Inhaltsverzeichnis

Tag 0 3
Disclaimer..... 3
Was wird benötigt..... 3
Optional..... 3
Das .NET Framework..... 3
Zum Thema Objektorientierung (OO)..... 4
Zum Thema Namespaces (Namensräume)..... 4
Zum Thema Klassen..... 5
Tag 1..... 7
Einrichten einer Umgebung..... 7
Coding Conventions..... 8
Unser erstes Programm..... 9
Ein erster Erklärungsversuch..... 9
Datentypen..... 10
Unser zweites Programm..... 11
Strings, „" und hä ? - Was soll das ?..... 13
Strichpunkte..... 14
Die Signatur einer Methode..... 14
Tag 2..... 17
Konvertierung..... 17
Verdopple Zahl – Teil 2..... 19
Kontrollstrukturen..... 20
Bedingungen:..... 20
Schleifen..... 21
Theorie: Operatoren und Schreibweisen..... 22
Strings..... 25
Tag 3..... 27
Klassen..... 27
Kapselung..... 29
Der Operator „new"..... 30
Konstruktoren..... 31
Tag 4..... 34
Schleifen und Bedingungen – Revisited..... 34
Arrays 37
Gültigkeit von Variablen..... 38
Tag 5..... 44
Vererbung – Revisited..... 44
Unser erstes Fenster – Hello World (Reloaded)..... 49
Ein erstes Control..... 52
Tag 6..... 54
Ahhh, Fehler 54
Events – Es ist was passiert..... 56
Appendix A: SharpDevelop..... 58
Credits..... 58
Installation..... 58
Das erste Projekt..... 58
Ein erster kurzer Test..... 58

Tag 0

Disclaimer

Ich bin weder hauptberuflich Programmierer, noch habe ich Germanistik studiert. Daher bitte ich jeden Leser, mir meine Rechtschreib- und Grammatikfehler zu verzeihen. Sollte auch einmal ein Satz nicht ganz "Deutsch" sein, so bitte ich dies zu entschuldigen. Ein bischen Vorstellungsvermögen gehört einfach dazu. Sollte auch so manches Programm nicht allen Regeln der Programmierkunst entsprechen, so ist jeder eingeladen, Verbesserungen einzubringen. Was ich schreibe, ist mein eigener Erfahrungsschatz. Gewisse Dinge entsprechen nicht 100%-ig den Regeln und Vorstellungen bzw. genauen Definitionen, aber ich konnte sie mir so besser merken. Ob das was ich schreibe in der richtigen Reihenfolge kommt, weiss ich auch nicht, aber ich versuche von einem auf das Andere zu kommen, sodass man schrittweise voran kommt und nicht abgehackte Teile lernt.

Was wird benötigt

- ein PC ;)
- Ein funktionierendes WindowsXP (Home/Prof), Windows 2003 Server, Win2k (das SDK/.NET Runtime ist nur für diese Betriebssysteme verfügbar) – Alternativen gibt es in Form des Mono – Projekts (siehe Sourceforge).
- Ein Editor deiner Wahl (Notepad reicht theoretisch aus, Ultraedit und Co. sind natürlich handlicher) – Eine gute Wahl ist Sharp Develop (#Develop)
- Das Microsoft .NET Framework SDK
- Ein klein wenig sonstiges Computerwissen (Verzeichnis wechseln in der Dos Box, ...) .

Optional

Visual Studio 2003: Da dies jedoch nicht für jeden günstig zu erwerben ist, wird es hier nicht benötigt. Sollte man sich jedoch später entschliessen, grössere Projekte anzugehen, so ist es sicher eine gute Wahl. Kleiner Tip: Für Schüler und Studenten gibt es Visual Studio so um die 100 €, für Studenten der TU Wien um ~ 10€ (genauer Preis nicht bekannt, aber so ca.)

Das .NET Framework

Das .NET Framework SDK stellt die Basis für die Programmierung mit C# (und den anderen .NET Sprachen wie ASP.NET und VB.NET) dar.
Ein Nachteil ist, dass wenn jemand ein Programm das mit .NET geschrieben wurde, ausführen will, er das .NET Redistributable Kit benötigt. Dieses ist im Endeffekt das SDK ohne die Entwickler Tools wie z.B. Compiler. Das Redistributable Kit wird von Microsoft imho auch über das Update Center

verteilt. Leider ist es mit ~ 25 MB nicht allzu klein geraten. Es muss jedoch nur einmal installiert werden und darf frei weitergegeben werden. Das SDK beinhaltet natürlich auch gleich das Redistributable Package. Es müssen auf Entwicklerseite also nicht beide installiert werden, das SDK reicht.

Zum Thema Objektorientierung (OO)

OO ist eine wunderbare Sache, da sie das Programmieren imho vereinfacht. War früher ein Programm ein einziger Blob voller Funktionen, so werden bei der objektorientierten Programmierung (OOP) einzelne Objekte zu einem Ganzen zusammengefügt. Dabei spielen Dinge wie Vererbung eine große Rolle. Ein erster Blick in die Vererbung:
Wir nehmen an, es gibt bereits ein Objekt "Fahrzeug" (irgendein Programmierer hat das schon mal entworfen, mit allem was dazu gehört). Charakteristisch für so ein Ding ist, dass es Räder und einen Treibstoffverbrauch hat (und noch ein paar Dinge). Nun hätten wir gerne ein Objekt Motorrad und ein Objekt Auto. Beide Objekte (Auto, Motorrad) leiten wir von Fahrzeug ab. Beide haben einen gewissen Treibstoffverbrauch und beide haben eine gewisse Anzahl an Rädern. Um diese Dinge muss ich mich nun nicht mehr kümmern, ein anderer Programmierer hat den Großteil bereits für mich gemacht. Ich baue sozusagen nur etwas "darauf". Von dieser Idee wird noch oft Gebrauch gemacht werden.

Zum Thema Namespaces (Namensräume)

Ein Namensraum ist sozusagen eine Abgrenzung. Man stelle sich 2 Programmierer vor, die an unterschiedlichen Projekten arbeiten. Einer in der Entwicklung von 3D Software, einer in der Entwicklung von 2D Software. Beide möchten ein Objekt vom Typ "Punkt" darstellen. Der 2D Programmierer weist einem Punkt 2 Variablen zu, eine X und eine Y Koordinate. Der 3D Designer weist einer Koordinate zusätzlich noch die Z Koordinate zu. Solange diese beiden nun nie etwas miteinander zu tun haben, ist alles in bester Ordnung. Nun kommt jedoch eine Firma und kauft die beiden bisher eigenständigen Unternehmen auf. Sie will ein Produkt entwickeln, das - sagen wir mal - Charts (Diagramme) sowohl 2dimensional als auch 3dimensional "zeichnen" kann. In genau diesem Moment bekommen wir nun ein Problem: Welcher Punkt ist jetzt gültig. Wir haben 2 mal die Definition für einen Punkt. Der Compiler wird nun hilflos sein und eine Fehlermeldung ausgeben, da er nicht weiss, welche Definition nun gültig ist. Man könnte das Problem auch lösen indem man z.B. einem Punkt den Namen Punkt2D und dem anderen den Namen Punkt3D gibt. Allerdings müsste man das Ganze dann jedoch im gesamten Projekt ändern, was bei mehreren tausend Zeilen Code doch einiges an Arbeit bedeuten würde. Über den Namespace ist immer definiert, welche Definition des Punktes verwendet wird. Ich verpacke also ein Objekt in eine Schachtel und auch wenn ich zwei gleiche Objekte habe, so sind sie in 2 unterschiedlichen Schachteln.

Unsere Programme werden zu Beginn nur in einem einzigen Namespace laufen. Nichts desto trotz werden wir auf andere Namespaces zugreifen müssen.
Namespaces können auch ineinander verschachtelt werden (Ich packe eine Schachtel in eine andere).

Zum Thema Klassen

Achtung: Eigene Definition
Klassen sind Vorlagen für Objekte. Man kann sich eine Klasse als abgeschottetes Ding vorstellen. Beispiel: Mensch. Ein Mensch hat 2 Beine, 2 Arme, einen Mund, usw.. Es ist ein Objekt an sich. Nichtsdestotrotz gibt es viele Menschen. Ich habe also eine Vorlage (Klasse) Mensch aus der ich einen ganz konkreten Menschen mache. Ich erlaube mir die erste Zeile C# Code einfach mal kommentarlos hin zuschreiben.
Mensch Christoph = new Mensch();
Was haben wir hier: Wir haben die Vorlage Mensch und wir haben ein Objekt vom Typ Mensch, nämlich "Christoph ". In etwas technischerer Sprache: Ich erzeuge eine Instanz eines Menschen. Ich nahm die Vorlage (Mensch) und sagte: Christoph = new Mensch() (Christoph ist ein neues Objekt vom Typ Mensch). Jetzt existiert irgendwo im Speicher Speicherplatz für alles was ein Mensch so haben kann (Anzahl von Haaren, Anzahl Beine, ...).
Man kann diese Zeile auch in 2 Zeilen schreiben, dazu jedoch später. Wenn man diese Zeile noch nicht versteht – nicht verzweifeln. Das wird schon noch deutlich.
Nun gut, aber was ist jetzt so eine komische Klasse, und v.a. warum nennt man das
„Klasse“ und nicht Objektvorlage. Tjo . .gute Frage ;) .. Das habe auch ich in 5 Jahren nicht wirklich erfahren können.
Jedenfalls besteht eine Klasse aus 2 großen Bereichen. Methoden und Variablen. Variablen beinhalten Werte wie z.B. die Haarlänge. Methoden sind - ganz grob gesagt - Funktionen, die die Variablen verändern (nicht nur, aber auch).
Man stelle sich wieder das Objekt Mensch vor. Eine von vielen Variablen, die die einzelnen Menschen unterscheidet ist die Haarlänge. Angenommen ich habe eine Haarlänge von 5 cm, so würde ich z.B. in der Variable Haarlaenge (Umlaute sind als Variablennamen NICHT erwünscht) einen Wert 5 speichern.
Nun hat die Vorlage "Mensch" eine Methode GeheZuFriseur. In C# Code würde man das schreiben als Christoph.GeheZuFriseur(). Christoph ist - wie wir wissen, ein Objekt vom Typ Mensch. Da "Mensch" eine Variable "Haarlaenge" und eine Methode "GeheZuFriseur" hat, hat eine Instanz von Mensch (ein konkretes Objekt) natürlich auch diese Variable und diese Methode.
Diese Methode ändert nun im Endeffekt die Variable Haarlaenge auf sagen wir den Wert 2.
Und genau hier spiegelt sich ein Prinzip der Objektorientierten Programmierung wider. Variablen sollen nicht mit dem Variablennamen verändert werden,

sondern nur durch Methoden. Dadurch wird u.a. auch sichergestellt, dass die Variable keinen ungültigen Wert einnimmt (z.B. -1). Dies kann alles in der Methode überprüft werden.
Das Objekt Christoph ist nach aussen hin abgeschottet, es kann die Haarlaenge nicht einfach so verkürzt werden (über die Variable), nein, er muss dafür zum Friseur (über die Methode).
Zusammenfassung:
Ein Objekt wird aus einer Vorlage erstellt. Man kann aus einer Vorlage hunderte von Objekten erstellen, die sich durch die unterschiedliche Variablen unterscheiden (bspw: Haarlänge). Die Variablen werden durch Methoden (Funktionen) geändert. Man nennt dies Kapselung (die Variablen sind von der Umgebung abgekapselt und nur über Methoden zu „erreichen").

Aufgaben:
- Download des SDKs (siehe Links)
- Installation desselben

Fragen:
- Wie ändere ich in einer Klasse eine Variable?
- Was versteht man unter Kapselung
- Wie kann man sich einen Namespace vorstellen, wozu ist sowas gut ?
- Es gibt ein Schlüsselwort, das ein neues Objekt von einer Vorlage anlegt – Wie lautet es.

Tag 1

Einrichten einer Umgebung

Im Endeffekt gibt es 3 Möglichkeiten:

1.) Die Spar - Variante: Notepad + SDK

Hierbei wird zum Eintippen des Source Codes Notepad verwendet. kompiliert wird dann von Hand in der DosBox

2.) Die gemässigte Variante: UltraEdit (oder gleichwertiges) + SDK

Hierbei wird UltraEdit zum Eintippen des Codes verwendet. Vorteil: Man hat ein gewisses Maß an Syntax Highlighting (Schlüsselworte werden farblich hervorgehoben), ausserdem wird immer die aktuelle Zeile angezeigt (sehr brauchbar, weil vom Compiler Fehlermeldungen mit der Zeilennummer ausgegeben werden) und noch ein paar Dinge mehr .. Ist auf jeden Fall einen Versuch wert - es gibt auch eine 30/45 Tage Testversion.

3.) Die Luxus Variante: Visual Studio .NET

Hier gibts das volle Programm, angefangen vom Syntax Highlighting bis zum Intellisense genannten Feature das alle möglichen Methoden schon einblendet, Syntax Vervollständigung, integrierter Compiler usw. - das volle Programm eben.

Jedoch würde ich gerade für die ersten paar Lektionen Version 2 vorschlagen, zumal man vom Visual Studio auf den ersten Blick erschlagen wird.

Vielleicht gibt es dann auch mal eine Einführungsstunde. Nichts desto trotz: Visual Studio ist NICHT Pflicht und es geht auch gut ohne - ich würde sagen man lernt sogar etwas besser - zumindest Anfangs.

Nachdem das SDK nun installiert wurde (und man sich wahrscheinlich nicht gemerkt hat wohin), sollte man nun den Pfad aktualisieren, sodass der Compiler immer gefunden werden kann. Der Compiler versteckt sich in der Datei mit dem aussagekräftigen Namen csc.exe. (CSC steht Für C Sharp Compiler - und genau so wird C# auch ausgesprochen) Wir bemühen also nun den Suchen Dialog und lassen uns den Pfad anzeigen wo diese liegt. Den Pfad fügen wir zu unserer Path Variable hinzu: In WinXP unter Systemsteuerung -> System -> Erweitert -> Umgebungsvariablen -> Dort gibt es eine die sich PATH nennt ... Wir bearbeiten diese und hängen an das bereits existierende Konstrukt einen ";" (Ohne die ") und dahinter den Pfad an. Den Pfad selbst umgeben wir mit "". Das Ganze sollte dann in etwa so aussehen:

```
Alt: "C:\MeinPfad\Irgendwo";....;"c:\der30stePfad"

Neu: "C:\MeinPfad\Irgendwo";....;"c:\der30stePfad";"C:\PfadZuUnsererCSCEXE"
```

Wir bestätigen das Ganze und öffnen eine Dos Box (unter XP: Start ->

Ausführen -> cmd <Enter>) und geben "csc" ein. Kommt nun eine Fehlermeldung von wegen Datei nicht gefunden -> bitte beginne von oben, da stimmt noch was mit dem Pfad nicht :) .. Kommt eine Meldung von wegen "Input not specified" so sind wir am Ziel. Der Compiler will was tun, hat aber noch keine Quelltext - Datei. Kein Wunder - Die entsteht erst.

4.) Von einem interessierten Leser wurde ich darauf gebracht, dass es eine frei verfügbare Entwicklungsumgebung für C# gibt. Diese heisst #Develop und sollte per Google oder ähnlichem zu finden sein. Dies ist die Lösung, welche ich forcieren werde.

Coding Conventions

Sofern unsere Programme nicht allzu groß sind, macht man sich darum, wie man Variablennamen schreibt noch keine Gedanken. Wird das Programm jedoch grösser, so ist es recht bald einmal vorbei mit der Übersicht über das gesamte Projekt und man wird froh sein, dass man sich an die Coding Conventions gehalten hat. Coding Conventions sind sozusagen die Groß und Kleinschreibung unter der Rechtschreibung ;).

Es geht darum: Wie erkenne ich in einem Programm am besten, was Variablen und was Methoden sind. Es kommen später noch einige andere Dinger dazu, aber hier erstmal eine kurze Geschichte: Vor noch nicht allzu langer Zeit wurde die sogenannte ungarische Notation verwendet. Man benannte jede Variable mit dem Datentyp und danach einem Namen. Angenommen ich hatte eine Integer Variable (Integer ist eine ganze Zahl - 1,2,35,8,10,2003,...) so wurde diese iMeinezahl (oder auch intMeinezahl) genannt. Man konnte sofort auf den ersten Blick erkennen welchen Typ die Variable hatte (in diesem Fall: Integer). Seit C# (wenn nicht schon länger) jedoch versucht Microsoft von der ungarischen Notation ab zukommen und verwendet eine Mischung aus Pascal und Camel Notation. Im Großen und Ganzen:

Coding Conventions Rule #1: Methoden werden am Anfang Groß geschrieben, danach jedes weitere Wort groß

Coding Conventions Rule #2: Variablen werden am Anfang klein geschrieben, danach jedes weitere Wort groß.

Methodennamen würden also lauten "GeheZuFriseur", "BerechneDieWurzel", "TuDiesTuJenes". Variablennamen hingegen lauten z.B. "zahl", "meineZahl", "meineHaarlaenge", "dosenInhalt", ...

Da ich noch nie ein großer Fan der ungarischen Notation war ("danke, ich weiss was meine Variablen für einen Typ haben"), werde ich mich an die Coding Conventions halten, die auch Microsoft forciert. An die oben genannten also. Weitere Coding Conventions werden Einzug halten, wenn wir sie benötigen.

Unser erstes Programm

In jedem Anfängerbuch ist das erste Programm ein "Hello, World" Programm. Dabei wird - wie auch immer - irgendwie der Text "Hello World" auf den Bildschirm geschrieben. Danach wird das Programm beendet - mehr nicht. Genau das wollen wir hier auch einmal erledigen.

Man nehme nun den Editor seiner Wahl (z.B. UltraEdit) und schreibe folgenden Text (schreiben, NICHT kopieren, will man was lernen hilft nur tippen - und wenn es 10 mal so lang dauert).

```
using System;

class HelloWorld

{

  public static void Main()

  {

    Console.WriteLine("Hello World");

  }

}
```

Wir speichern diese Datei als HelloWorld.cs (Vorsicht Editor-Benutzer - Notepad macht gerne noch zusätzlich ein .txt hinten dran) und öffnen wieder unsere Dos - Box. Wir wechseln in das Verzeichnis wo wir die Datei gespeichert haben und geben folgende Zeile ein:
csc HelloWorld.cs
Ein paar Sekunden später ist der Compiler fertig, und wir finden im Verzeichnis eine neue .exe - HelloWorld.exe.
Wir können diese nun mittels
HelloWorld
starten.
Erwartungsgemäss passiert nicht viel, wir erhalten einen Schriftzug, aber der Grundstein wurde gelegt. Unser erstes Programm ist fertig.

Ein erster Erklärungsversuch

Nur was macht das Ganze nun - Was bedeuten die Zeilen?
Wie wir wissen, ist alles in C# in irgendeiner Form ein Objekt - also eine Klasse (Aussage nicht ganz korrekt, aber ich lass sie mal so stehen). Wir haben unsere Klasse HelloWorld genannt. Die geschwungenen Klammern kennzeichnen, dass alles was innerhalb von Ihnen steht, nun Teil dieser Klasse ist. Wenn man Code schreibt, sollte man immer wenn man eine Klammer aufmacht, sie auch gleich zumachen und dann dazwischen schreiben - sonst

gibts Probleme - wenn man irgendwann mal tiefer verschachtelt ist. Wird eine Klammer geöffnet, so wird alles danach eingerückt (ob jetzt 2-3 Leerzeichen oder ein Tab Zeichen, das ist dem Compiler egal, theoretisch müsste man es nicht - aber das Programm ist schon bei 10 Zeilen deutlich leichter zu lesen).

Coding Conventions Rule #3: Öffnet man eine Klammer, so muss alles zwischen dieser und der schliessenden Klammer eine Ebene eingerückt werden.

Wo weiss der Compiler nun, wo er beginnen muss: In jedem C# Programm sucht er nach einer Methode die folgendermaßen deklariert ist:

```
"public static void Main()"
```

Diese ist der Einstiegspunkt bei der Programmausführung.
Was public, static und void bedeutet interessiert uns erstmals nicht. Wichtig ist nur: Diese Methode MUSS so definiert sein und sie muss in jedem Programm vorhanden sein. Wiederum - der Inhalt der Methode wird durch geschwungene Klammern gekennzeichnet. Und wiederum - EINRÜCKEN !
Es bleiben also nur noch 2 Zeilen.

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

Hier haben wir drei Dinge: Ein Objekt, eine Methode und einen String (Zeichenkette). Gefolgt von einem ";"
Das Objekt: Console. Console ist einfach ein Objekt auf dem man Zeichen textbasiert ein und ausgeben kann. Unter Linux würde man es wahrscheinlich als Terminal bezeichnen oder ähnliches. Console ist nichts anderes als eine Art Dos-Box - gewissermaßen halt .. (schwer zu erklären).
Die Frage ist nun, woher weiss der Compiler woher er dieses Objekt nehmen soll. Er weiss es durch die erste Zeile: using System;
System ist nun einer der ersten berühmten Namespaces. Objekte werden einem Namespace zugeordnet. Und im Namespace System liegt das Objekt Console.
Theoretisch wäre es möglich gewesen die erste Zeile wegzulassen und stattdessen

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

zu schreiben. Der Effekt wäre derselbe gewesen. Grundsätzlich: Wird ein Namespace mehr als 2,3 mal gebraucht, ab nach oben mit ihm. Das Schlüsselwort "using" sagt dem Compiler, er soll nach Objekten im Namespace System suchen.

Datentypen

Ein kleiner Einschub zum Thema Datentypen muss nun einfach sein. Es gibt unterschiedliche Datentypen, die verwendet werden können. ABER: Bei der

Umwandlung von einem Datentyp in einen anderen kann es zu Problemen kommen - dazu später noch mehr.
Die von uns zu Beginn verwendeten sind:

- int - Integer - Variable: Eine Ganze Zahl (z.B.: 31683)
- char - Character - Variable: Ein einzelner Buchstabe (z.B.: x)
- string - String - Variable: Eine Zeichenkette (mehrere Buchstaben hintereinander) (z.B.: Hallo)
- double - Kommazahl mit doppelter Präzision (0.5438373)
- void - Ein Zeiger ins Leere (oder auch - Nichts)

Wir haben oben bereits einen Datentyp verwendet, nämlich einen String ("Hello World").
Wir lassen das nun mal so im Raum stehen und schreiben unser nächstes Programm: Verdopple Zahl

Unser zweites Programm

Wir beginnen wieder mit der Grundstruktur eines jeden C# Programmes. Der Definition des Klassennamens, gefolgt von einer Methode namens Main. Ganz oben binden wir mittels using den Namespace System ein.

```
using System;

class VerdoppleZahl
{
      public static void Main()
      {

            Console.WriteLine("blaaa");
      }
}
```

Bis hierher kannten wir schon alles .. Aber jetzt wollen wir etwas Funktionalität in die Sache bringen. Was wir wollen:

- Das Programm soll uns nach einer Zahl fragen
- Wir sollen diese Zahl eingeben können
- Das Programm soll das Doppelte der Zahl wieder ausgeben

Schritt 1) Nach Einer Zahl fragen.
Das kennen wir mittlerweile, wir nutzen hier wieder die Methode WriteLine des Objekts Console.

```
Console.WriteLine("Bitte geben Sie eine Zahl ein:");
```

Schritt 2) Warten bis der Benutzer die Zahl eingegeben hat. Da wir das Console Objekt bereits kennen, könnte man sich doch vorstellen, dass man auf dem Console Objekt nicht nur etwas ausgeben, sondern auch irgendwie etwas einlesen kann. Nur wie kommt man zu dieser Information? Man blickt in die

MSDN – die Bibel. Die MSDN ist höchst umfangreich zumal sie nicht nur die gesamte Dokumentation für das gesamte .NET Framework enthält (Console ist z.B. ein Teil des .NET Frameworks). Wir bemühen evtl. die Suche und finden irgendwann den Namespace System (.NET Framework -> Reference -> Class Library -> System). Hier sehen wir eine Übersicht über sämtliche Klassen, die in System enthalten sind. Dazu gehört auch unsere Klasse Console. Ein Klick auf Console und wir sehen nun die Felder (Fields, Variablen) und Methoden der Klasse. Man nennt diese „Members". Wenn ich also von den Members der Klasse Console spreche, so meine ich damit deren Methoden und Felder. Wir sehen nun schon dass es auch eine Methode namens ReadLine gibt, was ja schon recht vielversprechend klingt.
Ein Klick darauf und schon sehen wir den gesamten Funktionskopf:

```
public static string ReadLine();
```

Die beiden ersten Worte lassen wir noch beiseite, interessant wird der Teil danach. Wir vergleichen den Kopf der Methode Main mit dem von ReadLine.

```
public static void Main()

public static string ReadLine()
```

Die Ähnlichkeit springt sofort ins Auge. Aber was bedeutet das alles nun. Erinnern wir uns an die Datentypen. String ist ein Datentyp und void ist ebenso einer. Das Wort VOR dem Methodennamen kennzeichnet einen sogenannten Rückgabewert. Sprich: Die Funktion wird ausgeführt und wenn der Code der Methode ausgeführt wurde, die Funktion also fertig ist, so liegt ein Wert vom Typ string vor.
Wie kann ich das nun nutzen.
Beispielsweise so:

```
string input;

input = ReadLine();
```

Was machen wir hier: Wir legen im Speicher des PCs Platz für ein Objekt (auch Datentypen sind Objekte) vom Typ string an. Danach weisen wir input (unserer Variable bzw. unserem Feld) einen Wert zu (über den Operator =), und zwar den, den ReadLine() zurückgibt.
Wie können wir das testen?
Wir könnten zum Beispiel einfach etwas von der Console lesen und das dann wieder ausgeben.
Und genau das machen wir:

```
using System;

class GibWasEin
```

```
{

	public static void Main()

	{

		Console.WriteLine("Bitte geben Sie etwas ein:");

		string input;

		input = Console.ReadLine();

		Console.WriteLine(input);

	}

}
```

Nicht verzweifeln wenn nun nach dem „Bitte geben Sie etwas ein" nichts mehr passiert. Kein Wunder – er wartet auf eine Eingabe. Wir geben also nun „Hallo" ein (ohne die „) und drücken Enter. Es erscheint nun „Hallo" und das Programm wird beendet.
Wie hat das nun funktioniert?
Console.WriteLine hat einen Text ausgegeben, danach wurde Platz für einen string im Speicher reserviert, auf den wir per „input" zurückgreifen können. Nun wird's interessanter. Wir rufen die Methode ReadLine() auf. Diese wartet so lange bis wir Enter gedrückt haben. Das Programm bleibt also eigentlich in dieser Zeile stehen (bis wir Enter drücken). In dem Moment, in dem wir Enter drücken, liest die Methode das geschriebene aus und liefert es an die Variable (an das Feld) input. Es steht nun also in input „Hallo".
Nun rufen wir wieder Console.WriteLine auf und übergeben das Feld input. Console.WriteLine gibt nun den Inhalt von input aus („Hallo"). Das Programm an sich ist fertig und wird beendet.

Strings, „" und hä ? - Was soll das ?

Wahrscheinlich ist jedem mittlerweile aufgefallen, dass ich scheinbar wahllos Gänsefüsse verteile. Dem ist natürlich nicht so.
Angenommen eine Funktion erwartet einen String als Übergabeparameter (das ist das, was in den Klammern steht – wird noch erklärt), so kann ich ihr entweder den string direkt übergeben oder ihr eine Variable übergeben, die den string enthält.
Die beiden Möglichkeiten:

```
#1:

string output;

output = „Hier ein Text";
```

```
Console.WriteLine(output);
```

und

```
#2:

Console.WriteLine(„Hier ein Text“);
```

sind vom Ergebnis her vollkommen ident.

Grundsätzlich würde ich es so handhaben: Benötige ich ein Feld später noch einmal, so speichere ich es in einer Variable. Gebe ich jedoch nur einen simplen Text wie z.B. eine Fehlermeldung aus, so kann ich den Text gleich als Übergabeparameter schreiben.

Strichpunkte

Genau wie bei den Anführungszeichen bei Strings, scheint es als würde ich auch diese wahllos vergeben. Dabei ist es eigentlich recht einfach. Jede Zeile Code hat am Ende einen Strichpunkt (semi-Colon).

Ausnahmen: Alles, wo danach eine geschwungene Klammer kommt.

Diese Erklärung sollte zu Beginn einmal reichen. Einen Strichpunkt zu vergessen ist nicht schlimm (passiert wohl jedem einmal), aber die Fehlersuche kann ziemlich mühsam werden, weil der Compiler nicht immer die richtige Fehlermeldung (missing „;“) ausspuckt, sondern meint, irgendeinen Fehler darüber oder darunter gefunden zu haben. Also: Strichpunkte besser nicht vergessen.

Die Signatur einer Methode

Jede Methode in einem Programm muss einzigartig sein. Dabei ist nicht nur der Name entscheidend sondern auch Übergabeparameter.

Hier wieder ein bischen Theorie:

Eine Methode kann eine unbegrenzte Anzahl an Übergabeparametern haben. Beispielsweise hat Microsoft selbst in der Anbindung an Excel eine Methode mit etwa 20 Übergabeparametern entworfen.

Ich selbst würde raten: Bei mehr als 6 wird's kompliziert, bei mehr als 10 ist endgültig Schluss. Nur was genau ist nun so ein Übergabeparameter.

Man kann sich das so vorstellen:

Es gibt eine Methode (also einen Block Code) der mit einem Wert etwas berechnen soll. Natürlich könnte ich eine Methode schreiben, die mir die Zahl 10 verdoppelt. Das ist jedoch nicht das was ich will. Ich will eine eingegebene Zahl, also eine, die ich in einem Feld (einer Variable) gespeichert habe,

verdoppeln. Ich übergebe also eine Variable an die Methode.

Natürlich kann ich auch mehrere Variablen an eine Methode übergeben, z.B. wenn ich die x-te Wurzel der Zahl y berechnen will.

Das würde Code mässig dann so aussehen:

```
public void BerechneWurzel(int x, int y) { ... }
```

Will ich nun dass die Methode einen Wert zurückgibt den ich wiederum weiterverwenden kann, so muss ich statt void (nichts) den Rückgabetyp angeben. Sagen wir, ich erwarte mir dass die Methode die x-te Wurzel aus y berechnet und mir dann wieder eine ganze Zahl zurückliefert, so müsste ich:

```
public int BerechneWurzel(int x, int y) { ... }
```

schreiben.

Hätte ich nun in derselben Klasse eine Methode:

```
public int BerechneWurzel(double x, int y) { ... }
```

so würde dieses Programm ohne Probleme lauffähig sein, obwohl es die Methode BerechneWurzel eigentlich zwei Mal gibt. Der Compiler jedoch interessiert sich nicht für den Namen, sondern für die Signatur einer Methode, und dazu zählt eben nicht nur der Name, sondern auch die Art und Anzahl der Übergabeparameter.

Aufgaben:

- Das Hello World Programm zum laufen bringen
- Schreibe das Grundgerüst einer jeden C# Applikation auf
- Das GibWasEin Programm muss funktionieren
- + Schreibe ein Programm, das dich zuerst nach deinem Vornamen fragt, dann nach deinem Nachnamen und das dann Vorname und Nachname ausgibt.

Fragen:

- Was würde bei Console.WriteLine(„output") passieren
- Was für Methoden kennt die Klasse Console noch
- Was gehört alles zur Signatur einer Methode
- Wofür steht das „void" vor Main()
- Welchen Rückgabetyp hat ReadLine und woher bekomme ich diese Information?
- Was macht die Zeile „string input;"
- Wie lauten die Coding Conventions für Methodennamen
- Was bedeutet „using"

- Wie heisst die ausführbare Datei des C Sharp Compilers
- Wie kompiliere ich ein Programm
- Was ist die Standard - Dateiendung für C# Quelltextdateien
- Zählt der Rückgabetyp zu der Signatur einer Methode?

Tag 2

Konvertierung

Wie ich im vorherigen Abschnitt erwähnt habe, gibt es Probleme beim Konvertieren von einem Datentyp in einen anderen. Man stelle sich ein Programm vor, das intern die Temperatur als double speichert (also als Kommazahl – Bsp.: 32,55°C) aber nach aussen hin nur einen Integer Wert angibt (Bsp: 33°C). Der Compiler weiss nun nicht, wie er die double Zahl ohne Informationsverlust nach Integer konvertieren soll. Er wirft eine Fehlermeldung aus („Cannot implicitly convert double to int").
Ein anderes Beispiel: Wir nehmen wieder unser Thermometer. Diesmal kann es jedoch intern nur mit der Genauigkeit von Integer Werten rechnen (also ganzzahlig). Angeschlossen ist es an ein LCD, das theoretisch jedoch auch double Werte anzeigen könnte. Der Compiler liefert hier keinen Fehler. Warum?
Egal wie groß der Integer Wert auch sein mag, das Display kann jeden Wert anzeigen. Der Wert kann also direkt umgewandelt werden. Man nennt dies implizite Konvertierung. Man muss kein Schlüsselwort angeben, um den Compiler zu einer Konvertierung zu zwingen.
Nun wäre es aber sinnlos, könnten wir einen double nicht doch irgendwie in einen int umwandeln. Diesen Vorgang nennt man explizite (ausdrückliche) Konvertierung. Man kann sich das als eine Art Sicherungsmechanismus des Compilers vorstellen. Er weiss, wann Daten verloren gehen „könnten" und warnt. Der Benutzer muss nun entscheiden – Gehe ich das Risiko ein, und meine Information könnte abgeschnitten werden, oder überlege ich mir etwas anderes.

Ein kurzes Schnipsel Code dazu (gehört alles direkt in die Main()):

```
double x;

x = 1.45;

int i;

i = x;

Console.WriteLine(i);
```

Was machen wir hier:
Ich nehme mir ein Feld vom Typ double, weise diesem den Wert 1,45 zu (Wichtig: Im Code gilt der „." (Punkt) als Kommatrennzeichen). Danach nehme ich mir eine integer Variable i und versuche i den Wert von x zuzuweisen. Und genau hier schreit der Compiler laut aus. Er kann 1.45 nicht zu einer ganzen Zahl konvertieren. Es würde Information verloren gehen. Also – eine implizite

Konvertierung ist nicht möglich.
Mir ist jedoch egal, ob mein Thermometer 32,2°C oder 32° anzeigt, also schreibe ich folgenden Code:

```
double x;

x = 1.45;

int i;

i = (int)x;

Console.WriteLine(i);
```

Eigentlich ist es genau dasselbe, bis auf das (int) vor dem x. Damit weise ich den Compiler explizit an, dass ich das konvertieren will und mir der Folgen bewusst bin. Daher wird diese Form explizite Konvertierung genannt.
Console.WriteLine gibt dann 1 aus.

Nun gibt es natürlich auch Konvertierungen wo dies prinzipiell nicht so einfach möglich ist. Man denke beispielsweise daran, einen Text (string) in einen int zu konvertieren. Solang in diesem Text wirklich schön eine Zahl steht, so ist es auch hier kein Problem, aber was sollte der Compiler machen, sollte auf einmal ein Buchstabe vorkommen. Lange Rede kurzer Sinn. Eine Konvertierung von einem string in einen int ist nicht ohne weiteres möglich.
Jedoch biete die Klasse int (ja .. auch Datentypen sind Klasssen) eine Methode namens Parse(..) an. Wir finden diese ebenfalls in der MSDN. Diese Methode nimmt einen string als Übernahmewert und liefert uns einen Integer zurück. Dies funktioniert wunderbar, solang wir wirklich nur Zahlen eingeben, geben wir einen Buchstaben ein, so wird unser Programm abrupt beendet werden. Dies soll uns jedoch momentan noch nicht stören, denn bald werden wir diese Fehler abfangen können.
Ein kleines Stück Code dazu:

```
string str;

str = Console.ReadLine();

int i = (int)str;
```

Wir wissen mittlerweile, was das nun bedeutet. Wir nehmen einen String, und weisen diesem den Wert zu, den Console.ReadLine() zurückliefert. Wir versuchen nun str explizit in einen int zu konvertieren. Dies schlägt fehl. Nicht einmal explizit ist das möglich.
Wir verändern nun die letzte Zeile in

```
int i = int.Parse(str);
```

Der Compiler wird dies ohne Probleme akzeptieren. Das Programm wird auch

funktionieren, solange man gültige Zahlen eingibt. Gibt man jedoch Buchstaben ein, so war's das. Wie wir das umgehen können, dazu später.

Verdopple Zahl – Teil 2

Wir haben nun eigentlich sämtliche Grundlagen gesammelt und sollten nun wissen wie wir einen string einlesen, diesen in eine Zahl umwandeln und dann wieder ausgeben können. Einzig das Verdoppeln fehlt noch – ist jedoch keine Hexerei.
Wir beginnen mit dem Grundgerüst:

```
using System;

class VerdoppleZahl
{
      public static void Main()
      {

            Console.WriteLine("blaaa");
      }
}
```

Schritt 1) Den Benutzer auffordern etwas einzugeben

```
Console.WriteLine(„Bitte geben Sie eine Zahl ein“);
```

Das kennen wir mittlerweile.
Schritt 2) Initialisieren einer Variablen vom Typ string. Dorthin kommt dann das, was der Benutzer eingibt.

```
string str;
```

Dem Feld str das zuweisen, was ReadLine() zurückliefert, nachdem der Benutzer etwas eingegeben hat.

```
str = Console.ReadLine();
```

Anlegen eines Feldes vom Typ int und die Eingabe in einen int parsen.

```
i = int.Parse(str);
```

Den Wert verdoppeln.

```
i = i * 2;
```

Das hier ist noch neu. Man sollte derartige Zeilen am besten von rechts nach links lesen. Wir nehmen den Wert i, multiplizieren ihn mit 2, und weisen ihn wieder dem Feld i zu.

Nun noch die Ausgabe und wir sind eigentlich fertig.

```
Console.WriteLine(i);
```

Kontrollstrukturen

Was wir bisher geschrieben haben, war immer relativ straight – Forward. Der Programmablauf war schon während des Schreibens eines Programmes fix fest gelegt.
Wir kommen daher zu einem Thema, wie man den Programmablauf steuern kann. Die grundlegenden Prinzipien hier sind Teil von fast jeder Programmiersprache, also nicht C# spezifisch. Es mag zwar sein, dass die Syntax etwas abweicht, aber das Prinzip ist immer dasselbe.
Die ersten beiden Themen sind: Bedingungen und Schleifen.

Bedingungen:

Eine Bedingung könnte man folgendermaßen charakterisieren: „Wenn Ein Ereignis eintrifft, tue etwas; Wenn es nicht eintrifft, tue etwas anderes". Genau so wird es in C# auch gehandhabt. Ich erlaube mir einmal wieder ein paar Zeilen Code hinzuschreiben:

```
Console.WriteLine("Bitte eine Zahl eingeben");

string str;

str = Console.ReadLine();

int i;

i = int.Parse(str);



if (i > 10)

{

      Console.WriteLine("Zahl ist Grösser als 10");

}

else

{

      Console.WriteLine("Zahl ist kleiner als 10");

}
```

Den oberen Teil kennen wir bereits. Nach eine Zahl fragen, string einlesen, in einen int umwandeln. Nun zum neuen Teil:
if (kleingeschrieben) kennzeichnet eine Bedingung („wenn"). Dem Schlüsselwort („if" ist ein Schlüsselwort (ein Wort mit spezieller Bedeutung), es darf z.B. nicht als Variable verwendet werden) folgt die Bedingung die überprüft wird (i > 10). Diese Bedingung wird immer in runden Klammern angegeben. Man kann dies genauso lesen: „Wenn i größer 10, dann". Es folgen nun wieder Klammern. Diese zeigen an, dass der Block nur ausgeführt wird, wenn die Bedingung „wahr" war. Es können auch mehrere Zeilen Code in den Klammern vorkommen. Danach folgt das Schlüsselwort „else". Es kennzeichnet den alternativen Teil der Bedingung (ansonsten ...). Ebenfalls wieder gefolgt von einem Codeblock in Klammern. Ich schreibe den if / else Block einfach mal in Worten an:

```
wenn i>10

      Gib einen Text auf der Console aus.

ansonsten

      gib einen anderen Text aus
```

Dies ist wohl die einfachste Form einer Bedingung und wir belassen es einmal dabei. Ich hoffe ich habe das etwas klar gemacht. Eine Zeile Code nach der geschlossenen Klammer des Else – Blockes würde wieder auf jeden Fall ausgeführt werden.

Schleifen

Die meiner Meinung nach einfachste Schleife ist die for – Schleife. Ich schreibe einen Beispielscode hierher:

```
int i;

for (i = 0; i<10; i = i + 1)

{

      Console.WriteLine(i);

}
```

Diese Syntax ist auf den ersten Blick schon etwas schwerer zu verstehen. Wir nehmen wieder eine Variable vom Typ Integer (also eine ganze Zahl). Die Schleife schreibt 10 Zeilen, wobei in der ersten Zeile 0 steht, in der zweiten 1, usw.
Aber warum?
Man liest die for – Zeile folgendermaßen. Beginne bei i = 0, mache das was

zwischen den Klammern steht solange, wie i < 10 ist und erhöhe nach jedem Durchgang i um 1.
Man kann sich das folgendermaßen vorstellen:
Das Programm springt zum ersten Mal in die for – Zeile. I wird auf 0 gesetzt und die Schleife zum ersten Mal durchlaufen. Ausgegeben wird im Endeffekt 0. Die erste Ausführung wurde beendet, i wird um 1 erhöht. Nun wird die Bedingung die oben steht wieder überprüft. Ist i < 10 ? -> ja -> die Schleife wird wieder durchlaufen, i ist momentan 1, also wird 1 ausgegeben. I wird wieder erhöht und ist jetzt zwei. Die Bedingung wird wieder überprüft -> wir sind immer noch kleiner als 10. Das geht so weiter bis i = 9 ist. Die Überprüfung findet wieder statt -> Ja, wir sind immer noch kleiner als 10, es wird der aktuelle Wert von i ausgegeben (9). Nun wird i um eines erhöht und ist jetzt 10. Die Bedingung wird überprüft und ist diesmal falsch (10 ist nicht kleiner als 10). Die Schleife wird beendet und die Programmausführung unter der schliessenden Klammer fortgesetzt.
Ich hoffe das ist alles gut verständlich, denn hier gibt es die meisten Probleme.
Es gibt noch andere Arten von Schleifen und Bedingungen oder Verzweigungen, aber meiner Meinung nach sind diese beiden die wichtigsten.

Theorie: Operatoren und Schreibweisen

Bisher – wollten wir eine neue Variable, wurde immer zuerst der Typ geschrieben und danach ein Wert zugewiesen. Dies ist meiner Meinung nach etwas unübersichtlich – aber vollkommen korrekt.
Man kann die Zeilen:

```
int i;

i = 3;
```

auch in eine Zeile schreiben und sich etwas Platz sparen:

```
int i = 3;
```

Man umgeht so auch ein Problem. Was ist wenn man nur die erste Zeile schreibt und danach z.B. 10 zu i hinzuzählen möchte. I hat in diesem Moment keinen gültigen Wert, es kommt zu einem Problem bei der Programmausführung. Daher: Legt man ein Feld an, so sollte man diesem auch gleich einen Wert zuweisen, z.B.

```
int i = 0;

string str = „Hallo“;

char c = 'x';
```

Auch wenn der Wert zu jenem Zeitpunkt nicht sinnvoll ist, es hilft, Fehler zu vermeiden.

Wir haben bereits ein paar Operatoren kennengelernt. Das beste Beispiel, den aus der Mathematik bekannten Zuweisungsoperator „=". Er weist dem Wert links von ihm, den Ausdruck rechts von ihm zu:

```
i = 7;      // i hat danach den Wert 7

a = b;      // a hat danach den Wert, den b hat

a = b = c;  // b hat danach den Wert den c hat, und diesen hat dann auch a

a = b + 7;  // a hat danach den Wert den b hatte + 7
```

Dies gilt z.B. für die Operationen mit int oder double Feldern. Ähnliches geht auch für string Felder.

```
strTeil1 = „Hallo";

strTeil2 = „Christoph";

strGesamt = „";

strGesamt = strTeil1 + strTeil2;

      // strGesamt enthält nach dieser Zeile „HalloChristoph"
```

Wer sich mittlerweile gefragt hat was die beiden „//" in meinem Text bedeuten, hat gut aufgepasst. Es handelt sich hier um sogenannte Kommentare. Alles was nach den beiden Schrägstrichen steht, wird vom Compiler vollkommen ignoriert. Kommentare helfen jedoch ungemein den Quelltext leserlicher zu machen. Normalerweise würde ich sagen 2/3 Codezeilen, 1/3 Kommentarzeilen. Bei kleinen Programmen wird dies generell etwas vernachlässigt, aber wenn später grössere Dinge anstehen, sind Kommentare unerlässlich. Auch in Firmenprojekten sind diese Pflicht, denn sollte irgendwo ein Fehler versteckt sein, kann sich ein anderer Programmierer leichter im Code zurechtfinden. Pflicht sind sie natürlich nicht. Eine weitere Möglichkeit einen grösseren Block an Kommentaren zu schreiben besteht darin, „/*" und „*/" zu verwenden. Alles was zwischen diesen beiden Markern steht, wird als Kommentar behandelt – also vom Compiler ignoriert. Ein Beispiel:

```
      /* Der folgende Block macht ein paar Ding

         Hier noch ein paar Kommentarzeilen

         Irgendwas macht er

      */
```

Ob das abschliessende Tag („*/“) noch in der letzten Zeile steht oder in der darunterliegenden ist Geschmacksache.
Wichtig ist nur: Alles dazwischen wird vom Compiler ignoriert.

Aber nun zurück zum Thema der Operatoren. Natürlich gibt es eine Vielzahl von Operatoren, die ich gar nicht alle vorstellen kann. Aber die wichtigsten:

<	kleiner als
>	größer als
<=	kleiner gleich
>=	grösser gleich
+=	X += 3; bedeutet: x = x + 3;
++	Inkrement Operator; Beispiel: x++ entspricht: x = x + 1
--	Dekrement Operator; Beispiel: x-- entspricht: x = x - 1
!	Not – Operator: Verneint eine Aussage
*, /, +, -	Multiplikation / Division / Addition / Subtraktion
%	Modulo Operator (NICHT Prozent)
&&	Logisches UND
||	Logisches ODER

Neben dem += Operator gibt es auch noch die -=, /=, *= Operatoren, mit analoger Bedeutung.
Etwas das man sich auf jeden Fall merken MUSS: **Vergleiche finden mit „==“ statt, Zuweisungen mit „=“.** Ich nehme eine if – Bedingung in der ich überprüfe ob die Variable den Wert 10 hat.

```
if (i == 10)
```

Dies ist immer einzuhalten und wird gerne vergessen. Glücklicherweise schreit der Compiler bei einem solchen Fehler die Fehlermeldung „Cannot implicitly convert type 'string' to 'bool'“.

Ein zusätzlicher Datentyp:
Neben int, string, char, double gibt es natürlich auch noch mehr Datentypen, aber einen der wichtigsten haben wir noch weggelassen. Den boolean Datentypen. Er kennt im Endeffekt nur zwei Werte: true (wahr) oder false (falsch). Er wird auch beispielsweise bei der if – Bedingung indirekt verwendet. Hier spielen nun auch einige Operatoren von der vorangegangen Tabelle mit. Der Not – Operator („!“) z.B. verneint eine Aussage.

```
if ( x == 10) { ... }
```

die Verneinung würde lauten:

```
if (!(x == 10)) { ... }
```

Man kann dies in etwa so verstehen. Es wird geprüft ob x gleich 10 ist. Man erhält also indirekt einen boolean (wahr/falsch). Nun wird diese Aussage verneint, aus wahr wird falsch, aus falsch wird wahr. Zusammen mit den UND / ODER Operatoren lassen sich darauf komplexe Bedingungen stricken.
Beispiel:

```
if ((x == 10) && (x<10))
```

Dieser Ausdruck kann niemals wahr sein, denn wenn x gleich 10 ist, kann nicht gleichzeitig kleiner als 10 sein. Derselbe Ausdruck mit einem ODER Operator anstatt des UND Operators, würde die Zahl auf kleiner / gleich 10 prüfen. Man könnte natürlich auch stattdessen

```
if (x <= 10)
```

schreiben.

Strings

Wie wir bereits wissen sind Strings Ketten von Zeichen. Wir haben auch schon gesehen wie man zwei Strings zu einem zusammenfügen kann. Genauso funktioniert es auch mit Zahlen. Hier ein Beispiel:

```
int x = 5;

int y = 10;

Console.WriteLine(x+y);

Console.WriteLine(„x" + „y");

Console.WriteLine(x + „" + y); // „" bezeichnet einen String ohne Zeichen
```

Was geben diese Zeilen nun aus. Die erste gibt erwartungsgemäss „15" aus. Die beiden Zahlen werden addiert und ausgegeben. Die zweite gibt ebenfalls erwartungsgemäss „xy". Der Compiler sieht hier nur zwei Strings die aneinander gehängt werden. Dies ist so als würde ich „Hallo" und „Christoph" aneinander hängen (siehe oben). Die dritte Zeile jedoch, obwohl sie eigentlich der ersten ziemlich gleicht, gibt „510" aus. Nur warum. Es ist doch auch hier ein + dazwischen. Warum addiert er beim ersten Versuch die beiden Zahlen, hängt sie beim Zweiten jedoch hintereinander.
Wieder kann man sich die Zeile von hinten ansehen. Y + „" ergibt zusammen einen string, also 5. Wir haben also nicht mehr zwei Datentypen vom selben Typ, sondern wir haben jetzt noch einen string und einen int. Strings und ints

kann man nicht miteinander multiplizieren (ein string könnte ja genauso gut ein Wort sein). ABER: Man kann einen int implizit (ohne extra den Typ anzugeben) in einen String umwandeln. Und genau das erledigt der Compiler. Wir haben also im Endeffekt 2 Strings, die standardmässig durch das + aneinander gehängt werden.
Das Plus (+) ist ein sogenannter überladener Operator. Je nachdem welche Datentypen links und rechts von ihm stehen, werden unterschiedliche Aufgaben ausgeführt. Im Fall von 2 ints ist es die Addition, im Falle von 2 Strings ist es das Aneinanderhängen. Wer hier noch nicht 100%ig durchblickt, braucht sich nicht zu fürchten, zu Beginn erscheint das alles etwas kompliziert. Mit der Übung jedoch ändert sich das schlagartig. Irgendwann ist es vollkommen normal ewig lange Strings aneinander zu hängen oder sonstwie zu manipulieren. Das Wort „überladen" vergessen wir auch gleich wieder, es kommt viel später wieder vor.

Fragen:

- Wie nimmt der Compiler eine explizite Konvertierung vor – rundet er oder schneidet er einfach alles nach dem Komma ab (kleines Programm schreiben, und dieses einfach ausprobieren)
- Welche Möglichkeiten habe ich, eine Variable von einem Datentyp in einen anderen zu konvertieren.
- Was bedeutet der Ausdruck: „x+=-3"
- Ist es egal ob Variablennamen groß oder klein geschrieben werden

Aufgaben:

- Schreibe ein Programm, das den Benutzer nach zwei Zahlen fragt, diese miteinander multipliziert und dann entscheidet, ob die Zahl größer 30 ist.
- Schreibe einen Ausdruck für eine if Anweisung, die überprüft, ob eine Zahl positiv ist aber dennoch kleiner als 5 ist. Das vollständige Programm soll dann einen sinnvollen Text ausgeben.
- Schreibe ein Programm, das zwei Zahlen einliest, und dann die Rechnung als String mit Ergebnis ausgibt. (z.B.: „12 + 13 = 25")

Tag 3

Klassen

Nachdem nun die Grundlagen erklärt sind, stürzen wir uns kopfüber in die Grundlagen der Objektorientierung. Auch hier werde ich nicht alles erklären, sondern mich zuerst auf die Grundlagen konzentrieren. Schlüsselworte wie public, private und class sind Inhalt dieser Vorlesung. Ausserdem unterhalten wir uns über Konstruktoren und den Garbage Collector. Wir kommen auf „new" und „this" zu sprechen. Diese Einheit ist wohl die absolute Grundlage der Objektorientierung. Sie gilt nicht nur für C# sondern auch für andere objektorientierte Sprachen wie Java, C++ und andere. Einzig die Syntax (die „Rechtschreibung") ist unterschiedlich. Das dahinter steckende Prinzip ist dasselbe.

Ich gehe nun von 2 Klassen aus und erlaube mir einfach die Klassen anzuschreiben. Standardmässig schreibe ich jede Klasse in eine eigene Datei. Die Datei wird mit dem Klassennamen benannt. Dies ist jedoch NICHT zwingend notwendig. Andererseits habe ich mit dieser Arbeitsweise gute Erfahrungen, sodass ich sie nur weiterempfehlen kann. In anderen Sprachen ist dies zwingend nötig.

Das Kürzel

```
//--file-dateiname.cs
```

zeigt in dieser Dokumentation an, in welcher Datei der folgende Inhalt vorkommt. Diese Zeile muss natürlich nicht mit eingetippt werden.

Um mehrere Dateien zu kompilieren und sich nicht um Abhängigkeiten (manche Klassen benötigen eine andere, ...) kümmern zu müssen, kennt der C# Compiler auch die einfache Möglichkeit des:

```
csc *.cs
```

Dabei werden alle Klassen kompiliert und um Abhängigkeiten muss man sich nicht mehr kümmern. Dies ist v.a. Für kleine Projekte zu empfehlen. Bei grösseren Projekten – wenn die Zeit, die der Compiler benötigt, schon mal mehrere Stunden beträgt, so ist dies natürlich nicht zweckmässig. Aber in dieser Vorlesung werden wir keine derartigen Projekte erledigen.
Jetzt aber weiter zum eigentlichen Code:

```
//--file-Hauptklasse.cs

using System;



class Hauptklasse
```

```
{
	public static void Main()
	{
		Datenspeicher ds = new Datenspeicher();
		int z = ds.GetZahl();
		Console.WriteLine(z);
	}
}

//--file-Datenspeicher.cs
class Datenspeicher
{
	private int zahl;

	public Datenspeicher()
	{
		zahl = 0;
	}

	public Datenspeicher(int neueZahl)
	{
		zahl = neueZahl;
	}

	public int GetZahl()
	{
		return zahl;
```

```
	}


	public void SetZahl(int neueZahl)
	{
		zahl = neueZahl;
	}
}
```

Keine Angst – Ich erwarte nicht, dass man hiervon schon etwas versteht und wir werden diesen – übrigens vollkommen funktionsfähigen Code – Schritt für Schritt kennen lernen. Ich schlage vor den Code auszudrucken, zumal man eine bessere Vorstellung bekommt.
Als erstes fällt die Aufteilung in zwei unterschiedliche Dateien bzw. Klassen auf. Die Namen dürften verständlich sein. Das Prinzip der Anwendung ist folgendes:

- Hauptklasse beinhaltet die Main() - sie ist daher der Einstiegspunkt
- Datenspeicher beinhaltet als einziges Feld eine Zahl. Dazu ein paar Methoden die diese Zahl verändern oder sonstwie manipulieren.

Kapselung

Wir haben bereits gehört, dass Klassen Felder (Variablen) und Methoden haben. Die Methoden sind dafür zuständig, die Felder zu verändern. Sie sind also eine Art Zugriffskontrolle.
Und genau hier spielen die Begriffe private und public aus „Datenspeicher" eine große Rolle.
Wir betrachten das Feld

```
private int zahl;
```

Wir definieren hier ein Feld namens „zahl", das eine ganze Zahl (einen Integer) darstellt als private. Private bedeutet, dass diese Variable von einer anderen Klasse nicht „gesehen" werden kann.
Nun betrachten wir die Methode

```
public int GetZahl()
```

Diese hat, wie wir sehen, einen Rückgabewert (int) und einen Namen (getZahl). Ihr werden keine Parameter übergeben.
Sie ist weiters als public definiert. Das bedeutet, dass andere Klassen diese Methode „sehen" können. Sie können darauf zugreifen.
Nun betrachten wir ihren Rumpf. Dieser besteht aus einem einzigen Ausdruck:

```
return zahl;
```

„return" ist ein Schlüsselwort und liefert einen Wert an die aufrufende Methode zurück. Rufe ich also die Methode GetZahl auf, so bekomme ich den Wert des Feldes „zahl" geliefert.
Nun möchte man denken, dass es eigentlich höchst kompliziert ist, extra zwei Methoden zu schreiben (Setzen und Auslesen des Feldes Zahl), wenn man auch direkt auf das Feld zugreifen kann (man müsste hierzu nur das Feld zahl als public anstelle von private markieren). Nun ja – normalerweise steckt in den Methoden viel mehr Logik, die evtl. auch mehrere Felder zu einem komplexen, sinnvollen Ergebnis kombinieren. Ausserdem ist dies ein überaus wichtiges Prinzip der Objektorientierung, das unbedingt eingehalten werden muss. Wir werden später noch sogenannte Properties kennenlernen, die den Zweck haben, Felder direkt zu verändern und dennoch das Prinzip der Objektorientierung nicht verletzen.

Wir gehen etwas weiter und betrachten nun

```
public void SetZahl(int neueZahl)
```

Mittlerweile sollten wir wissen was diese Methode macht. Wir können diese Methode von einer anderen Klasse aus aufrufen, und ihr eine neue Zahl übergeben (kein Rückgabewert, aber ein Übergabeparameter). Die Zeile

```
zahl = neueZahl;
```

weist nun dem Feld „zahl" der Klasse den übergebenen Wert zu. Dieser kann irgendeinen Wert haben. Einzige Bedingung: Es muss ein Integer – Wert sein.
Damit wären 3/5 der Klasse schon abgearbeitet. Die Frage ist nun, was machen die komischen Methoden ganz zu Beginn der Klasse, die sich von den „normalen" Methoden dadurch unterscheiden, dass sie gar keinen Rückgabetyp haben (nicht einmal void). Ausserdem haben sie beide genau denselben Namen wie die Klasse. Sehr eigenartig, aber nicht mysteriös, denn dies sind die gefürchteten Konstruktoren. Um diese zu verstehen, müssen wir aber erst verstehen, wie denn nun aus einer Vorlage ein konkretes Objekt entsteht (Klassen sind lt. Meiner eigenen Definition ja nur Vorlagen für Objekte). Dazu kommen wir zu einem der wichtigsten Schlüsselwörter der Objektorientierung.

Der Operator „new"

Wir betrachten jetzt die Hauptklasse, genauer gesagt den Inhalt der Methode Main().

```
Datenspeicher ds = new Datenspeicher();
```

Ein ähnliches Ding hatten wir schon einmal

```
Mensch christoph = new Mensch();
```

Wir erinnern uns: Wir hatten eine Vorlage „Mensch" und erzeugten ein neues Objekt namens „Christoph" vom Typ Mensch. Damit hatte Christoph alle Eigenschaften (Haarlaenge, ...) und Methoden (GeheZuFriseur). Unter den Eigenschaften versteht man nun die Felder. Unter Handlungsmöglichkeiten die Methoden.
Nun betrachten wir wieder die obere Zeile:
Wir haben eine Vorlage Datenspeicher (Unsere Klasse Datenspeicher) und generieren ein neues Objekt namens „ds". Nach dieser Zeile gibt es also eine Variable bzw. ein Feld namens ds, das sämtliche Felder (zahl) und Methoden (SetZahl, GetZahl, ...) der Vorlage Datenspeicher kennt. 'ds' ist also dann ein konkretes Objekt, es ist eine Instanz der Klasse Datenspeicher. Das Wort Instanz wird den Leser wohl noch bis in den Schlaf verfolgen. Unter einer Instanz versteht man ein konkretes Objekt, das von einer Vorlage (Klasse) erstellt wurde und jetzt irgendwo im Speicher liegt. Man kann die obige Zeile in etwa folgendermaßen lesen:
Der Datenspeicher namens ds ist ein neues Objekt vom Typ Datenspeicher (also eine Instanz von Datenspeicher).
Ich bitte den Leser sich diesen Absatz mehrmals durchzulesen, da er das wichtigste Prinzip der Objektorientierung darstellt.
Ich habe eine Vorlage, und mache aus dieser ein konkretes Objekt. Ich könnte von der Vorlage auch 100e oder 1000e Objekte machen. Man spricht dann von 100 oder 1000 Instanzen. Alle würden sich durch die Werte der Eigenschaften voneinander unterscheiden. Beispielsweise könnte bei einem die Zahl 3 sein, bei einem anderen 17, beim dritten 425. Aber alle Objekte haben gemein, dass sie ein Feld vom Typ Integer mit dem Namen 'zahl' haben und alle dieselben Methoden haben.
Wie dieser Vorgang vor sich geht, hat uns nicht zu interessieren, wichtig ist nur, dass wir ihn in gewissen Maßen steuern können. Und dies geschieht über die Konstruktoren (to construct – aufbauen).

Konstruktoren

Nachdem wir nun wissen dass 'new' das Schlüsselwort ist, welches eine neue Instanz erzeugt, wollen wir auch aussen wie wir dies steuern können. Dies wird durch Konstruktoren erledigt. Konstruktoren können z.B. verwendet werden, um Feldern gleich zu Beginn einen Wert zu geben, oder gewisse Aufgaben gleich zu Beginn zu erledigen.
Konstruktoren haben keinen Rückgabewert, können jedoch einen oder mehrere Übergabeparameter haben. Sie heissen immer gleich wie die Klasse selbst.

Wir betrachten wieder die Klasse Datenspeicher:

```
public Datenspeicher()

{
```

```
        zahl = 0;

    }


    public Datenspeicher(int neueZahl)

    {

        zahl = neueZahl;

    }
```

Und nochmal zur Wiederholung:
- kein Rückgabewert
- keinen, einen oder mehrere Übergabeparameter
- gleicher Name wie die Klassen

Welcher Konstruktor aufgerufen wird, wird anhand der Signatur entschieden. Wir erinnern uns:
- Name (bei Konstruktoren zwingend gleich wie die Klasse)
- Anzahl und Typ der Übergabeparameter

Nun zum praktischeren Teil:

```
Datenspeicher ds = new Datenspeicher();
```

legt ein neues Objekt an und verwendet den Standardkonstruktor. Dieser hat keine Übergabeparameter. Hat eine Klasse keinen vom Programmierer geschrieben Konstruktor, so wird dennoch dieser Konstruktor automatisch erzeugt. Er tut in dem Fall zwar nichts, aber es muss eine Möglichkeit geben, das Objekt zu erzeugen.
Unsere beiden Konstruktoren haben bereits etwas Funktionalität verliehen bekommen. Der Standardkonstruktor (()) weist dem Feld 'zahl' den Wert 0 zu. Rufe ich also die Zeile

```
Datenspeicher ds = new Datenspeicher();
```

auf, so hat das Objekt ein Feld mit dem Wert '0'.
Alternativ habe ich einen zweiten Konstruktor implementiert. Dieser übernimmt einen Integer als Parameter. Ich könnte ihn beispielsweise mit

```
Datenspeicher ds = new Datenspeicher(7);
```

aufrufen. Der Compiler erkennt, dass hier der alternative Konstruktor verwendet werden soll. Er ruft also

```
public Datenspeicher(int neueZahl)
```

```
    {
        zahl = neueZahl;
    }
```

auf. Dieser Konstruktor weist dem Feld 'zahl' den übergebenen Wert (7) zu. Wir haben also nach einem Aufruf dieses Konstruktors ein Objekt, das ein Feld namens 'zahl' mit dem Wert '7' hat.

Hier eine kleine Zusammenfassung:

```
Datenspeicher ds = new Datenspeicher(); // 'zahl' ist danach 0

Datenspeicher ds = new Datenspeicher(9); // 'zahl' ist danach 9

Datenspeicher ds = new Datenspeicher(0.4); // Fehler – weil 0.4 ist kein Integer

Datenspeicher ds = new Datenspeicher(1,5);

// Fehler – es gibt keinen Konstruktor, der 2 Parameter übernimmt
```

// REVIEW BEFORE PUBLISH
Um das Zerstören eines Objektes müssen wir uns in C# nicht mehr kümmern. Das übernimmt eine Einrichtung namens Garbage Collector. Den Garbage Collector (GC) kann man sich als Objekt vorstellen, welches durch den Speicher scannt und Objekte die nicht mehr benötigt werden, löscht. Wir können bisweilen davon ausgehen, dass alle Objekte die wir erzeugen, irgendwann automatisch gelöscht werden. Wir werden später sehen, dass dies bei gewissen Dingen nicht der Fall ist, das braucht uns jetzt jedoch noch nicht zu interessieren. Ebenso wenig interessieren uns die Destruktoren. Diese sind die Gegenstücke zu den Konstruktoren und werden von uns vernachlässigt.
// END REVIEW

Fragen:
- Welchen Zweck hat ein Konstruktor
- Welchen Zweck hat der Operator 'new'
- Wie wäre eine Methode definiert, der 2 Integer Werte übergeben werden und die einen string zurückliefert.

Aufgaben:
- Schreibe zwei Klassen: #1 ist die Hauptklasse – siehe oben, #2 kennt eine Methode namens GetName, die ein Feld namens 'name', das ein String ist, zurückliefert. Der zurückgegebene String wird dann über Console.WriteLine ausgegeben.
- Schreibe zu obigem eine Methode namens SetName. Ein String soll mittels ReadLine eingelesen werden können, dieser wird dann über die Methode SetName in 'name' gespeichert. Danach soll er wieder ausgegeben werden.

Tag 4

Schleifen und Bedingungen – Revisited

Wir haben am zweiten Tage bereits Schleifen und Bedingungen kennengelernt. Wer sich nicht mehr erinnert (was eher schlecht ist ;)) - for und if waren die Schlüsselwörter.

Wir werden nun zwei weitere Vertreter dieser beiden Gattungen kennenlernen – while und switch.

Wie bereits das Wort erahnen lässt – handelt es sich bei while um eine Schleife und bei switch um eine Art Bedingung.

Wieder einmal ein Stück Code:

```
int x = 0;

while (x < 7)

{

	Console.WriteLine(x);

	x++;

}
```

Es lässt sich beinahe erahnen was hier geschieht: Solange x kleiner als 7 ist, wird x ausgegeben, danach um eines erhöht.

Was auch hier unbedingt beachtet werden muss: Wird auf Gleichheit überprüft, so müssen zwingend doppelte == verwendet werden.

Dasselbe Konstrukt hat auch eine andere Schreibweise:

```
do

{

	Console.WriteLine(x);

	x++;

} while (x < 7);
```

Der einzige Unterschied der hier besteht ist der, dass die zweite Version mindestens einmal durchlaufen wird – unabhängig vom Wert der Variablen. Die Überprüfung findet erst am Ende statt.

Man spricht hier von Kopf- und Fußgesteuerten Schleifen. Variante 1 ist eine Kopfgesteuerte Schleife, da hier die Bedingung am Kopf ist – sie wird als erstes überprüft.

Es gibt noch eine weitere Art von Schleife, die foreach – Schleife. Diese ist jedoch sehr C# - spezifisch und wird von mir erst später – wenn überhaupt – behandelt.

Angenommen, ich habe eine if – Abfrage, die je nach Status einer Variable entscheiden soll, was geschieht, so funktioniert dies wunderbar mit einer if. Gibt es jedoch mehrere Möglichkeiten, so bietet sich hierfür switch an. Man spricht hier von einer Mehrfachverzweigung.

```
public static void Main(string[] args)

{

	Console.WriteLine("Bitte Farbe eingeben");

	string s = Console.ReadLine();

	switch (s)

	{

		case "Blau":

		{

			Console.WriteLine("RGB Wert ist 0000FF");

			break;

		}

		case "Rot":

		{

			Console.WriteLine("RGB Wert ist FF0000");

			break;

		}

		case "Hellgruen":
```

```
            case "Gruen":
            {
                  Console.WriteLine("RGB Wert ist 00FF00");
                  break;
            }
      }
}
```

Was hat das jetzt wieder zu bedeuten. Ganz einfach. Überprüft wird die Variable „s". Ist sie „Rot", so springt das Programm zu dem Abschnitt bei dem case „Rot" steht. Das break danach bedeutet dass er aus der switch Anweisung springen soll und ist obligatorisch. Ein break darf nur weggelassen werden, sofern ein Case keine Anweisung enthält. Dies ist beispielsweise bei Hellgruen der Fall. In diesem Fall wird der nächste Case Block ausgeführt (Grün). Auf diese Art und Weise können mehrere Case's denselben Block ausführen. Ich gehe mal davon aus, dass dies nichts ist, was sich durch ein wenig Übung und Probieren verfeinern lässt. Switch – Anweisungen lassen sich wie alle übrigen Konstrukte (also auch for / while / if / ...) ineinander verschachteln. Dabei wird auch klar, warum das Einrücken besonders bei komplexeren Strukturen sehr wichtig wird. Es fördert die Lesbarkeit eines Problems.
Oft stösst man in Code auf Konstrukte wie:

```
if (bedingung == true)
      Console.WriteLine(„true");
else
      Console.WriteLine(„falsch");
```

Dem aufmerksamen Leser dürfte wohl aufgefallen sein, dass hier die geschwungenen Klammern fehlen. Dies ist erlaubt, solange nur ein einziger Ausdruck folgt. Allerdings sollten immer Klammern vorgesehen werden. Beispielsweise kann es passieren, dass in einer späteren Phase des Projekts noch etwas verändert werden muss und auf einmal werden aus einem Ausdruck zwei, was unweigerlich zu entweder einem Compiler – Fehler oder – was noch schlimmer ist – zu einem unerwünschten Verhalten führt, weil der Compiler die zweite Zeile nicht als Teil der if – Bedingung interpretiert. Diese Schreibweise gilt für alle bisher gelernten Konstrukte.

Coding Conventions Rule #4: Bei sämtlichen Anweisungen sollten Klammern verwendet werden, um ein späteres Fehlverhalten auszuschliessen.

Das nächste Kapitel führt uns in wohl einen der verhasstesten Bereiche der Programmierung – besonders was Anfänger anbelangt.

Arrays

Arrays sind eine Art Felder von Werten. Man könnte sie sich als eine Art Tabelle vorstellen. Diese kann ein, zwei aber auch mehrdimensional sein. Jeden Wert einer zweidimensionalen Tabelle beispielsweise kann man mit zwei Indizes eindeutig charakterisieren. So ist dies auch bei Arrays.

Was hier ganz wichtig ist, ist die Tatsache, dass man mit dem zählen mit 0 beginnt. Ein Array mit 10 Werten, spreche ich also über die Indizes 0 bis 9 an und nicht von 1 bis 10.

Ich kann beispielsweise ein Array erzeugen – diesmal ein eindimensionales – welches 10 Werte vom Typ Integer beinhalten kann. Diese könnte ich mit den Primzahlen füllen. Aussehen würde das Ganze folgendermaßen:

```
int[] primzahlen = new int[10];

primzahlen[0] = 1;

primzahlen[1] = 3;

primzahlen[2] = 5;

primzahlen[3] = 7;

primzahlen[4] = 11;

primzahlen[5] = 13;

primzahlen[6] = 17;

primzahlen[7] = 19;

primzahlen[8] = 23;

primzahlen[9] = 29;
```

Die eckigen Klammern kennzeichnen ein Array, die Zahl in den eckigen Klammern die Größe des Arrays. Einem Array muss immer beim instanzieren (new) die Größe mit angegeben werden. Es ist nicht möglich ein Array mit variabler Größe zu erstellen (es gibt jedoch andere Klassen, die dies ermöglichen).
Wozu das nun gut ist mag man sich fragen. Angenommen ich würde ein Programm schreiben, das mir die Primzahlen ausgibt und sie danach zusammenzählt.
Man kann nun beispielsweise mit einer for – Schleife durch das Array springen und alle Primzahlen ausgeben.

```
for (int i = 0; i < 10; i++)

{

	Console.WriteLine(primzahlen[i]);

}
```

Das Ganze ist keine Hexerei. Bei jedem Durchlauf wird i um eines erhöht und somit der Index des Arrays erhöht. Danach wird die Primzahl an dieser Stelle ausgegeben. Im ersten Durchlauf würde er also primzahl[0] nehmen -> an dieser Stelle steht 1, im zweiten Durchlauf primzahl[1] -> hier steht: 3.
Um mehrdimensionale Arrays werden wir uns später noch kümmern. Wir bleiben bis auf weiteres bei den Arrays die wir uns einfach vorstellen können – also beim eindimensionalen.
Die .NET Laufzeitumgebung stellt für ein Array mit der Klasse System.Array (Die Klasse Array ist im Namespace, ohne das „using System;" in der obersten Zeile würde er die Klasse Array nicht kennen) bereits einige Methoden bereit. Gibt man beispielsweise in SharpDevelop (#Develop) das Wort „primzahlen." ein, so erhält man eine Liste mit Methoden, die auf dieses Objekt angewandt werden können.
Gibt man „Array." ein, so kann man auf die statischen (wir werden noch darauf zu sprechen kommen) Methoden der Klasse Array zugreifen. Dazu gehören beispielsweise Sort (Sortieren), Reverse (Umdrehen) und IndexOf (sucht den Index eines bestimmten Objektes).
Aber auch darum wollen wir uns jetzt noch nicht kümmern. Wir bleiben bei der Grundidee: Ein Array ist eine Liste von Werten, die bequem in einer Schleife durchgegangen werden können. Wir können auf jeden Wert über einen Index zugreifen.

Gültigkeit von Variablen

Wie intuitiv wohl klar sein dürfte, darf ich eine Variable nicht irgendwo definieren und an einer vollkommen anderen Stelle wieder verwenden.
Angenommen sei folgende Klasse mit zwei Methoden.

```
using System;

namespace DefaultNamespace

{

	class MainClass

	{

		public static void Main(string[] args)

		{
```

```
            TestClass myClass = new TestClass();
        }
    }

    class TestClass
    {
        private string sKlasse;

        public TestClass()
        {
            Input();
            Output();
        }

        public void Input()
        {
            sKlasse = Console.ReadLine();
        }

        public void Output()
        {
            string sKlasse = "Hallo";
            Console.WriteLine(sKlasse);
            Console.WriteLine(this.sKlasse);
        }
    }
}
```

Wird man dieses Programm ausführen so wird wider Erwarten nicht zweimal das Eingegeben ausgegeben, sondern zwei unterschiedliche Dinge.
Aber gehen wir das Programm von Anfang an durch:

- Wir haben eine Klasse namens MainClass, deren einziger Zweck es ist, ein Objekt vom Typ „TestClass" zu erstellen – es also zu instanzieren.
- Wir haben eine Klasse namens TestClass, die ein Feld hat, nämlich „sKlasse".
- Diese Klasse hat einen Konstruktor, der beim Erzeugen des Objektes aufgerufen wird und zuerst die Methode Input() und dann die Methode Output() aufruft.
- Auffallend ist, dass wir zweimal eine Variable namens sKlasse definieren. Wider Erwarten, haben diese nun nicht denselben Wert.

Das Feld sKlasse steht innerhalb der Klassendefinition und ist daher für die gesamte Klasse gültig. Die Variable sKlasse befindet sich in der Methode Output und ist daher nur hier gültig. Ich habe also zwei gleichnamige Variablen, die aber eigentlich unterschiedlich sind. Man spricht hier von einem Scope – einem Gültigkeitsbereich. Nun habe ich in einer Methode zwei Variablen, beide strings, beide haben den gleichen Namen. Wie kann ich nun gezielt auf eine zugreifen. Die Klassenvariable (das Feld) wird hierbei von der lokalen Variable überschrieben. Greife ich also auf sKlasse zu, so bekomme ich den Wert, der in der Methode Output der Variable sKlasse der Methode Output zugewiesen wurde.
Will ich auf die – weiter „unten" liegende Variable zugreifen, so kann ich mir mit dem Schlüsselwort „this" behelfen. „this" ist ein Zeiger auf das aktuelle Objekt. Da die Variable sKlasse dem Objekt zugehört, kann ich also auf die unter der lokalen Variablen liegende Klassenvariable zugreifen. Dies mag mit nur einem Beispiel etwas verwirrend klingen, man lernt aber recht schnell wofür das Schlüsselwort „this" noch gut sein kann.

Wenn wir schon beim Gültigkeitsbereich sind, so kommen wir auch gleich auf die Schlüsselworte public und private zu sprechen.
„public" kennzeichnet eine Methode oder Klasse als von aussen sichtbar. „private" macht eine Methode (oder auch Klasse) von aussen unsichtbar. Es können beispielsweise auf „private" Methoden nur Methoden zugreifen, die sich innerhalb derselben Klasse befinden.
Dies spiegelt sich nun auch wieder im obigen Beispiel wieder. Wir wiederholen: Kapselung: Ein Feld einer Klasse sollte von aussen nicht zugänglich sein, sondern nur über Methoden ansprechbar sein.
Wir definieren also das Feld selbst als private, die Methoden als public. So ist das Feld selbst von ausserhalb der Klasse nicht „sichtbar". Die Methoden jedoch sind von aussen sichtbar (public) und da sie sich innerhalb derselben Klasse befinden wie das Feld, dürfen sie das Feld ändern. Es gibt noch weitere Schlüsselwörter neben public und private. Dies wären beispielsweise protected oder sealed. Diese spielen jedoch (noch) keine so große Bedeutung.

Damit wir nun unsere public static void Main(..) vervollständigen können, fehlt

noch das Schlüsselwort „static".
Wie wir wissen, kann ich mit dem Schlüsselwort new ein neues Objekt von einer Vorlage erzeugen (instanzieren). Nehmen wir an ich bin eine Bank und habe 1000 Konten. Ich hätte also 1000 verschiedene Objekte vom Typ Konto. Jedes davon hätte einen gewissen Kontostand als Feld und gewisse Methoden.
Man sagt: Jede Instanz einer Klasse hat ihre eigenen Instanzen der Mitglieder (Felder, Methoden).
Wir nehmen nun an es gibt eine Methode zur Umrechnung einer Währung in eine andere. Nun wäre es etwas Speicherverschwendung, 1000 Instanzen von dieser zu haben. Was ich will ist eine Methode, die einmal im Speicher vorkommt und für die gesamte Klasse gültig ist.
Und genau das erledigt das Schlüsselwort static.
Auch die Klasse Main() darf nur ein einziges Mal vorhanden sein – hierfür sorgt das Schlüsselwort static. Das Kontobeispiel:

```
using System;



namespace DefaultNamespace

{

	class MainClass

	{

		public static void Main(string[] args)

		{

			Konto myKonto1 = new Konto(13);

			Konto myKonto2 = new Konto(26);



			Console.WriteLine("Kontostand von Konto 1 " +
                                                myKonto1.GetKontostand());

			Console.WriteLine("Kontostand von Konto 2 " +
                                                myKonto2.GetKontostand());

			Console.WriteLine("Kontostand von Konto 2 in EUR " +
                                        Konto.Umrechnen(myKonto2.GetKontostand()));

		}

	}
```

```
	class Konto
	{
		private int kontostandATS;
		public Konto(int kontostand)
		{
			this.kontostandATS = kontostand;
		}

		public int GetKontostand()
		{
			return this.kontostandATS;
		}

		public static int Umrechnen(int ATS)
		{
			return ATS / 13;
		}
	}
}
```

Wo liegt nun hier der Hase begraben. Wir erzeugen zwei Konten mit unterschiedlichem Guthaben. Wir benutzen hierfür einen Konstruktor, dem gleich ein Wert mit auf den Weg gegeben wird. Wir benutzen im Konstruktor das Schlüsselwort this, mit dem wir auf die Klassen-interne Variable zugreifen können. Hier wäre es nicht zwingend notwendig, da die beiden Variablen unterschiedliche Namen haben. Meiner Meinung erleichtert es jedoch die Lesbarkeit wenn für Felder (für eine ganze Klasse gültige Variablen) immer this. vorangeschrieben wird.

Die Zeile:

```
Console.WriteLine("Kontostand von Konto 2 " +  myKonto2.GetKontostand());
```

dürfte eigentlich kein Problem darstellen. Im Endeffekt geben wir einen String

aus, der aus einem String und einem Integer zusammengesetzt wird. GetKontostand liefert uns ja bekanntlich einen Integer Wert zurück (siehe Definition der Methode GetKontostand). Interessanter ist die Zeile:

```
Console.WriteLine("Kontostand von Konto 2 in EUR " + Konto.Umrechnen
(myKonto2.GetKontostand()));
```

Wir benutzen hier eine Methode, die nur ein einziges Mal existiert und für alle Instanzen einer Klasse gültig ist – nämlich Umrechnen. „Umrechnen" übernimmt laut Methodendeklaration einen Integer und liefert einen Integer zurück.
Von dem Ganzen sollte man sich jedoch nicht gleich verwirren lassen – man benötigt anfangs hauptsächlich nicht – statische Methoden (abgesehen von der Main()). Und wenn man einmal eine statische Methode braucht, so wird man sich an meine Wort erinnern und es ausprobieren und draufkommen, wozu dies alles gut sein kann.

Fragen:

- Wozu ist die Methode Main() als static deklariert.
- Schreibe ein Programm mit einer for – Schleife und beobachte, wo die Zählvariable gültig ist.

```
Beispiel # 1:

int i = 0;
for (i = 0; i < 10; i++)
{
// tue etwas
}
// ist i hier noch gültig ?

Beispiel #2:
for (int i = 0; i < 10; i++)
{
// tue etwas
}
// ist i hier noch gültig ?
```

Aufgaben:

- Schreibe ein Programm, das die ersten 5 Primzahlen in einem Array speichert, sie danach der Reihe ausgibt und zum Schluss die Summe der Primzahlen ausgibt.
- Schreibe ein Programm, das alle Zahlen von 0 bis 100 durchgeht und alle durch 7 teilbaren Zahlen ausgibt. Hinweis: Der Modulo – Operator (%) liefert immer den Rest einer Bruchrechnung (17 % 5 = 2 -> 17/5 = 3 Ganze + 2 Rest, der Modulo Operator liefert hier 2)
- Schreibe ein Programm, das den Benutzer solange nach seinem Namen

fragt, bis dieser Christoph eingibt.

Tag 5

Vererbung – Revisited

Wie wir evtl. bereits wissen, können Objekte Eigenschaften von anderen Objekten erben.
Sprich: Ich kann eine Klasse Tier generieren, deren Eigenschaften (Füsse, Augen, Geschlecht, ...) und Methoden (Friss, Stirb, Lauf, Gehe, ...) allen Tieren zugrunde liegt. Aufbauend auf diesen Eigenschaften, kann ich dann erweiterte Tiere generieren. Man könnte auch sagen spezialisierte Versionen von Tieren, wie z.B.: Schwein, Hase, Giraffe. Ein anderes Beispiel wäre das unten angegebene:

```
001 using System;
002
003 namespace DefaultNamespace
004 {
005   class MainClass
006   {
007         public static void Main(string[] args)
008         {
009               Console.WriteLine("Hello World!");
010
011               // Wir erzeugen ein Fahrzeug
012               Fahrzeug fz = new Fahrzeug();
013               // Alle Daten ausgeben
014               Console.Write("Das Fahrzeug fährt " + fz.MaxSpeed);
015               Console.WriteLine(" auf " + fz.AnzahlRaeder + " Rädern");
016
017               Auto at = new Auto();
018               Console.Write("Das Auto fährt " + at.MaxSpeed);
019               Console.WriteLine(" auf " + at.AnzahlRaeder + " Rädern");
020
021               Motorrad mt = new Motorrad();
022               Console.Write("Das Motorrad fährt " + mt.MaxSpeed);
023               Console.WriteLine(" auf " + mt.AnzahlRaeder + " Rädern");
024
025               Console.WriteLine(at.Alter);
026               at.HabeUnfall();
027               Console.WriteLine(at.Alter);
028
029         }
030   }
031
032   public class Fahrzeug
033   {
034         protected int anzahlRaeder;
035         protected int anzahlBeifahrer;
036         protected int alter;
037         protected int maxSpeed;
038         protected string fahrerName;
039
040         // Konstruktor - Wird jedesmal aufgerufen, wenn ein Fahrzeug erzeugt
wird
041         public Fahrzeug()
042         {
```

```
043               anzahlRaeder = 0;
044               anzahlBeifahrer = 0;
045               alter = 0;
046               maxSpeed = 50; // Wir haben ein Moped - Fahrzeug ;)
047               fahrerName = "";
048         }
049
050         public void HabeUnfall()
051         {
052               alter = 200; // Restwert geht staaaaark nach unten
053         }
054
055         public int AnzahlRaeder
056         {
057               get
058               {
059                     return anzahlRaeder;
060               }
061         }
062
063         public int MaxSpeed
064         {
065               get
066               {
067                     return maxSpeed;
068               }
069               set
070               {
071                     maxSpeed = value;
072               }
073         }
074
075         public int Alter
076         {
077               get
078               {
079                     return alter;
080               }
081         }
082   }
083
084   public class Auto : Fahrzeug
085   {
086         public Auto()
087         {
088               anzahlRaeder = 4;
089               anzahlBeifahrer = 4; // wir haben kein Cabrio
090               alter = 10;
091               maxSpeed = 250; // Der Audi TT den phj getestet hat
092               fahrerName = "Christoph";
093         }
094   }
095
096   public class Motorrad : Fahrzeug
097   {
098         public Motorrad()
099         {
100               anzahlRaeder = 2;
101               anzahlBeifahrer = 1;
102               alter = 3;
103               maxSpeed = 200;
```

```
104                 fahrerName = "Ich - Irgendwann";
105         }
106   }
107 }
```

So. Ich gehe mal davon aus, dass der Leser jetzt erschlagen ist. Hier sind viele neue Dinge enthalten, die aber alle von höchster Bedeutung sind. Wir betrachten die Klasse Auto und wiederholen.

- Felder eine Klasse sind als private definiert, man kann nicht von ausserhalb darauf zugreifen
- Methoden sind public, man kann von „aussen" auf sie zugreifen, sie ändern die Felder

Warum private, wenn da doch protected steht, was redet er denn jetzt für einen Blödsinn? Dies mögen sich nun manche Fragen und das ist auch gut so.
Das Ganze kann ich natürlich auch erklären:
„private" kennzeichnet Felder oder Methoden als von aussen nicht sichtbar.
„public" kennzeichnet Felder oder Methoden als von aussen sichtbar
„protected" kennzeichnet Felder oder Methoden als von aussen nicht sichtbar sind. Klassen, die von der Klasse, in der ein Feld oder eine Methode als protected deklariert wurde, abgeleitet wurden, können auf dieses Feld jedoch zugreifen.

Zur Erklärung das obere Programm:
Wir haben eine Klasse Fahrzeug, die sämtliche Felder kennt. Diese hat einen Konstruktor, der die Werte mit (nicht ganz sinnvollen) Standardvorgaben belegt. Weiters haben wir zwei Klassen Auto und Motorrad, welche von Fahrzeug abgeleitet werden. Sie erben also sämtliche Felder und Methoden von Fahrzeug. Nun haben wir aber das Problem, dass die Felder ja eigentlich als „private" deklariert sind und somit nur von der Klasse Fahrzeug eingesehen werden kann. Würde man sie als „public" definieren, so würde das alles zwar auch funktionieren, aber man würde das Prinzip der Kapselung (Methoden ändern die Felder einer Methode) verletzen – was in der Objektorientierung nicht erwünscht ist. Und genau zu diesem Zweck gibt es das Schlüsselwort „protected". Alle Klassen, die als Basisklasse die Klasse Fahrzeug haben, dürfen auf die Felder direkt zugreifen. Für alle übrigen Klassen ist dies nicht möglich. Versuche ruhig einmal die Felder als private zu deklarieren. Der Compiler wird sich melden, dass er nicht auf die Felder zugreifen darf.
Daher: „protected" Felder erlauben den Zugriff von abgeleiteten Klassen und der eigenen Klasse selbst. Alle anderen Klassen haben keinen Zugriff auf das Feld.

In Zeile 50 habe ich auch eine Methode eingebaut (als „public" deklariert). Angenommen, man würde den Restwert des Fahrzeuges nach dem Alter berechnen, so entspräche das Alter nach einem Unfall 200 Jahre. Wäre also nichts mehr wert. Methoden kann man immer an den Klammern nach dem

Methodennamen erkennen (Würde die Methode einen Parameter übernehmen, so würde dieser in diesen Klammern stehen; da die Methode jedoch keine hat, sind die Klammern eben leer).

Nun stellt sich wieder ein Problem. Ich will eine Variable direkt auslesen. Grundsätzlich ist die Situation klar. Ich bastele eine Methode, welche mir den Wert der Variable zurückgibt. Nicht sonderlich effektiv, denn würde ich die Variable auch setzen wollen, so bräuchte ich noch eine zweite.
Zu diesem Zweck gibt es die sogenannten Properties – Eigenschaften. Bei Properties gelten dieselben Konventionen wie bei Methoden. Erster Buchstabe groß, danach jedes Wort wieder groß. Die zugehörigen Felder kann man gleich schreiben, mit dem Unterschied dass der erste Buchstabe klein ist. So kommt man sich nicht in die Quere, weiss aber trotzdem immer welche Property welcher Eigenschaft zuzuordnen ist.
Wie werden Properties definiert. Der Kopf der Methode ist derselbe wie bei einer Methode. Er beginnt mit dem Schlüsselwort public, danach kommt der Datentyp und dann folgt der Name der Property. Im Gegensatz zur Methode sind hier jedoch keine Klammern, stattdessen folgt eines (oder beide) der Schlüsselwörter get oder set (Zeile 63 ff). Weiters gibt es noch ein spezielles Schlüsselwort value. Wird die Property von aussen gesetzt, so beinhaltet „value" den neuen Wert.
Zu den Properties ein kurzes Beispiel:

```
01 using System;
02
03 namespace DefaultNamespace
04 {
05    class MainClass
06    {
07          public static void Main(string[] args)
08          {
09                Person christoph = new Person();
10                Console.WriteLine(christoph.Alter);
11                Console.WriteLine(christoph.Haarfarbe);
12                christoph.Alter = 33;
13                Console.WriteLine(christoph.Alter);
14          }
15    }
16
17    public class Person
18    {
19          private int alter;
20          private string haarfarbe = "Gruen";
21
22          public int Alter
23          {
24                get
25                {
26                      return alter;
27                }
28                set
29                {
30                      alter = value;
31                }
```

```
32          }
33
34          public string Haarfarbe
35          {
36                get
37                {
38                      return haarfarbe;
39                }
40          }
41    }
42 }
```

Wir haben hier eine Klasse Person mit zwei Feldern (haarfarbe und alter). Diese beiden Felder sind über Properties zugänglich, wobei „Alter" gelesen und gesetzt werden kann, „Haarfarbe" aber nur gelesen werden kann. Würde ich also „Haarfarbe" von aussen einen Wert zuweisen wollen, würde der Compiler dies nicht erlauben.

Man sieht in Zeile 30 auch schön das zuweisen von dem übergebenen Wert. Nach Zeile 12 hat die „Variable" value von Alter den Wert 33; „alter" wird also auf 33 gesetzt.
Bei „get" wird ein Wert mittels return (wie bei Methoden) zurückgeliefert. Von Properties werden wir bei Windows Forms noch ausserordentlich Gebrauch machen.

Ich fasse zusammen: Properties sind eine Vereinfachung um auf Variablen direkt zuzugreifen ohne das Prinzip der Kapselung zu verletzen. Nutzt man Properties, so muss man nicht für jede Variable zwei Methoden schreiben – die Klasse bleibt übersichtlich.
Setzen erfolgt mittels dem implizit gegebenen „value". Properties können nur gelesen oder gesetzt werden, oder beides.

Wir kehren zum grösseren Programm zurück:
Wir haben, wie wir bereits fest gestellt haben 2 Klassen, die beide von Fahrzeug abgeleitet werden. Wenn ich also ein neues Objekt vom Typ Auto erzeuge, so ist beinhaltet dies auch alle Eigenschaften von Fahrzeug. Wie wir wissen kann man im Konstruktor einer Klasse (das ist die Methode, die aufgerufen wird, wenn ein Objekt instantiiert wird) den Feldern gewisse vordefinierte Werte geben. Nun hat aber ein Objekt vom Typ Auto 2 Konstruktoren. Einmal einen von der Basisklasse, der Klasse Fahrezug und einen von der Klasse Auto selbst. Welcher der beiden wird nun aufgerufen wenn ich ein neues Objekt vom Typ Auto erstelle.
Richtig: Beide. Zuerst der Konstruktor der Basisklasse und dann der der abgeleiteten Klasse. Das folgende Beispiel veranschaulicht dies:

```
using System;

namespace DefaultNamespace
{
      class MainClass
      {
```

```
		public static void Main(string[] args)
		{
			Auto at = new Auto();
		}
	}

	class Fahrzeug
	{
		public Fahrzeug()
		{
			Console.WriteLine("Konstruktor der Basisklasse");
		}
	}

	class Auto : Fahrzeug
	{
		public Auto()
		{
			Console.WriteLine("Konstruktor der Abgeleiteten Klasse");
		}
	}
}
```

Startet man das Programm so sieht man genau was denn nun genau aufgerufen wird. Wichtig ist hier eben wieder, dass die Klasse Auto von Fahrzeug abgeleitet ist. Dies ist an dem Doppelpunkt zu erkennen.
Lesen kann man die Zeile folgendermaßen: Die Klasse (class) Auto (Auto) ist abgeleitet von (:) Fahrzeug (Fahrzeug).
Im großen Beispiel kann man erkennen, dass wir den Konstruktor von Fahrzeug für alle anderen Klassen gleich wieder überschreiben. Er wird zwar aufgerufen (Basisklasse vor abgeleiteter Klasse) aber die Werte werden dann sofort vom Konstruktor der abgeleiteten Klasse überschrieben.
Soweit so gut, eigentlich sollte jetzt die obere Klasse vollkommen verständlich sein und wir haben alle Grundlagen gesammelt um unser erstes Fensterchen zeichnen zu können.

Unser erstes Fenster – Hello World (Reloaded)

Wir beginnen in C# ein neues Combine mittels Datei -> Neu -> Combine. Wir wählen als „Schablone" eine Windows Anwendung aus und nennen sie HelloWorld2. Nach einem Klick auf erstellen, erhalten wir folgenden Sourcecode:

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

namespace DefaultNamespace
{
	public class MainForm : System.Windows.Forms.Form
	{
		public MainForm()
		{
			InitializeComponent();
		}
```

```
            [STAThread]
            public static void Main(string[] args)
            {
                  Application.Run(new MainForm());
            }

            private void InitializeComponent()
            {
                  this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
                  this.ClientSize = new System.Drawing.Size(292, 266);
                  this.Text = "MainForm";
                  this.Name = "MainForm";
            }
      }
}
```

Ich habe mir erlaubt sämtliche Kommentare wegzulöschen, was normalerweise ein unerwünschtes Verhalten später mit sich bringen kann. Aus Übersichtsgründen für den ersten Versuch ist dies jedoch zweckdienlich.
Was fällt auf:

- Wir haben 2 using Statements
- Wir haben eine Klasse MainForm, die von der Klasse Form abgeleitet ist
- Wir haben einen Konstruktor MainForm(), der eigentlich nur eine Methode namens InitializeComponent aufruft.
- Wir haben einen Eintrittspunkt – die Methode static void Main()
- Wir haben eine Methode namens InitializeComponent, die auffallend oft das Wort this enthält.

Gehen wir es der Reihe nach durch:
Wir kennen bereits das using Schlüsselwort. Es zeigt an, dass wir einen Namensraum, der gewisse Objekte behält nutzen wollen (bzw. eigentlich die Objekte in diesem Namensraum). Wir wissen auch dass Namensräume verschachtelt werden können. Der Namensraum System kann also einen weiteren Namensraum beinhalten, und dieser wieder einen. Genau das ist hier der Fall. Wir nutzen den Namensraum Forms aus dem Namensraum Windows. Windows selbst ist wieder ein Namensraum in System.
Wie haben eine Klasse MainForm die von Form abgeleitet ist. Genauer gesagt von System.Windows.Forms.Form. Die Klasse Form ist im Namensraum System.Windows.Forms enthalten.
Was ist nun so eine „Form". Wie sieht das aus. Dazu starten wir das Programm einmal. Es sollte jetzt – sofern nichts schiefgegangen ist ein Windows Fenster auftauchen. Genau DAS ist eine Form. Wie belassen es einmal dabei und merken uns nur, dass eine Form gewisse Eigenschaften hat. Sie hat beispielsweise eine gewisse Größe und eine gewisse Location, also einen gewissen Punkt an dem sie gezeichnet wird. Weiters hat sie einen „Namen" (name). Dieser Name ist so etwas wie eine Variable über den sie bzw. ihre Eigenschaften angesprochen werden (genau wie wir at.Alter angesprochen haben, nur dass das „at" nun eben unsere Form ist und Alter z.B. die Eigenschaft Size (also die Größe)). Als letzte wichtige Eigenschaft hat sie eine

Eigenschaft namens „Text", die das Feld ganz links oben kennzeichnet (Momentan steht dort: MainForm).

Im Konstruktor wird eine Methode namens InitializeComponent aufgerufen, die wir uns ansehen. Wir betrachten aus dieser nur die letzten beiden Statements. „this" verweist wie wir wissen auf die aktuelle Klasse, also auf die Form selbst. Wie wir ebenfalls wissen, hat eine Form Eigenschaften namens Name und Text. Genau an dieser Stelle werden sie zum ersten Mal gesetzt. Man kann sich die InitializeComponent wie eine Methode vorstellen, die die Grundeigenschaften einer Form (und der Controls auf ihr) setzt. Was Controls sind, lernen wir noch. Ehrlich gesagt, ist es höchst mühsam, sämtliche Eigenschaften von Hand zu setzen, darum ist dieser Bereich auch standardmässig ausgeblendet (mit dem + am Anfang einer Zeile lässt sich ein Bereich ein oder ausblenden) und #Develop bietet ein komfortables Werkzeug, diese Eigenschaften zu ändern.

Kurzer Einschub: Es gibt gewisse spezielle Tags – also Codefragmente, die es erleichtern Code zu strukturieren – dazu gehören natürlich Kommentare aber auch die Tags wie Region. Der Bereich zwischen einem „#region Name des Bereichs" und einem „#endregion", jeweils ohne die Anführungszeichen, kann von einer Entwicklungsumgebung ausgeblendet werden. Dies ist natürlich von der Entwicklungsumgebung abhängig (auch in Visual Studio ist es so).

So – weiter im Text: Wir suchen nun unterhalb des Quellcodes einen Button der mit „Design" beschriftet ist und klicken auf diesen. Wir sehen nun die Form als Graphikobjekt und rechts daneben die Eigenschaften derselben schön in einer Reihe aufgelistet.
Was macht nun dieser graphische Editor. Er macht nichts anderes als dass er Zeilen zu unserem Quelltext hinzufügt oder diese ändert. Würde ich beispielsweise die Eigenschaft „Text" ändern so würde er die Zeile this.Text = „MainForm" durch die Zeile „this.Text = „Mein neuer Text"" ändern. Er funkt also in unserem Sourcecode herum. Auf jeden Fall würde ich empfehlen, solche Eigenschaften, die zu Beginn des Programms fest gelegt werden können, vom Designer festlegen zu lassen. Der Designer ist – was den Quelltext anbelangt recht pingelig und reagiert auf kleine Veränderungen gern damit, dass er nicht mehr korrekt arbeitet.
Wir suchen nun rechts bei den Eigenschaften die Eigenschaft „Text" und ändern diese in „Hello World 2".
Wir starten wieder das Programm und sehen, dass sich die Zeile ganz oben von MainForm auf „Hello World 2" verändert hat.
Die Eigenschaften, die fett gedruckt sind, sind die, die im Quelltext von den Standardeigenschaften abweichen. Jedes Objekt hat ja von Haus aus seine Eigenschaften zu Beginn mit mehr oder weniger sinnvollen Werten belegt, die wir überschreiben können (MainForm ist unsere abgeleitete Klasse, Form ist die Klasse von der wir ableiten -> zuerst wird der Konstruktor von Form aufgerufen, danach der von MainForm).
Unser Eintrittspunkt – die Methode Main() ruft nur eine weitere statische

Methode namens Application.Run auf, der eine neue Instanz unserer Form übergeben wird. Dies ist der Standard, wie ein solches Form – basiertes Programm gestartet wird.

Ein erstes Control

So ein Fensterchen ist schon was tolles, zumal man es vergrößern, verkleinern, maximieren, minimieren und weiss der Teufel was alles noch kann. Aber irgendwas fehlt da definitiv noch. Nämlich Controls. Controls heissen übersetzt auch Steuerelemente. Es gibt verschiedene Arten von Controls. Beispielsweise TextBoxes (hier lässt sich ein Text eingeben) oder Buttons (ein Knopf zum klicken).
Wir werden als erstes das Einfachste aller Controls verwenden – ein sogenanntes Label. Dazu wählen wir ganz Links unten den Karteireiter „Tools" und dort den Unterbereich „Windows Forms". Das zweite Objekt ist eh bereits jenes, welches wir suchen. Ein Klick auf das Feld mit dem Label und danach ein Klick auf die Form und schon existiert auf unserer Form das erste Control. Wir markieren dieses mit einem Klick darauf und betrachten die Eigenschaften dieses Controls (rechts, Eigenschaften). Wichtig: Ganz oben muss label1 stehen. Steht dort MainForm, so ist nicht das Control das aktive Objekt sondern die Form. In diesem Fall auf das Control klicken, sodass es aktiviert ist. Es werden immer die Eigenschaften für das aktivierte Objekt angezeigt.
Dieses Control hat nun ebenfalls eine Eigenschaft namens Text, die wir sogleich ändern. Aus „label1" wird „Hallo Welt – Mein erstes Control". Der Effekt sollte nach dem Bestätigen mit Enter direkt auf dem Control zu sehen sein. Wir starten die Applikation und – tataaa – unser erstes Control.
Wir betrachten nun wieder den Quelltext und davon wieder den Bereich InitializeComponent. Wir sehen, dass der Designer uns nun wieder etwas Quelltext eingefügt hat. Er greift dabei z.B. auf das Objekt this zu (die Form) und auf der Form auf das Control namens „label1". Er verändert die Location, und den Text (und sonstiges, das uns nicht interessiert).
Theoretisch wäre es natürlich auch möglich den Quelltext manuell einzufügen – aber seien wir ehrlich – warum? - es ist so definitiv einfacher und weniger fehleranfällig.
Ausserdem hat der Designer direkt unter der Klassendefinition eine Variable eingeführt (das Control) die für die gesamte Klasse gilt. Ich kann also von jeder Methode aus auf das Control zugreifen und das Control existiert solange, wie auch die Klasse existiert.
Das wars auch schon für heute – Viel Spass beim ausprobieren.

Fragen:

- Wofür steht protected
- Wann verwende ich private und wann public
- Was sind Controls
- Was ist eine Form

- In welchem Namespace finde ich eine Form
- Was versteht man unter Eigenschaften
- Was sind Properties
- Was sind Felder

Aufgaben:

- Schreibe ein Programm, das eine Klasse Mensch und eine Klasse HomoSapiens kennt. Eine Klasse muss von der anderen abgeleitet werden (welche ist sinnvoll)
- Jeder Mensch hat eine gewisse Anzahl an Beinen und Händen. Diese soll durch einen Unfall geändert werden können (Methode).
- Schreibe eine Windows – Anwendung, die eine Form zeichnet, die automatisch den ganzen Windows Bildschirm bedeckt (Hinweis: Spiele dich mit den Eigenschaften)

Tag 6

Ahhh, Fehler

Wir haben ja bereits gesehen was passieren kann, wenn ein Fehler auftritt, der von uns nicht abgefangen wird.

Wir erinnern uns an dieses Beispiel:

```
Console.WriteLine("Bitte eine Zahl eingeben");
string str;
str = Console.ReadLine();
int i;
i = int.Parse(str);

if (i > 10)
{
      Console.WriteLine("Zahl ist Grösser als 10");
}
else
{
      Console.WriteLine("Zahl ist kleiner als 10");
}
```

Solange wir eine Zahl eingeben, funktioniert das Programm vollkommen problemlos. Wir starten dieses Programm nun wieder und geben einmal einen Text ein (schliesslich kann man von einem User nicht erwarten, dass er nur das eingibt, was er soll).

Das gesamte Programm stürzt ab und das wars. Eigentlich nicht das was wir wollen.

Aber warum? Wir betrachten hierzu den MSDN Artikel über Int.Parse. Dazu gehen wir auf http://msdn.microsoft.com und geben im Suche Feld „int.Parse" ein. Wir nehmen den ersten Link (Parse Method) und wählen dann links Parse Method (string). Da finden wir im Text einen Bereich „Exceptions". Und genau dies sind unsere Fehler, die auftreten können.
FormatException ist eine Klasse die von System.Exception abgeleitet ist. D.h. wenn ich System.Exception abfange, fange ich alle davon abgeleiteten Klassen ab.
Aber was bedeutet das nun. Gewisse Methoden können unter bestimmten Umständen nicht weitermachen, das Ergebnis wäre vollkommen verfälscht und da das Programm trotzdem weitermacht, würde der Benutzer vollkommen falsche Ergebnisse bekommen. Dies sind meist Dinge, die der Compiler noch nicht zur Compile – Time erkennen kann. Eben weil die Eingabe des Benutzers erst stattfindet nachdem das gesamte Projekt kompiliert wurde. In einem solchen Fall wirft C# eine Exception (die schlechte deutsche Übersetzung würde wohl Ausnahme bedeuten – ich bleibe aber bei der englischen). Dementsprechend müssen wir solche Ausnahmen behandeln. Wir haben 2 Möglichkeiten.

Wir fangen jede einzelne Ausnahme ab (z.b. FormatException) oder wir fangen alle Ausnahmen (System.Exception) ab. Auch wenn die zweite Art eigentlich die unschönere der beiden ist, so ist sie doch einfacher. Natürlich ist es ein schönerer Programmierstil, jede einzelne abzufangen und dementsprechend zu behandeln.
Womit können wir nun bewirken, dass das Programm nicht gleich abbricht sondern uns vernünftige Fehlermeldungen bringt.
Wir formulieren mal einen möglichen Ansatz.
„Versuche, den eingegeben String in einen Int zu parsen. Wenn dies fehlschlägt, so fange die Exception ab".
Und genauso wird es auch realisiert.
Aus dem oben angegebenen wird nun:

```
try
{
      Console.WriteLine("Bitte eine Zahl eingeben");
      string str;
      str = Console.ReadLine();
      int i;
      i = int.Parse(str);

      if (i > 10)
      {
            Console.WriteLine("Zahl ist Grösser als 10");
      }
      else
      {
            Console.WriteLine("Zahl ist kleiner als 10");
      }
}
catch (System.Exception e)
{
      Console.WriteLine("Fehler - Programmende");
}
```

Starten wir dieses Programm nun noch einmal und geben wieder einen Blödsinn ein, so erhalten wir unsere Fehlermeldung und das Programm wird beendet.
Was genau passiert:
Wir befinden uns ganz oben. Wir geben etwas aus und lesen etwas ein. Nun (int.Parse) soll das eingegebene umgewandelt werden. Da dies fehlschlägt, wird einer Fehlermeldung „geworfen". Unser try – Block fängt diese ab und verzweigt sofort in den catch – Block. Die Fehlermeldung wird ausgegeben und da sonst nichts mehr ansteht, wird das Programm beendet.
Wichtig: Der gesamte restliche try – Block wird im Falle eines Fehlers übersprungen und es wird sofort auf den catch – Block verzweigt (Der try – Block wird also nicht fertig abgearbeitet).
Tritt kein Fehler auf, so wird der catch – Block übersprungen. Dem „catch" können wir die „Art" des Fehlers zeigen, für den er verantwortlich ist.

```
catch (ArgumentException e)
```

In diesem Fall fängt er nur „ArgumentExceptions" ab. Tritt aber ein anderer Fehler auf, so führt dies wieder zu einem Programmabsturz. Nur abgefangene Fehler führen nicht zu einem Absturz. Welche Exceptions auftreten können, findet sich in der MSDN zu dem jeweiligen Befehl.

Events – Es ist was passiert

So, nachdem wir nun wissen (sollten) wie man evtl. auftretende Fehler zur Laufzeit abfangen kann, wieder zu einem Thema, welches die Fensterchen – Programmierung sehr stark betrifft: Ereignisse.

Man stelle sich eine Windows Form vor, auf der es einen Knopf gibt. Dieser Knopf wird nun gedrückt, es soll wahrscheinlich irgendwas passieren. Aber wo ist nun die Verknüpfung zwischen dem was passiert und dem Ereignis selbst. Diese Verknüpfung nennt man Event Handler. Wir kümmern uns also um auftretende Events.
Wir erstellen in #Develop eine neue Windows Anwendung und betrachten gleich die Form (Design Knopf unter dem Code Fenster). Aus den Tools ziehen wir einen Button auf die Form. Wir markieren diesen Button nun und betrachten im Fenster wo wir die Eigenschaften gesehen haben den Knopf mit dem gelben Blitz (Ereignisse). Wir klicken darauf und die Eigenschaften verschwinden, dafür erscheinen nun alle möglichen Events die das Objekt (der Button) loswerden kann. Der wichtigste Event eines Buttons ist wahrscheinlich „Click". Mit einem Doppelklick auf das weisse Feld neben „Click" generieren wir automatisch einen Event Handler. Das Code Fenster geht auf und wir fügen folgenden Text ein:

```
MessageBox.Show("Button gedrückt");
```

Wir starten das Programm, und klicken den Knopf – eigentlich nicht schlecht für so wenig Arbeit.
Wie funktioniert das Ganze nun. Wir haben einen Event (den Click Event) und wir haben eine Methode die ausgeführt wird. Wo steckt nun aber der Event Handler – die Verknüpfung zwischen den beiden (auch wenn es auf den ersten Blick so aussieht, es ist nicht der Name der Methode).
Wir wissen dass der gesamte Code für eine Form (wie das Layout aussieht usw.) in der Datei steckt (im standardmässig ausgeblendeten Teil „Windows Forms Designer generated Code". Wir blenden diesen Teil ein und sehen im Abschnitt Button folgende Zeile:

```
this.button1.Click += new System.EventHandler(this.Button1Click);
```

Da ist er. Unser Button „button1" kennt einen Event namens „Click". Zu diesem, fügen wir einen neuen „new" Event Handler hinzu (beachte den Operator +=, dieser bedeutet ja eigentlich: Nimm das was wir schon haben, und füge etwas hinzu), der auf die Methode Button1Click der Form zeigt. Und genau dies ist die Verknüpfung.

Wie das nun genau funktioniert und wie man selbst Events implementieren kann, ist ein fortgeschritteneres Thema und ist vorerst nicht von Bedeutung. Wichtig ist, dass jedes Control (es muss nicht unbedingt ein Control sein, aber hier ist es halt klar ersichtlich) viele Events hat, über die der Programmierer informiert wird, dass etwas passiert ist und er darauf angemessen reagieren soll / kann / muss (je nachdem).
Ausserdem kennt #Develop für jedes Control einen Standard – Event Handler. Ich ziehe nun also noch einen Button auf die Form und doppelklicke auf diesen. Es wird automatisch der Click Event generiert (und nicht einer von den anderen).
In diesen EventHandler tippe ich nun folgendes ein:

```
MessageBox.Show("2ter Button");
```

Starte ich das Programm, so wird für jeden Button der passende Event Handler aufgerufen. Im Endeffekt funktionieren alle Controls ähnlich.

- Schreibe das Programm mit den 2 Buttons so um, dass bei beiden Buttons dieselbe Methode aufgerufen wird (egal auf welchen Button ich klicke).
- Schreibe das Zahleneingabe - Programm so um, dass er so lange fragt, bis eine Zahl eingegeben wird, ansonsten soll er sagen „Bitte eine Zahl eingeben".
- Probiere verschiedene Controls aus (TextBox, Button, Label, HScrollBar). Betrachte die Eigenschaften und Ereignisse und überlege was sie bedeuten.

Appendix A: SharpDevelop

Credits

Ein herzliches Dankeschön an den GH – Forum User 'Dr. Watson'. Er hat mich auf diese frei verfügbare, mächtige Entwicklungsumgebung gebracht.

Installation

Die Installation verläuft Menü – geführt und sollte zu keinen Problemen führen. Unter Code Completion wählt man, dass eine neue Datenbank erstellt werden soll. Die folgenden Schritte sind wieder selbst – erklärend.

Das erste Projekt

Man wählt Datei -> Neu -> Combine. Unter einem Combine (in Visual Studio nennt sich dasselbe „Solution") versteht man eine Zusammenfassung von Projekten, die zusammen ein Programm ergeben.

Wir wählen C# Konsolenanwendung für unsere ersten Projekte und wählen „HelloWorld" als Name. Es wird für uns automatisch das Grundgerüst erstellt. Einzige Änderungen zu unserem bisherigen Gerüst.

- Die Main() hat einen Übergabeparameter bekommen
- Alles wurde in einen Namespace verpackt
- Über dem Code befindet sich ein Kommentarblock.

Im linken Fenster (Projekte) finden wir Referenzen und Ressourcen. Diese beiden können wir solang wir nicht mit bereits bestehenden Komponenten arbeiten weglassen. Darunter finden wir eine Datei AssemblyInfo.cs, die uns vorerst auch noch nicht zu interessieren hat. Zuallerletzt finden wir die Datei main.cs welche unser (erweitertes) Grundgerüst kennt. Eigentlich steht schon ein ganzes Programm vor uns, das wir mittlerweile kennen sollten.
Um es zu kompilieren und zu starten, reicht ein klick auf die grüne Play – Taste. Dies sind die bisher wichtigsten Elemente die wir kennen sollten.

Ein erster kurzer Test

Wir löschen die Zeile Console.WriteLine ... weg und schreiben sie selbst nochmals an. Sobald wir 'Console' fertig geschrieben haben und auf '.' gedrückt haben, erscheint ein Fensterchen mit allen möglichen Methoden und Feldern die das Objekt 'Console' anbietet. Dies ist genau dieselbe Information, wie sie uns auch die MSDN geben würde. Ist das richtige ausgewählt, so genügt ein Enter und die Zeile wird automatisch vervollständigt. Dieses Feature nennt sich Code – Completion. Bitte nicht wundern, wenn man nun auf '(' drückt und sich

wieder ein Fensterchen öffnet. Es handelt sich hierbei um sämtliche Überladungen der Methode WriteLine. Was das genau ist, wird erklärt, wenn es ums Überladen von Methoden geht.
Was evtl. noch interessant ist:
Ganz unten kann man von Projektansicht auf Klassenansicht wechseln. Man bekommt in der Klassenansicht eine gute Information über die hierarchische Struktur des gesamten Combine.
Wenn man den ganzen Baum ausklappt, lautet die Hierarchie wie folgt:
Combine -> Projekt -> Namespace -> Klasse -> Methode(n)