Behandlung verwenden. Aus Zeitgründen wird aber im Rahmen dieser Vorlesung auf dieses Thema nicht näher eingegangen und es sei bei Interesse auf die Literatur verwiesen.
- **Text-Dateien** enthalten, wie ihr Name schon sagt, reine Text-Daten. Die Funktionen zum öffnen und lesen von Text-Dateien dürfen nur dann angewendet werden, wenn man sicher weiß, dass es sich um eine Text-Datei handelt.

Das Mittel, mit dem auf Dateien zugegriffen wird sind sog. "**Streams**" (Datenströme).

### 12.2 Streams

Sollen Daten aus einer Datei, durch das Netzwerk oder über das Internet bewegt werden, müssen sie in einen **Stream** (Datenstrom) verwandelt werden. Daten fließen in einem Stream als Paket hintereinander, ähnlich wie Luftblasen in einem Strohhalm.

Im weiteren sollen nur die Streams behandelt werden, die für den Zugriff auf Dateien zur Verfügung stehen. Darüber hinaus gibt es noch weitere, mit deren Hilfe man Daten über das TCP/IP-Protokoll in Netzwerken und im Internet bewegen kann.

Der Ausgangspunkt eines Streams wird als *Backing-Store* (Zusatzspeicher) bezeichnet, der die Quelle für den Stream darstellt, so wie ein See für einen Bach.

Dateien und Verzeichnisse werden im .NET-Framework, und damit auch in C#, als Klassen dargestellt, die Methoden zur Verfügung stellen, mit denen man Dateien und Verzeichnisse erzeugen, benennen, manipulieren und löschen kann.

Es gibt **ungepufferte** und **gepufferte Streams**, wodurch die Geschwindigkeit beim Lesen einer Datei günstig beeinflusst werden kann. Wir betrachten im weiteren nur ungepufferte Streams.

Darüber hinaus gibt es **synchrone und asynchrone Streams**: asynchrone Streams können Daten Bit für Bit aus einer Datei von der Festplatte holen, während das Programm etwas anderes tut. Asynchrone Streams melden, wenn sie fertig sind. Wir betrachten im weiteren nur synchrone Streams, die bewirken, dass das Programm, während der Stream fließt, wartet, bis die Stream- Behandlung angeschlossen ist.

Objekte, die in C# angelegt und abgespeichert werden sollen, müssen *serialisiert*, d.h. in Bitfolgen zerlegt werden, bevor sie über Streams bewegt werden können. Dazu stehen im Namensraum **System.IO** Klassen zur Verfügung, die entsprechende Funktionen anbieten.

So stellt die Klasse *Directory* z.B. folgende Funktionen zur Verfügung: CreateDirectory, Delete, Exist, GetDirectoryRoot, GetFiles, GetLastAccessTime, ... usw.

Die Klasse *File* bietet z.B. folgende Funktionen an: AppendText, Copy, Create, CreateText, Delete, Move, OpenRead, OpenWrite, ... usw. Die letzten beiden Befehle etwa öffnen, bzw. erzeugen einen Stream zum Lesen, bzw. Lesen und Schreiben einer Datei.

Die Stream-Klassen sind übertoll mit Methoden, die wichtigsten zur Behandlung von Text-Dateien werden wir jetzt kennenlernen.

## 12.3 Mit Textdateien arbeiten

Viele Programme können neben binären Dateien auch Text-Dateien erzeugen, z.B. WinWord mit der Option "speichern als Text". Im Falle von WinWord gehen dabei zwar alle Informationen verloren, die für die Formatierung erforderlich sind (die werden z.B. im binären .doc abgespeichert), aber die reine Text-Information bleibt erhalten. Sehr häufig haben Text-Dateien die Endung .txt und können mit einfachen Text-Editoren gelesen und verändert werden.

Wenn sicher feststeht, dass die Datei, die in C# geöffnet/geschrieben werden soll eine **reine Textdatei** ist, können die Klassen **StreamReader** und **StreamWriter** verwendet werden.

Diese Klassen bieten u.A. die Methoden **ReadLine()** und **WriteLine()** an. WriteLine haben wir bereits bei dem Console-Objekt benutzt. D.h., ohne es zu wissen, haben wir einen Stream für Text- Daten erzeugt und zur Console geschickt, die diesen dann über die Grafikkarte weitergeleitet hat, die ihn dann letztlich am Bildschirm dargestellt hat.

Wie schon bei der Anwendung von Arrays verwenden wir auch bei den Streams den **new-Operator**, um eine Stream-Instanz zu erhalten.

Um mit Streams arbeiten zu können, muss der Namespace **System.IO** benannt werden.

### 12.3.1 Text-Daten in Datei schreiben

Um Text-Daten in eine Text-Datei schreiben zu können, rufen Sie den **StreamWriter**- Konstruktor auf und übergeben den vollständigen Namen, den Sie schreiben wollen:

z.B.

```
StreamWriter schreiben = new StreamWriter(@"C:\CsharpTest\Test1.txt",
false);
```

Es wird hier also eine Instanz mit dem Namen "schreiben" angelegt. Der zweite Parameter ist das Argument *append* vom Typ boolean. Steht es auf *false*, wird die Datei neu angelegt, bzw. überschrieben, falls sie existiert. Steht es auf *true* wird die Datei neu angelegt, bzw. am Ende angehängt, falls sie bereits existiert.

#### Achtung:

Der Pfad muss vollständig vorhanden sein, der Pfad wird nicht angelegt. Wenn er noch nicht existiert, führt das zu einer Programm-Ausnahme, das Programm stürzt in diesem Fall ab. Auch müssen Sie beachten, dass Sie auf den gewünschten Pfad Lese- und Schreibrechte haben.

mit

```
schreiben.WriteLine(text);
```

Kann jetzt Textzeile für Textzeile in die Datei geschrieben werden. Vergleichen Sie dazu noch einmal den Befehl, mit dem wir schon früher Texte auf der Console ausgegeben haben:

```
Console.WriteLine(text);
```

**WriteLine()** schreibt eine Textzeile komplett raus und fügt danach eine Carriage Return ein, d.h. der "Druckwagen" steht danach wieder am Zeilen anfang. Soll auf dem Bildschirm oder in der Datei jedoch eine Text-Zeile Stück für Stück zusammengesetzt werden, steht dafür der Befehl **Write()** zur Verfügung. Der "Druckwagen" bleibt nach der Ausgabe über Write() nach dem letzten Buchstaben in der Zeile stehen und wartet auf den nächsten Write()-Befehl. Falls Sie Write() verwenden, vergessen Sie nicht am Zeilenende wenigstens einen WriteLine()-Befehl mit leerer Parameterliste abzusetzen, damit der "Druckwagen" für die nächste Zeile wieder am Anfang steht.

Hier ein vollständiges Programm, das eine Textdatei anlegt und beschreibt:

```
using System;
using System.IO;

namespace TextDateiSchreiben
{
 class Class1
 {
 [STAThread]
 static void Main(string[] args)
 {
 //Erzeugt eine Instanz dieser Klasse
 //und startet ihre erste Methode
 Class1 t = new Class1();
 t.Run();
 }

 private void Run()
 {
 // Es wird die Instanz "schreiben" des Datenstroms StreamWriter
 // angelegt,
 // der zum Beschreiben von Text-Dateien dient.
 StreamWriter schreiben = new StreamWriter(@"C:\CsharpTest\Test1.txt",
 false);

 //Text-Datei beschreiben
 schreiben.WriteLine("Meine erste Zeile in einer Text- Datei");
 schreiben.WriteLine("Meine zweite Zeile in einer Text- Datei");

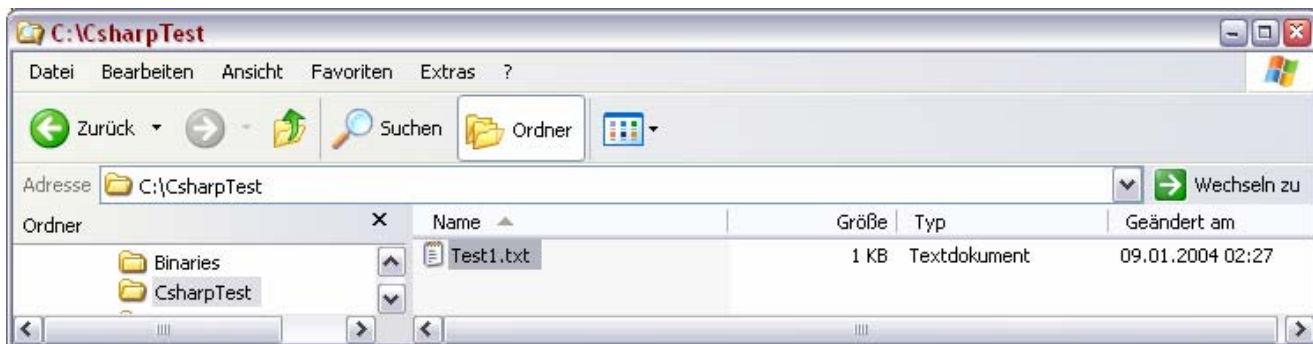
 // Nachricht für den Anwender,
 // wobei die \-Zeichen mit der \-Escape-Sequenz rausgeschrieben
 // werden müssen
 Console.WriteLine("Text-Datei C:\\CsharpTest\\Test1.txt wurde
 geschrieben");

 //Aufräumen
 schreiben.Close();
 }
 }
}
```

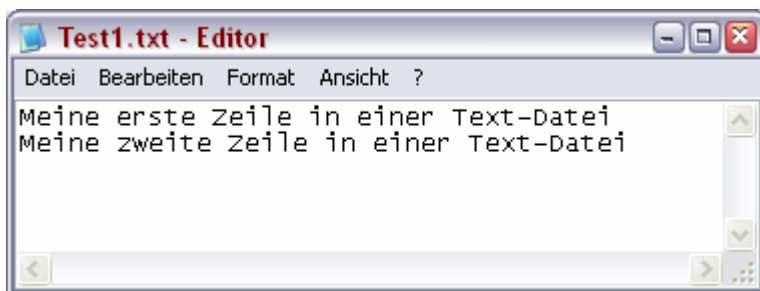
Ausgabe:



Datei wurde angelegt:



Datei kann mit einem Text-Editor geöffnet werden:



### 12.3.2 Text-Daten aus Datei lesen

Um eine **StreamReader**-Instanz zu erhalten, muss man zunächst eine Instanz der **FileInfo**- Klasse erzeugen und kann dann die Methode **OpenText()** auf diesem Objekt aufrufen. Die Instanz der **FileInfo**- Klasse wird hier "dieQuellDatei" genannt, und die Instanz des **StreamReader** wird hier "lesen" genannt.

```
FileInfo dieQuellDatei = new FileInfo(@":\CsharpTest\Test1.txt");
StreamReader lesen = dieQuellDatei.OpenText();
```

**OpenText** gibt einen **StreamReader** zurück, mit dem die Datei Zeile für Zeile über eine Schleife mit folgendem Kommando gelesen werden kann:

```
string text = lesen.ReadLine();
```

Die letzte Zeile einer Text-Datei ist leer und der **StreamReader** liefert deshalb den Wert **null**. Dieser kann als Abbruch-Kriterium für die Schleife verwendet werden.

Hier ein vollständiges Programm, dass eine Textdatei öffnet und liest:

```
using System;
using System.IO;

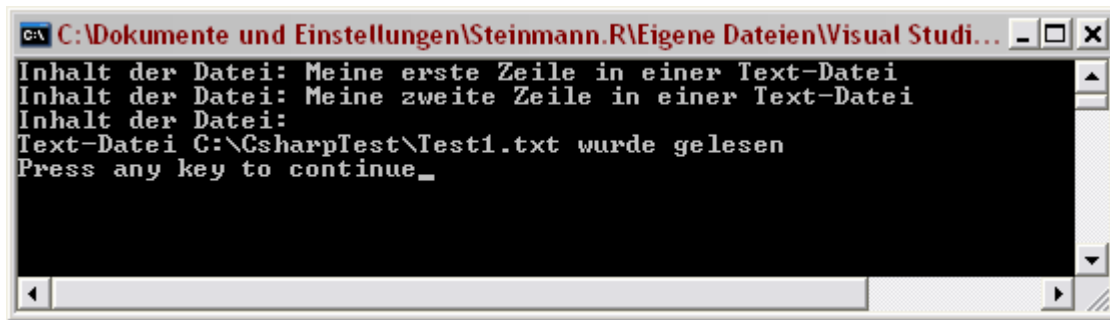
namespace TextDateiLesen
{
 class Class1
 {
 [STAThread]
 static void Main(string[] args)
 {
 //Erzeugt eine Instanz dieser Klasse
 //und startet ihre erste Methode
 Class1 t = new Class1();
 t.Run();
 }

 private void Run()
 {
 // Es wird die Instanz "dieQuellDatei" der Klasse FileInfo angelegt
 FileInfo dieQuellDatei = new FileInfo(@"C:\CsharpTest\Test1.txt");
 // Jetzt steht der Stream zur Verfügung, der "lesen" genannt wird
 StreamReader lesen = dieQuellDatei.OpenText();
 //Text-Datei lesen und den gelesenen Text am Bildschirm darstellen
 string text;
 do
 {
 text = lesen.ReadLine();
 Console.WriteLine("Inhalt der Datei: " + text);
 }while (text != null);

 // Nachricht für den Anwender,
 // wobei die \-Zeichen mit der \-Escape-Sequenz rausgeschrieben
 // werden müssen
 Console.WriteLine("Text-Datei C:\\CsharpTest\\Test1.txt wurde
 gelesen");

 //Aufräumen
 lesen.Close();
 }
 }
}
```

Ausgabe:



```
C:\Dokumente und Einstellungen\Steinmann.R\Eigene Dateien\Visual Studi...
Inhalt der Datei: Meine erste Zeile in einer Text-Datei
Inhalt der Datei: Meine zweite Zeile in einer Text-Datei
Inhalt der Datei:
Text-Datei C:\CsharpTest\Test1.txt wurde gelesen
Press any key to continue_
```

## 12.4 Übung 7

Verändern Sie das Programm aus Übung 6 so, dass es seine Ergebnisse nicht nur am Bildschirm, sondern auch in eine Text-Datei ausgibt.

## 12.5 Übung 8

Verändern Sie das Programm aus Übung 7 (bzw. Übung 6) so, dass es die Ausgangsdaten, die sortiert werden sollen, aus einer Text-Datei liest, die Sie vorher mit einem Text-Editor erstellt haben.

## 12.6 Übung 9

Erstellen Sie mit einem Text-Editor eine Text-Datei. Erstellen Sie dann ein Programm, das dieses Datei öffnet, liest und eine Kopie davon erzeugt. Überprüfen Sie mit einem Text-Editor, ob die Kopie Fehler-frei ist.

## 13 2D-Grafik

### 13.1 Einführung

Interaktionen zwischen Programm und Benutzer werden über sogenannte Ereignisse gesteuert. Aber auch das Zeichnen von Grafik wird über ein Ereignis verwaltet. Der Zugriff auf Ereignisse erfolgt über sogenannte Handler (oder auf Englisch handles).

Das **Paint-Ereignis** ist für die Steuerung von Grafik vorgesehen. Der Zugriff auf den Handler eines Paint-Ereignisses erfolgt über folgende Methode:

```
protected override void OnPaint(PaintEventArgs pea)
{
 //Befehle der Zeichenroutine
}
```

Damit der Compiler damit etwas anfangen kann, muss folgendes using-Kommando vorhanden sein, das der Forms- Editor ohnehin schon automatisch anlegt:

```
using System.Drawing;
```

Der 0-Punkt der Koordinatensystems ist immer der linke obere Eckpunkt des Ausgabefensters, wobei die Positive x- Achse nach rechts und die positive y-Achse nach unten weist.

### 13.2 Pen

Gezeichnet wird mit einem Stift, der verschiedene Eigenschaften wie Farbe und Dicke (gemessen in Pixel) haben kann. Bevor ein Pen (Stift) verwendet werden kann, muss man ihn einrichten. Pen ist eine Klasse, von der man eine Instanz bilden muss.

z.B.:

```
protected override void OnPaint(PaintEventArgs pea)
{
 Pen stift = new Pen(Color.Red);

 ...

 stift.Color = Color.Green;
 stift.Width = 3;

 ...
}
```

### 13.3 Zugriff auf Grafik-Befehle

Der Zugriff auf einen Grafikbefehle sieht so aus:

```
protected override void OnPaint(PaintEventArgs pea)
{
 Graphics grfx = pea.Graphics;
 Pen stift = new Pen(Color.Red);
 grfx.DrawLine(stift, 0,0, 10, 10);

 ...
}
```

Die Zuweisung der Struktur *pea.Graphics* auf z.B. *grfx* ermöglicht eine verkürzte Schreibweise beim Aufruf der Grafikbefehle. Andernfalls müsste man vor jedem Grafikbefehl *pea.Graphics* setzen, damit der Compiler ihn finden kann.



## 13.4 Auswahl aus 2D Grafikbefehlen

Neben den unten aufgeführten Grafikbefehlen gibt es noch eine Vielzahl weiterer, für die auf die Literatur verwiesen sei.

### Clear

Mit Clear kann das Grafikfenster mit einer einheitlichen Farbe gefüllt werden. Alle bis dahin gezeichneten Grafiken werden dadurch überblendet und erscheinen dem Anwender dadurch als gelöscht.

```
grfx.Clear(Color.BlanchedAlmond);
```

### DrawString

Text in Grafikfenster ausgeben:

```
DrawString("Test", Font, Brushes.DarkRed, 10,10);
```

### DrawLine

Linie zeichnen zwischen zwei Punkten. Dabei können die Koordinaten entweder als Integer- Werte (=Pixel) oder als Float- Werte übergeben werden. Über pen wird die Instanz eines Stiftes mit dem Namen pen (frei gewählt) übergeben, und damit in dieser Struktur alle seine Eigenschaften.

```
DrawLine(pen, 0,0, ClientSize.Width -1, ClientSize.Height - 1);
```

### DrawArc (=Ellipse)

Die Ellipse erwartet als Übergabeparameter eine Instanz der Klasse Rectangle, die gebildet werden muss, bevor man DrawArc aufrufen kann. Die Ellipse orientiert sich mit ihren Achsen an dem umschreibenden Rechteck. Außerdem wird ein Anfangswinkel (gemessen im Uhrzeigersinn ab 3:00 Uhr) und ein Deltawinkel übergeben. Negative Gradzahlen weisen in den Gegen-Uhrzeigersinn.

```
Rectangle rect = new Rectangle(60, 60, 200, 200);
grfx.DrawArc(pen, rect, 15, -180);
```

### AntiAliasing

AntiAliasing ermöglicht, den Treppen-Effekt von Linien am Bildschirm abzumildern. Dabei wird die Linie am Rand etwas unschärfer gemacht, was manche Betrachter als störende Unschärfe empfinden.

Mögliche AntiAliasingwerte:

- HighQuality und AntiAlias für on
- u.A. None, Standard, HighSpeed für off

Außerdem muss der Namespace *using System.Drawing.Drawing2D;* definiert sein!

```
grfx.SmoothingMode = SmoothingMode.HighQuality;
```

### 13.5 Funktionsgrafen zeichnen

Grafen von Funktionen können als Näherung gezeichnet werden, indem man in kleinen Delta-Schritten die Funktion berechnet und man dann immer den vorgehenden Wert mit dem neu errechneten verbindet.

z.B.:

```
float d;
float dVorher = 0.0F;
float yVorher = 0.0F;
for (int i=0; i<=100; i++)
{
 d=i/10;
 grfx.DrawLine(pen, dVorher*10, yVorher*2, d*10, d*d*2);
 dVorher = d;
 yVorher = d*d;
}
```

### 13.6 Beispiel-Programm

Das folgende Programm ist eine Sammlung der oben besprochenen Grafik-Befehle, und zeigt wie sie aufgerufen werden können:

```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace Zeichnen
{
 public class Form1 : System.Windows.Forms.Form
 {
 private System.ComponentModel.Container components = null;

 public Form1()
 {
 //
 // Erforderlich für die Windows Form-Designerunterstützung
 //
 InitializeComponent();

 //
 // TODO: Fügen Sie den Konstruktorcode nach dem Aufruf von
 InitializeComponent hinzu
 }
 }
}
```

```

 //
}

/// <summary>
/// Die verwendeten Ressourcen bereinigen.
/// </summary>
protected override void Dispose(bool disposing)
{
 if(disposing)
 {
 if (components != null)
 {
 components.Dispose();
 }
 }
 base.Dispose(disposing);
}

#region Windows Form Designer generated code
/// <summary>
/// Erforderliche Methode für die Designerunterstützung.
/// Der Inhalt der Methode darf nicht mit dem Code-Editor geändert
werden.
/// </summary>
private void InitializeComponent()
{
 //
 // Form1
 //
 this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
 this.ClientSize = new System.Drawing.Size(292, 271);
 this.Name = "Form1";
 this.Text = "Form1";

}
#endregion

/// <summary>
/// Der Haupteinstiegspunkt für die Anwendung.
/// </summary>
[STAThread]

```

```

static void Main()
{
 Application.Run(new Form1());
}

protected override void OnPaint(PaintEventArgs pea)
{
 Graphics grfx = pea.Graphics;
 Pen pen = new Pen(Color.Red);

 grfx.Clear(Color.BlanchedAlmond);
 grfx.DrawString("Test1", Font, Brushes.DarkRed, 10,10);
 grfx.DrawLine(pen, 0,0, ClientSize.Width -1, ClientSize.Height - 1);
 Rectangle rect = new Rectangle(60, 60, 200, 200);
 grfx.DrawArc(pen, rect, 15, -180);
 //Mögliche Anti-Aliasingwerte:
 //HighQuality und AntiAlias für on
 //u.A. None, Standard, HighSpeed für off
 //using System.Drawing.Drawing2D; muss definiert sein!
 grfx.SmoothingMode = SmoothingMode.HighQuality;
 pen.Color = Color.Blue;
 grfx.DrawString("Test2", Font, Brushes.DarkOrchid, 10,100);
 grfx.DrawLine(pen, 0,ClientSize.Height - 1, ClientSize.Width -1, 0);
 pen.Width = 4;
 grfx.DrawRectangle(pen, 20, 20, 50, 50);
 grfx.DrawArc(pen, rect, 15, 180);

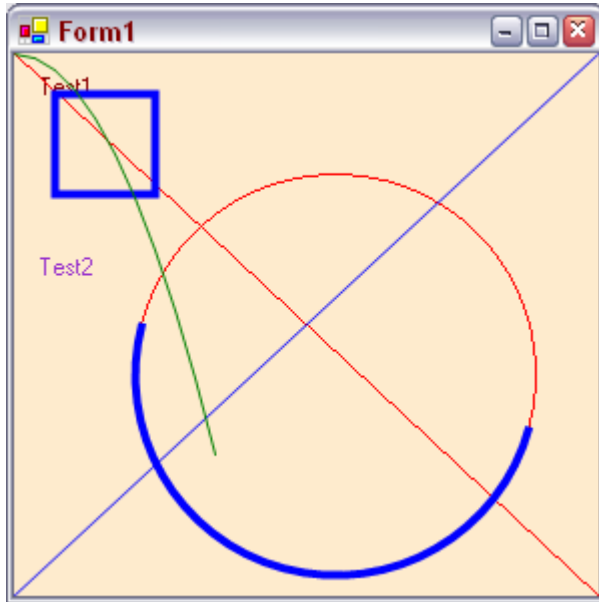
 pen.Color = Color.Green;
 pen.Width = 0;
 float d;
 float dVorher = 0.0F;
 float yVorher = 0.0F;
 for (int i=0; i<=100; i++)
 {
 d=i/10;
 grfx.DrawLine(pen, dVorher*10, yVorher*2, d*10, d*d*2);
 dVorher = d;
 yVorher = d*d;
 }
}

```

```
}
```

```
}
```

```
}
```



### 13.7 Übung 11

Erstellen Sie ein Programm, das den Funktionsgrafen für die Funktion

$$f(x) = \text{SQRT}(x+a) / (x+b)$$

darstellt.

Die Eingangsgrößen a und b, sowie der Wertebereich von x und die Teilung der Berechnungsschritte sollen aus einer Datei eingelesen werden, die vorher mit Hilfe eines Text-Editors beschrieben wurde.

Die Koordinatenachsen sollen schwarz, die Polstelle durch eine vertikale grüne Linie und der Graph in rot dargestellt werden.

## 14 Verschiedene Themen

### 14.1 Escape-Sequenzen

Escape-Sequenzen werden für die Ausgabe von Texten benötigt. Ihre Bezeichnungen stammen noch aus der Zeit, als Ausgaben vorwiegend auf Druckern erfolgten. Eine Escape-Sequenz entspricht dem Typ `char` und wird intern in Unicode Zeichen dargestellt. In einem String werden Escape-Sequenzen mit einfachen Anführungszeichen eingerahmt.

Gebräuchliche Escape-Sequenzen sind:

| Zeichen         | Bedeutung                      |
|-----------------|--------------------------------|
| <code>\'</code> | Einfaches Anführungszeichen    |
| <code>\"</code> | Doppeltes Anführungszeichen    |
| <code>\\</code> | Backslash                      |
| <code>\0</code> | Null                           |
| <code>\a</code> | Alert                          |
| <code>\b</code> | Rücktaste                      |
| <code>\f</code> | Seitenvorschub                 |
| <code>\n</code> | Newline                        |
| <code>\r</code> | Carriage Return (= neue Zeile) |
| <code>\t</code> | Horizontaler Tabulator         |
| <code>\v</code> | Vertikaler Tabulator           |

In der Textausgabe werden diese Escape-Sequenzen heute sinngemäß verwendet, auch wenn die Ausgabe nicht mehr unbedingt auf einem Drucker, sondern etwa auf dem Bildschirm, in einen String oder in eine Datei erfolgt. So versteht man heute unter "Carriage Return" (wörtlich übersetzt: "Druckwagen zurück") eine "neue Zeile".

z.B.:

```
System.Console.WriteLine("Spalte A" + '\t' + "Spalte B");
```

### 14.2 Mathematik-Klasse

`Math` ist eine sog. versiegelte Klasse, die Naturkonstanten und Methoden für viele trigonometrische, logarithmische und andere mathematische Operationen enthält. Der Aufruf einer Methode erfolgt über die Klasse `Math`.

z.B. Wurzel-Funktion:

```
double a = 4.0;
double z = Math.Sqrt(a);
```

Jede Methode der `Mathematik`-Klasse liefert einen Wert von einem bestimmten Typ zurück, häufig `double`. Wichtig ist auch, dass man darauf achtet, den richtigen Typ als Parameter zu übergeben. Einige Methoden liegen mehrfach, d.h. mit überladenen Bezeichnern vor, so dass man mit dem selben Bezeichner unterschiedliche Typen bearbeiten kann.

`Math.E` liefert 2.7182818..... als Typ `double`

`Math.PI` liefert 3.141592653.... als Typ `double`

In der folgenden Tabelle finden sie eine Auswahl:

| Rückgabe-Typ | Methode                                                     | Parameter-Typ     |
|--------------|-------------------------------------------------------------|-------------------|
| decimal      | Abs(a)                                                      | decimal a         |
| double       | Abs(a)                                                      | double a          |
| int          | Abs(i)                                                      | int i             |
|              | usw.<br>entspr.:<br>Round(a),<br>Min(a,b), Max(a,b)         |                   |
| double       | Sin(a)                                                      | double a          |
|              | entspr.: Cos, Tan,<br>Sinh, Cosh, Tanh,<br>Asin, Acos, Atan |                   |
| double       | Exp(a)                                                      | double a          |
| double       | Pow(x,y)                                                    | double x, y       |
| double       | Log(a)                                                      | double a          |
| double       | Log(a, newBase)                                             | double a, newBase |
| double       | Log10(a)                                                    | double a          |
| double       | Sqrt(a)                                                     | double a          |
| double       | IEEERemainder(x, y)                                         | double x, y       |
| double       | Ceiling(a)                                                  | double a          |
| double       | Floor(a)                                                    | double a          |

Ceiling (=Decke) ist eine sehr große Zahl, Floor (=Boden) ist eine sehr kleine Zahl. Diese Methode braucht man z.B., wenn man in einer Iteration einen Wertebereich eingrenzen möchte, und dabei von einem jeweils sehr großen und sehr kleinen Wert starten möchte.





## 15 Vorrangstufen von Operatoren

Die folgende Tabelle gibt einen Überblick über Vorrangstufen und Assoziativität von Operatoren. Die Operatoren sind nach ihrer Vorrangstufe sortiert, wobei oben die Operatoren mit der höchsten Vorrangstufe stehen, unten die mit der niedrigsten:

| Kategorie     | Operatoren |                              | Assoziativität  |
|---------------|------------|------------------------------|-----------------|
| primär        | ( )        | überschreibt Priorität       | links -> rechts |
| unär          | !          | logisches NICHT              | links -> rechts |
| multiplikativ | *          | Multiplikation               | links -> rechts |
|               | /          | Division                     |                 |
|               | %          | Modulo                       |                 |
| additiv       | +          | Addition                     | links -> rechts |
|               | -          | Subtraktion                  |                 |
| relational    | <          | kleiner als                  | links -> rechts |
|               | <=         | kleiner gleich als           |                 |
|               | >          | größer als                   |                 |
|               | >=         | größer gleich als            |                 |
| Gleichheit    | ==         | gleich                       | links -> rechts |
|               | !=         | ungleich                     |                 |
| boolsche      | &&         | logisches UND                | links -> rechts |
|               |            | logisches ODER               |                 |
| Zuweisung     | =          | Wertzuweisung                | links <- rechts |
|               | *=         | Verbundzuweisungs-Operatoren |                 |
|               | /=         |                              |                 |
|               | %=         |                              |                 |
|               | +=         |                              |                 |
|               | -=         |                              |                 |

## 16 Debugger

Das Wort "bug" ist die englische Bezeichnung von Käfer oder Ungeziefer. Wenn Programmierer in ihren Programmen Fehler suchen, sagen sie, dass sie ihr Programm "debuggen", man könnte übersetzen "entlausen".

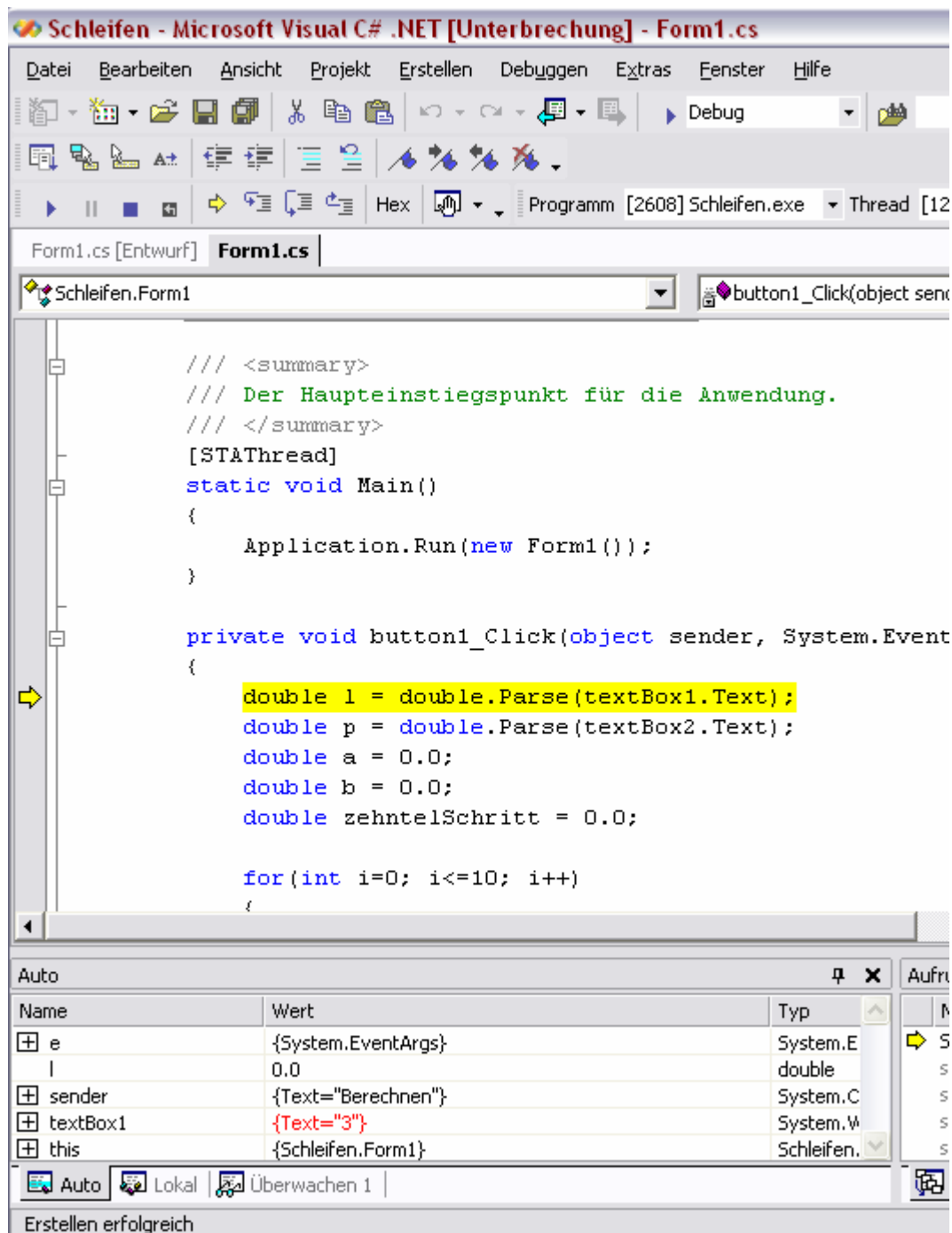
Der Compiler deckt syntaktische Fehler auf, nicht jedoch logische Ablauffehler. Die bemerkt man erst, wenn das Programm nicht so funktioniert, wie man sich das eigentlich vorstellt. Mitunter kann die Suche nach den Gründen logischer Programmfehler oder gar Programmabstürzen sehr aufwändig sein.

Moderne Programmierumgebungen liefern deshalb Werkzeuge zur Fehlersuche mit, sog. Debugger. Damit kann man:

- Das Programm Zeile für Zeile, also Schritt für Schritt ablaufen lassen.
- Nach der Ausführung jeder einzelnen Zeile kann man sich den Inhalt der Variablen ansehen und überprüfen, ob der Wert richtig ist.
- Außerdem kann man durch die schrittweise Ausführung entdecken, ab wann und unter welchen Bedingungen ein Programm falsch läuft.

Vorgehen:

- Verkleinern Sie das Fenster mit der Entwicklungsumgebung, so dass Sie einen Teil des Desktops sehen
- Klicken Sie den Textcursor an die Stelle des Programmcodes, ab der Sie den Programmablauf beobachten wollen
- Öffnen Sie an dieser Stelle mit der rechten Maustaste das Kontext-Menü
- Wählen Sie: Ausführen bis Cursor
- Ihr Programm wird geöffnet und Ihr Form oder das Konsole Fenster wird geöffnet.
- Legen Sie dieses Fenster neben das Fenster Ihrer Entwicklungsumgebung
- Machen Sie in Ihrem Programm die erforderlichen Eingaben, damit es startet
- Das Programm stoppt an der oben festgelegten Stelle



- Eine zusätzliche Menüleiste wurde eingeblendet, über die der Debugger gesteuert werden kann (z.B. Einzelschritt, Prozedur-Schritt, Haltepunkte setzen usw.)
- Im unteren Fenster kann man beobachten, welche Werte den Variablen zugewiesen wurde.
- Gehen Sie nun Schritt für Schritt durch Ihr Programm und beobachten Sie den Ablauf, sowie den Inhalt der Variablen.

## 17 Reihenfolge des Stoffes

- Hintergrundinformationen
- Begleit-Literatur
- Erstes C#-Programm
- Erstes Windows-Programm
- C#-Bausteine
  - C#-Syntax
  - Schlüsselworte
  - Variablen
  - Arithmetische Operatoren
- Übung 1
- Programm-Steuerung
  - if
  - switch
  - break
- Methoden
- Übung 2
- Auswahl-Anweisungen
  - Boolsche Variable
  - Boolsche Operatoren
- Vorrangstufen von Operatoren
- Übung 3
- C#-Bausteine
  - Arithmetische Operatoren
    - Verbundzuweisungen
    - Inkrement-/ Dekrement-Operatoren
- Programmsteuerung
  - continue
  - Programm-Schleifen
- Debugger
- Übung 4
- Typ Konvertierung: Ergänzung
- Escape-Sequenzen
- Mathematik-Bibliothek
- Stack und Heap
- Arrays
- Übung 5
- Sortieren
- Übung 6

- Mit Dateien arbeiten
- Übung 7
- Übung 8
- Übung 9
- ArrayList
- Übung 10
- Grafikbefehle
- Übung 11

## 18 Musterlösungen

### 18.1 Übung 4

Das Ergebnis mit dem Form-Editor könnte so aussehen:

Hier können Sie das lauffähige Programm runterladen und ausführen:

<http://www.bau.fhm.edu/steinmann/Apps/Schleifen.exe>

Der dazugehörige Quellcode könnte so aussehen:

```
// ++++++
// Dieser Code wurde dem Form entsprechend
// automatisch generiert
// ++++++

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace Schleifen
{
 /// <summary>
 /// Zusammendfassende Beschreibung für Form1.
 /// </summary>
 public class Form1 : System.Windows.Forms.Form
 {
```

```

private System.Windows.Forms.TextBox textBox1;
private System.Windows.Forms.TextBox textBox2;
private System.Windows.Forms.Button button1;
private System.Windows.Forms.ListBox Ergebnisse;
private System.Windows.Forms.Label label1;
private System.Windows.Forms.Label label2;
private System.Windows.Forms.Label label3;
private System.Windows.Forms.PictureBox pictureBox1;
private System.Windows.Forms.Button button2;
/// <summary>
/// Erforderliche Designervariable.
/// </summary>
private System.ComponentModel.Container components = null;

public Form1()
{
 //
 // Erforderlich für die Windows Form- Designerunterstützung
 //
 InitializeComponent();

 //
 // TODO: Fügen Sie den Konstruktorcode nach dem Aufruf von
InitializeComponent hinzu
 //
}

/// <summary>
/// Die verwendeten Ressourcen bereinigen.
/// </summary>
protected override void Dispose(bool disposing)
{
 if(disposing)
 {
 if (components != null)
 {
 components.Dispose();
 }
 }
 base.Dispose(disposing);
}

```

```

#region Windows Form Designer generated code
/// <summary>
/// Erforderliche Methode für die Designerunterstützung.
/// Der Inhalt der Methode darf nicht mit dem Code-Editor geändert
werden.
/// </summary>
private void InitializeComponent()
{
 System.Resources.ResourceManager resources = new
System.Resources.ResourceManager(typeof(Form1));

 this.textBox1 = new System.Windows.Forms.TextBox();
 this.textBox2 = new System.Windows.Forms.TextBox();
 this.button1 = new System.Windows.Forms.Button();
 this.Ergebnisse = new System.Windows.Forms.ListBox();
 this.label1 = new System.Windows.Forms.Label();
 this.label2 = new System.Windows.Forms.Label();
 this.label3 = new System.Windows.Forms.Label();
 this.pictureBox1 = new System.Windows.Forms.PictureBox();
 this.button2 = new System.Windows.Forms.Button();
 this.SuspendLayout();

 //
 // textBox1
 //
 this.textBox1.Location = new System.Drawing.Point(8, 192);
 this.textBox1.Name = "textBox1";
 this.textBox1.Size = new System.Drawing.Size(88, 20);
 this.textBox1.TabIndex = 0;
 this.textBox1.Text = "";

 //
 // textBox2
 //
 this.textBox2.Location = new System.Drawing.Point(112, 192);
 this.textBox2.Name = "textBox2";
 this.textBox2.Size = new System.Drawing.Size(88, 20);
 this.textBox2.TabIndex = 0;
 this.textBox2.Text = "";

 //
 // button1
 //
 this.button1.Location = new System.Drawing.Point(8, 224);
 this.button1.Name = "button1";
 this.button1.Size = new System.Drawing.Size(136, 24);

```



```

this.button1.TabIndex = 2;
this.button1.Text = "Berechnen";
this.button1.Click += new System.EventHandler(this.button1_Click);
//
// Ergebnisse
//
this.Ergebnisse.Location = new System.Drawing.Point(216, 32);
this.Ergebnisse.Name = "Ergebnisse";
this.Ergebnisse.Size = new System.Drawing.Size(232, 212);
this.Ergebnisse.TabIndex = 4;
//
// label1
//
this.label1.Location = new System.Drawing.Point(8, 176);
this.label1.Name = "label1";
this.label1.Size = new System.Drawing.Size(88, 16);
this.label1.TabIndex = 5;
this.label1.Text = "Einzellast in kN";
//
// label2
//
this.label2.Location = new System.Drawing.Point(112, 176);
this.label2.Name = "label2";
this.label2.Size = new System.Drawing.Size(96, 16);
this.label2.TabIndex = 5;
this.label2.Text = "Trägerlänge in m";
//
// label3
//
this.label3.Font = new System.Drawing.Font("Microsoft Sans Serif",
12F, System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point,
((System.Byte)(0)));
this.label3.Location = new System.Drawing.Point(8, 0);
this.label3.Name = "label3";
this.label3.Size = new System.Drawing.Size(440, 32);
this.label3.TabIndex = 6;
this.label3.Text = "Auflagerkräfte für wandernde Einzellast auf
Einfeldträger";
this.label3.TextAlign = System.Drawing.ContentAlignment.MiddleCenter;
//
// pictureBox1
//

```

```

 this.pictureBox1.Image =
((System.Drawing.Bitmap)(resources.GetObject("pictureBox1.Image")));
 this.pictureBox1.Location = new System.Drawing.Point(8, 32);
 this.pictureBox1.Name = "pictureBox1";
 this.pictureBox1.Size = new System.Drawing.Size(192, 136);
 this.pictureBox1.TabIndex = 7;
 this.pictureBox1.TabStop = false;
 //
 // button2
 //
 this.button2.Location = new System.Drawing.Point(160, 224);
 this.button2.Name = "button2";
 this.button2.Size = new System.Drawing.Size(40, 24);
 this.button2.TabIndex = 8;
 this.button2.Text = "Ende";
 this.button2.Click += new System.EventHandler(this.button2_Click);
 //
 // Form1
 //
 this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
 this.ClientSize = new System.Drawing.Size(464, 259);
 this.Controls.AddRange(new System.Windows.Forms.Control[] {
 this.button2,
 this.pictureBox1,
 this.label3,
 this.label1,
 this.Ergebnisse,
 this.button1,
 this.textBox1,
 this.textBox2,
 this.label2});

 this.Name = "Form1";
 this.Text = "Beispiel für \"for- Schleife\"";
 this.ResumeLayout(false);

 }
#endregion

/// <summary>
/// Der Haupteinstiegspunkt für die Anwendung.
/// </summary>

[STAThread]

```

```

static void Main()
{
 Application.Run(new Form1());
}

// ++++++
// Ab hier beginnt der Quellcode
// der eigenen Applikation
// ++++++

private void button1_Click(object sender, System.EventArgs e)
{
 // Dieser Code wurde absichtlich ausführlich geschrieben,
 // um die einzelnen Komponenten zu veranschaulichen.
 // Man könnten dieses Programm wesentlich kompakter Schreiben,
 // indem man die Formeln ineinander einsetzt, allerdings wird
 // der Programmcode dann auch unleserlich.
 double fP = double.Parse(textBox1.Text);
 double l = double.Parse(textBox2.Text);
 double a = 0.0;
 double b = 0.0;
 double fA = 0.0;
 double fB = 0.0;
 // ListBox freimachen, wichtig für wiederholte Berechnungen
 Ergebnisse.Items.Clear();
 // Kopfzeile in ListBox
 // \t ist die Escape-Sequenz für einen Tabulator
 Ergebnisse.Items.Add("Abstand:" + '\t' + " Auflagerkräfte:");

 for(int i=0; i<=10; i++)
 {
 // a wird in jedem Durchlauf um 1/10*1 vergrößert.
 // Die int-Variable i wird in der nächsten Formel
 // implizit in eine double- Variable umgewandelt.
 // ausführlich könnte man auch schreiben:
 // double z = i;
 // a = 0.1*z * 1;
 // alternativ könnte man auch eine explizite
 // Konvertierung programmieren:
 // a = 0.1*(double) i * 1;
 a = 0.1*i * 1;
 b = 1-a;
 }
}

```

```

fA = fP * b/l;
fB = fP * a/l;

// Ergebnisse in Listbox schreiben
Ergebnisse.Items.Add("a= " + a + '\t' + " A= " + fA);
Ergebnisse.Items.Add(" " + '\t' + " B= " + fB);
Ergebnisse.Items.Add(" ");
}

// Schluss-Strich nach Berechnung und Ausgabe aller Ergebnisse
Ergebnisse.Items.Add("----- Ende ----- ");
}

private void button2_Click(object sender, System.EventArgs e)
{
 //Diese Funktion beendet das Programm und
 // kann für einen "Ende"- Button eingebaut werden
 Application.Exit();
}
}
}

```

## 18.2 Übung 5

Dieses Programm wurde als Console-Anwendung programmiert:

```
using System;
namespace QuadratGleichg
{
 /// <summary>
 /// Zusammendfassende Beschreibung für Class1.
 /// </summary>
 class Class1
 {
 /// <summary>
 /// Der Haupteinstiegspunkt für die Anwendung.
 /// </summary>
 [STAThread]
 static void Main(string[] args)
 {
 // Dieser Konstruktor ist erforderlich, damit man Methoden
 // Klasse aufrufen kann. Mit new wird eine Instanz q der
 // eingerichtet.
 Class1 q = new Class1();
 // Durch die Zuordnung zur Instanz c und damit zu einem
 // kann weiter unten die Methode loesQuadratGleichg
 // aufgerufen werden

 double a = 2.0;
 double b = 23.0;
 double c = 7.0;

 const int loesungen = 2;
 double[] loesungsArray = new double[loesungen];

 Console.WriteLine("Lösung der Gleichung:");
 Console.WriteLine(a + " * x**2 + " + b + " * x + " + c + "=
0");

 if (q.loeseQuadratGleichg(a, b, c, loesungsArray))
 {
 Console.WriteLine("Reelle Lösung:");
 Console.WriteLine("x1= " + loesungsArray[0]);
 Console.WriteLine("x2= " + loesungsArray[1]);
 }
 }
 }
}
```

```

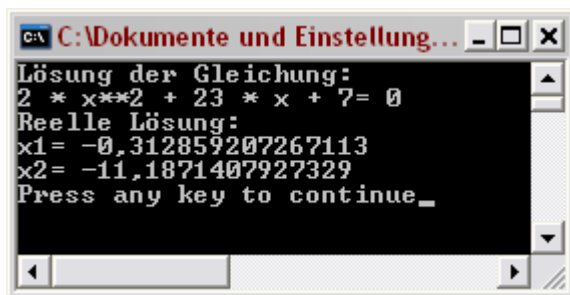
 }
 else
 Console.WriteLine("Komplexe Lösung");
}

public bool loeseQuadratGleichg(double a, double b, double c,
double[] loesungsArray)
{
 double diskriminante = 0;

 diskriminante = Math.Pow(b,2) - (4*a*c);
 if (diskriminante < 0)
 return false;
 else
 {
 loesungsArray [0] = (-b + Math.Sqrt(diskriminante)) / (2*a);
 loesungsArray [1] = (-b - Math.Sqrt(diskriminante)) / (2*a);
 return true;
 }
}
}
}

```

Anzeige:



The screenshot shows a Windows command prompt window with the title bar "C:\Dokumente und Einstellung...". The text inside the window is as follows:

```

Lösung der Gleichung:
2 * x**2 + 23 * x + 7 = 0
Reelle Lösung:
x1 = -0,312859207267113
x2 = -11,1871407927329
Press any key to continue_

```

## 18.3 Übung 6

Das Hauptprogramm "Main" und die Methode "zeigeWerte" wurden hier nicht als ein Windows- Programm, sondern als Console-Programm gestaltet, also ein Programm, das im System- Fenster läuft. Die Methode "bubble", die das Sortieren des Feldes übernimmt, ist hingegen so neutral gestaltet, dass man sie, wie hier, in einem Console-Programm verwenden kann, oder aber auch in einem Windows-Programm. Dies wurde erreicht, indem die Methode "bubble" von allen Umgebungs- spezifischen Befehlen, wie Ein-/Ausgabe freigehalten wurde.

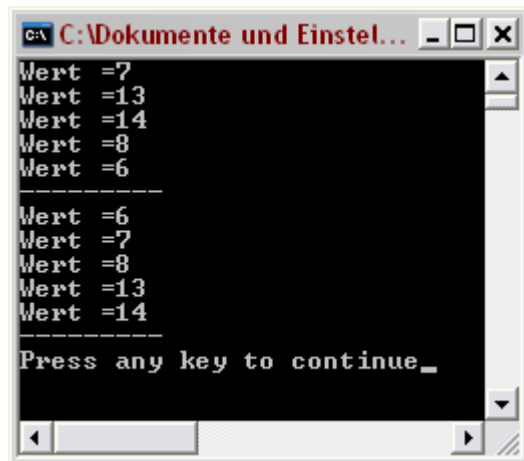
```
using System;
namespace BubbleSort
{
 class Class1
 {
 [STAThread]
 static void Main(string[] args)
 {
 Class1 c = new Class1();
 int[] explizitesArray = new int[5] {7,13,14,8,6};
 c.zeigeWerte(explizitesArray);
 c.bubble(explizitesArray);
 c.zeigeWerte(explizitesArray);
 }

 public void zeigeWerte(params int[] intWerte)
 {
 foreach (int i in intWerte)
 {
 Console.WriteLine("Wert =" + i);
 }
 Console.WriteLine("-----");
 }

 public void bubble(params int[] intArray)
 {
 for (int i = 0; i < intArray.Length; i++)
 {
 for (int j = intArray.Length-1; j > i; j--)
 {
 if(intArray[j] <= intArray[j-1])
 {
 int temp = intArray[j];
 intArray[j] = intArray[j-1];
 intArray[j-1] = temp;
 }
 }
 }
 }
 }
}
```

```
 }
 }
 }
 }
}
```

Ausgabe:





## 18.4 Übung 7

Das Programm aus der Übung 6 wurde durch die Methode "schreibeWerte" erweitert, in der die sortierten Zahlenwerte in die Datei "SortOut.txt" geschrieben werden:

```
using System;
using System.IO;
namespace BubbleErgebnisInFile
{
 class Class1
 {
 [STAThread]
 static void Main(string[] args)
 {
 Class1 c = new Class1();
 int[] explizitesArray = new int[5] {7,13,14,8,6};
 c.zeigeWerte(explizitesArray);
 c.bubble(explizitesArray);
 c.zeigeWerte(explizitesArray);
 c.schreibeWerte(explizitesArray);
 }
 public void zeigeWerte(params int[] intWerte)
 {
 foreach(int i in intWerte)
 {
 Console.WriteLine("Wert =" + i);
 }
 Console.WriteLine("-----");
 }
 private void schreibeWerte(params int[] intWerte)
 {
 // Es wird die Instanz schreiben des Datenstroms
StreamWriter angelegt,
 // der zum Beschreiben von Text-Dateien dient.
 StreamWriter schreiben = new
StreamWriter(@"C:\CsharpTest\SortOut.txt",false);

 //Text-Datei beschreiben
 foreach(int i in intWerte)
 {
 schreiben.WriteLine(i);
 }
 // Nachricht für den Anwender,
```

```

 // wobei die \-Zeichen mit der \-Escape-Sequenz
 rausgeschrieben werden müssen
 Console.WriteLine("Die sortierte Datei
C:\\CsharpTest\\SortOut.txt wurde geschrieben");
 //Aufräumen
 schreiben.Close();
 }
 public void bubble(params int[] intArray)
 {
 for(int i = 0; i < intArray.Length; i++)
 {
 for(int j = intArray.Length-1; j > i; j--)
 {
 if(intArray[j] <= intArray[j-1])
 {
 int temp = intArray[j];
 intArray[j] = intArray[j-1];
 intArray[j-1] = temp;
 }
 }
 }
 }
}

```

Ausgabe aus Programm:

```

C:\Dokumente und Einstellungen\Steinmann.R\Eigene Dateien\Visual Studio-P...
Wert =7
Wert =13
Wert =14
Wert =8
Wert =6

Wert =6
Wert =7
Wert =8
Wert =13
Wert =14

Die sortierte Datei C:\CsharpTest\SortOut.txt wurde geschrieben
Press any key to continue_

```

Inhalt der Datei "SortOut.txt":



## 18.5 Übung 9

Kopieren einer Text-Datei:

```
using System;
using System.IO;

namespace TextDateiKopieren
{
 class Class1
 {
 [STAThread]
 static void Main(string[] args)
 {
 Class1 c = new Class1();

 // Es wird die Instanz "dieQuellDatei" der Klasse FileInfo angelegt
 FileInfo dieQuellDatei = new FileInfo(@"C:\CsharpTest\Quelle.txt");

 // Jetzt steht der Stream zur Verfügung, der "lesen" genannt wird
 StreamReader lesen = dieQuellDatei.OpenText();

 // Es wird die Instanz schreiben des Datenstroms StreamWriter
 // angelegt,
 // der zum Beschreiben von Text-Dateien dient.
 StreamWriter schreiben = new StreamWriter(@"C:\CsharpTest\Ziel.txt",
 false);

 string text;
 do
 {
 //Quelle-Datei lesen
 text = lesen.ReadLine();
 //Ziel-Datei schreiben
 schreiben.WriteLine(text);
 }while (text != null);

 // Nachricht für den Anwender,
 // wobei die \-Zeichen mit der \-Escape-Sequenz rausgeschrieben
 // werden müssen
 Console.WriteLine("Text-Datei C:\\CsharpTest\\Quelle.txt wurde
 kopiert in die");
 Console.WriteLine("Text-Datei C:\\CsharpTest\\Ziel.txt");
 }
 }
}
```

```

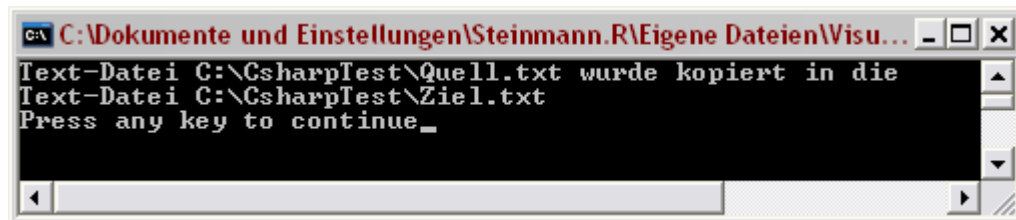
 //Aufräumen
 lesen.Close();
 schreiben.Close();
 }
}

```

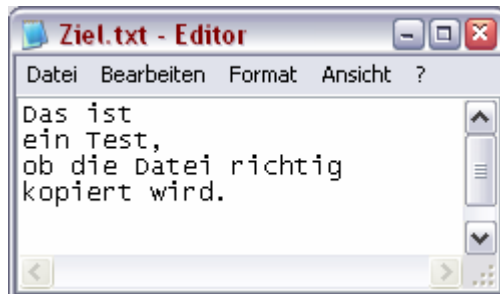
Ausgangsdatei:



Programmausgabe:



Ergebnisdatei:



## 18.6 Übung8

Zunächst wird hier eine Lösung mit dem **Array** vorgeschlagen, wie es in der Vorlesung behandelt wurde. Das Problem hierbei ist, dass die Größe eines Arrays bereits vor dem Kompilieren festgelegt werden muss. In dieser Übung sollen die Eingangswerte jedoch aus einer Datei eingelesen werden. Zum Zeitpunkt des Programmierens wissen wir jedoch noch nicht, wieviele Zahlen die Datei mit den Eingangswerten enthalten wird. Wir können unser Array also entweder nur zu groß oder zu klein dimensionieren. Eine Sicherheitsabfrage verhindert, dass über die Array-Grenzen geschrieben wird, falls die Datei mit den Eingangswerten sehr groß ist.

Alternativ wird deshalb noch eine zweite Lösung gezeigt, die dieses Problem umgeht. Es wird hier ein anderer Typ Array verwendet, ein sog. **ArrayList**, der nicht in der Vorlesung behandelt wurde.

### 18.6.1 Lösung mit Array

Hier also zunächst die Lösung mit dem starren Array. In diesem Fall wurde es auf 7 Elemente dimensioniert, d.h. es kann nur 7 Werte aus der Eingangsdatei "SortIn.txt" lesen.

```
using System;
using System.IO;

namespace BubbleLeseUndSchreibe
{
 class Class1
 {
 [STAThread]
 static void Main(string[] args)
 {
 Class1 c = new Class1();
 int[] explizitesArray = new int[7];
 c leseWerte(explizitesArray);
 c zeigeWerte(explizitesArray);
 c bubble(explizitesArray);
 c zeigeWerte(explizitesArray);
 c schreibeWerte(explizitesArray);
 }

 private void leseWerte(int[] intWerte)
 {
 // Es wird die Instanz "dieQuellDatei" der Klasse FileInfo angelegt
 FileInfo dieQuellDatei = new FileInfo(@"C:\CsharpTest\SortIn.txt");

 // Jetzt steht der Stream zur Verfügung, der "lesen" genannt wird
 StreamReader lesen = dieQuellDatei.OpenText();

 //Text-Datei lesen:
 //Die Schleife läuft so lange, bis entweder das Ende der Text- Datei
 //erreicht wurde ("text" hätte dann den Wert "null" oder bis die
```

```

//größte Adresse des Arrays erreicht wurde.
//Enthält die Datei also mehr Werte, als das Array aufnehmen kann,
//würden die restlichen Werte nicht gelesen.
//Andernfalls ist das Array überdimensioniert und wird nicht
//bis zu seiner Kapazitätsgrenze genutzt.
//Leider hat man bei Arrays keine Wahl zwischen diesen zwei
//Extremen und muss sich für einen Weg entscheiden.
//Ausweg: anstelle von Array den Typ ArrayList verwenden.
int i = 0;
string text;
do
{
 text = lesen.ReadLine();
 intWerte[i] = int.Parse(text);
 i++;
}while ((text != null) && (i < intWerte.Length));

// Nachricht für den Anwender,
// wobei die \-Zeichen mit der \-Escape-Sequenz rausgeschrieben
werden müssen
 Console.WriteLine("Text-Datei C:\\CsharpTest\\SortIn.txt wurde
gelesen");

//Aufräumen
lesen.Close();
}

public void zeigeWerte(params int[] intWerte)
{
 foreach (int i in intWerte)
 {
 Console.WriteLine("Wert =" + i);
 }
 Console.WriteLine("-----");
}

private void schreibeWerte(params int[] intWerte)
{
 // Es wird die Instanz schreiben des Datenstroms StreamWriter
angelegt,
 // der zum Beschreiben von Text-Dateien dient.

 StreamWriter schreiben = new
StreamWriter(@"C:\CsharpTest\SortOut.txt", false);

```

```

//Text-Datei beschreiben
foreach (int i in intWerte)
{
 schreiben.WriteLine(i);
}

// Nachricht für den Anwender,
// wobei die \-Zeichen mit der \-Escape-Sequenz rausgeschrieben
werden müssen
Console.WriteLine("Die sortierte Datei C:\\CsharpTest\\SortOut.txt
wurde geschrieben");

//Aufräumen
schreiben.Close();
}

public void bubble(params int[] intArray)
{
 for (int i = 0; i < intArray.Length; i++)
 {
 for (int j = intArray.Length-1; j > i; j--)
 {
 if(intArray[j] <= intArray[j-1])
 {
 int temp = intArray[j];
 intArray[j] = intArray[j-1];
 intArray[j-1] = temp;
 }
 }
 }
}
}

```

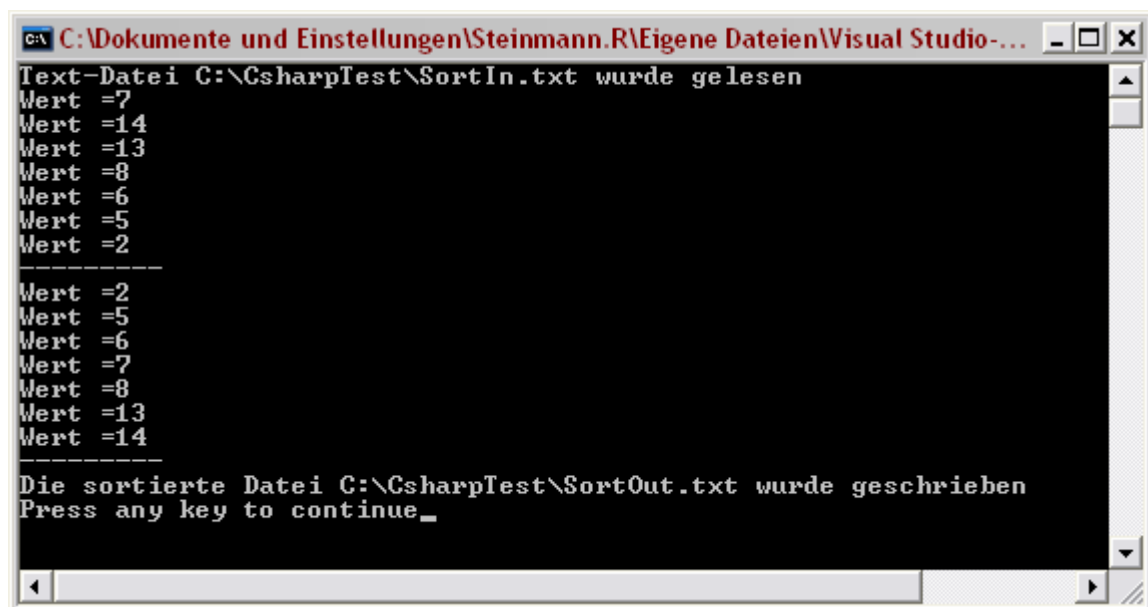
Ausgangsdatei:





Ausgabe aus Programm:

Wie man sieht, wurden nur die ersten 7 Werte der Ausgangsdatei berücksichtigt.



Ergebnisdatei:



## 18.6.2 Übung 10: Lösung mit dem dynamischen ArrayList

Hier die Lösung mit dem dynamischen Array-Typ ArrayList:

```
using System;
using System.IO;
//Wenn man ArrayList verwenden möchte, muss man eine Referenz auf den
//System.Collections-Namespace einfügen.
using System.Collections;

namespace BubbleSortArrayListLeseSchreibe
{
 class Class1
 {
 [STAThread]
 static void Main(string[] args)
 {
 Class1 c = new Class1();

 ArrayList dynamischesArray = new ArrayList();

 c.leseWerte(dynamischesArray);
 c.zeigeWerte(dynamischesArray);
 c.bubble(dynamischesArray);
 c.zeigeWerte(dynamischesArray);
 c.schreibeWerte(dynamischesArray);
 }

 private void leseWerte(ArrayList intWerte)
 {
 // Es wird die Instanz "dieQuellDatei" der Klasse FileInfo angelegt
 FileInfo dieQuellDatei = new FileInfo(@"C:\CsharpTest\SortIn.txt");

 // Jetzt steht der Stream zur Verfügung, der hier "lesen" genannt
 wird
 StreamReader lesen = dieQuellDatei.OpenText();

 //Text-Datei lesen:
 //Die Schleife läuft so lange, bis das Ende der Text-Datei
 //erreicht wurde ("text" hätte dann den Wert "null"). Über die
 //Add-Methode kann das ArrayList befüllt werden. Bei einem ArrayList
 //braucht man sich um die oberste Array-Grenze beim Beschreiben
 //keine Sorgen zu machen. Diese wird vom ArrayList-Objekt selbst
```

```

//verwaltet.
string text;
do
{
 text = lesen.ReadLine();
 //Einem ArrayList darf kein Wert "null" zugewiesen werden.
 //Deshalb muss die Zuweisung in diesen Fällen ausgefiltert
 //werden. Zumindest einmal, am Ende der zu lesenden Datei,
 //kommt der Wert "null".
 if (text != null)
 intWerte.Add(int.Parse(text));
}while (text != null);

// Nachricht für den Anwender,
// wobei die \-Zeichen mit der \-Escape-Sequenz rausgeschrieben
werden müssen
Console.WriteLine("Text-Datei C:\\CsharpTest\\SortIn.txt wurde
gelesen");

//Aufräumen
lesen.Close();
}

public void zeigeWerte(ArrayList intWerte)
{
 foreach (int i in intWerte)
 {
 Console.WriteLine("Wert =" + i);
 }
 Console.WriteLine("-----");
}

private void schreibeWerte(ArrayList intWerte)
{
 // Es wird die Instanz schreiben des Datenstroms StreamWriter
angelegt,
 // der zum Beschreiben von Text-Dateien dient.
 StreamWriter schreiben = new
StreamWriter(@"C:\CsharpTest\SortOut.txt", false);

 //Text-Datei beschreiben
 foreach (int i in intWerte)
 {

```

```

 schreiben.WriteLine(i);
 }

 // Nachricht für den Anwender,
 // wobei die \-Zeichen mit der \-Escape-Sequenz rausgeschrieben
 // werden müssen
 Console.WriteLine("Die sortierte Datei C:\\CsharpTest\\SortOut.txt
wurde geschrieben");

 //Aufräumen
 schreiben.Close();
}

public void bubble(ArrayList intArray)
{
 //Das Attribut für die höchste Adresse heißt bei ArrayList "Count"
 //und nicht "Length", wie bei einem Array.
 for (int i = 0; i < intArray.Count; i++)
 {
 for (int j = intArray.Count-1; j > i; j--)
 {
 //Im Gegensatz zu einem Array, das mit einem Datentyp definiert
 //wird und damit nur Werte dieses Typs enthalten kann,
 //ist ein ArrayList eine Klasse, die alle möglichen Objekte
 //enthalten kann.
 //Da ein Objekt unbekannten Typs nicht einfach einer Variablen
 //mit bestimmtem Typ zugewiesen werden kann, muss eine
 //explizite Typkonvertierung erfolgen:
 int iRechts = (int) intArray[j];
 int iLinks = (int) intArray[j-1];

 if(iRechts <= iLinks)
 {
 int temp = iRechts;
 intArray[j] = intArray[j-1];
 intArray[j-1] = temp;
 }
 }
 }
}
}
}
}
}

```

Ausgangsdatei:



Programmausgabe:



Ergebnisdatei:



## 19 Studienarbeit

Abgabe im ersten Semester.

Prädikat "mit Erfolg teilgenommen" ist Prüfungszulassung.

### 19.1 1. Aufgabe

Schreiben Sie ein C#-Programm, in dem die Auflagerkräfte und das maximale Biegemoment für einen Einfeldträger unter einer Linienlast errechnet und ausgegeben werden.

Der Anwender soll die Wahl zwischen folgenden Lastfällen haben:

- Gleichmäßig verteilte Last (Rechteck)
- Dreieckslast mit Maximum am rechten Auflager
- Trapezförmige Last
- Parabelförmig verteilte Last mit Maximum in Trägermitte

Das Programm soll die Ergebnisse mit einer Last-Toleranz von  $\pm 20\%$  der angegebenen Lasten ermitteln und die Ergebnisse in 1/5-tel Schritten zwischen minimaler und maximaler Last in einer Übersicht ausgeben.

### 19.2 2. Aufgabe

Erweitern Sie das Programm aus Aufgabe 1 wie folgt:

- Der Anwender soll die Wahl haben, seine Eingabewerte in einer Datei abspeichern zu können, und dabei den Dateinamen bestimmen können
- Bei Neustart des Programmes soll der Anwender die Wahl haben, entweder bereits frühere Dateien mit Eingabewerten verwenden, oder aber neue Eingabewerte eingeben zu können.
- Die Ergebnisse sollen ebenfalls in einer Datei abgespeichert werden, wobei der Anwender auch den Namen dieser Datei bestimmen können soll.

### 19.3 3. Aufgabe

Entwickeln Sie ein Programm, das die Momentenlinie für den Träger aus Aufgabe 1 unter dem Lastfall "Trapezförmige" Last in 1/10-tel Schritten berechnet und darstellt. Das Programm soll dabei auf eine Eingabe-Datei zugreifen, die der Anwender mit dem Programm aus Aufgabe 2 abgespeichert hat.





