mitp

Ulla Kirch

inklusive CD-ROM

C#

Lernen und professionell anwenden

- Gezielter Lernerfolg durch überschaubare Kapiteleinheiten
- Vollständige Darstellung – Schritt für Schritt
- Konsequent objektorientiert programmieren
- Auf der CD: Microsoft Visual C# 2008 Express Edition, Beispielprogramme und Musterlösungen

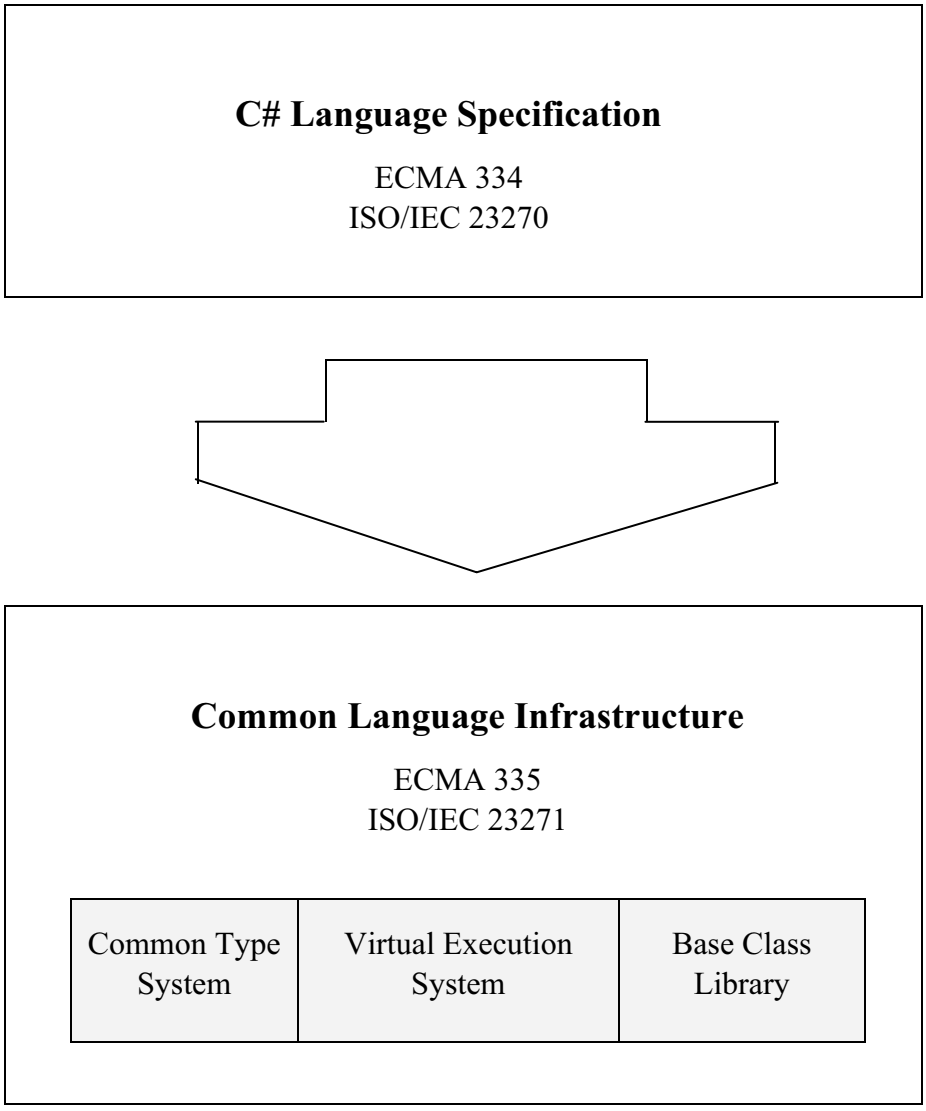
Kapitel 1

Erste Schritte mit C#

Dieses Kapitel beschreibt grundlegende Eigenschaften von C#. Sie werden die Struktur von C#-Programmen und die nötigen Schritte zur Erstellung einer Konsolenanwendung kennen lernen. Anhand von einführenden Beispielen können Sie diese Schritte auf Ihrem System nachvollziehen.

Historisches und Eigenschaften von C#

Standardisierungen



Historisches

Die Programmiersprache C# wurde Ende der 90er Jahre von Anders Hejlsberg und seinem Team bei Microsoft im Rahmen der *.NET Plattform* entwickelt. .Net ist eine Middleware-Plattform, deren zentraler Bestandteil das *.Net Framework* ist. Als einheitliche Serverplattform ermöglicht .Net die Entwicklung verteilter Anwendungen. Portierungen von .NET auf unixoide Systeme wie Linux und Mac OS X bietet das Open-Source-Projekt Mono.

Auf Initiative von Hewlett-Packard, Intel und Microsoft wurde im Jahr 2000 bei der ECMA (European Computer Manufacturers Association) ein Komitee zur Standardisierung von C# gebildet. Ende 2001 wurde eine erste Version der »C# Language Specification« (ECMA 334) verabschiedet. Diese Spezifikation wurde von der ISO (International Standardization Organization) als ISO/IEC-Standard 23270 ratifiziert. In folgenden Jahren wurde der Standard mehrfach aktualisiert und ergänzt.

Parallel dazu wurden wesentliche Teile des .NET Frameworks standardisiert (ECMA 335, ISO/IEC 23271). Die *Common Language Infrastructure*, kurz CLI, spezifiziert ein einheitliches *Typsystem* (Common Type System), eine *Laufzeitumgebung* (Virtual Execution System) und *Klassenbibliotheken* (Base Class Libraries). So wurde die Basis geschaffen, um CLI-konforme Implementierungen von Sprachen wie Basic, SmallTalk, Perl, Python und C++ bei der Entwicklung von Software-Systemen einzusetzen und miteinander interagieren zu lassen.

Eigenschaften von C#

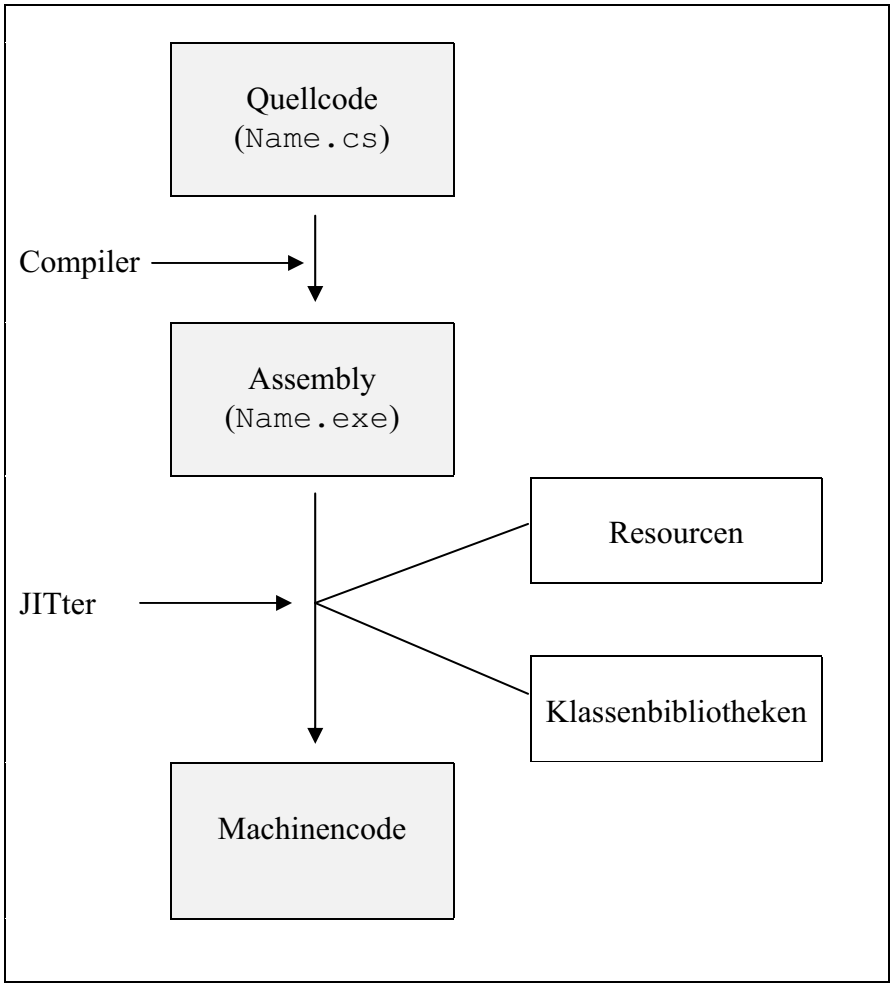
C# ist eine Weiterentwicklung von C und C++. Der Name C# (sprich: C-sharp) ist der Musik entlehnt, wo C# (= cis) der um einen Halbton erhöhte Grundton C ist. Im Gegensatz zu C und C++ ist C# eine rein objektorientierte Programmiersprache. C# unterstützt:

- strenge Typüberprüfungen, Bereichsüberprüfung bei Vektoren und automatische Speicherverwaltung
- die Entwicklung von Anwendungen für unterschiedliche Systeme, vom leistungsstarken Host-basierten System über Handys und PDAs bis zum Embedded System mit eingeschränkter Funktionalität
- die Entwicklung von erweiterbaren, Binärcode-kompatiblen Software-Komponenten zum Einsatz in verteilten Systemen

Damit ist C# eine sehr mächtige und sichere Programmiersprache, die trotz ihrer umfangreichen Funktionalitäten einfach zu handhaben ist.

Entwicklung eines lauffähigen C#-Programms

Managed Code eines C#-Programms



Eine Assembly kann auch direkt in Maschinencode übersetzt werden, der beim Programmaufruf vollständig vorliegt.

Zum Erstellen eines lauffähigen C#-Programms wird der Quellcode des Programms zunächst mit einem Texteditor eingegeben und in einer *Quelldatei* mit der Kennung `.cs` gespeichert. Die Quelldatei wird dann dem Compiler zur Übersetzung gegeben.

Assemblies

Falls der Quellcode fehlerfrei übersetzt werden konnte, generiert der Compiler ein Modul, dessen Code der *Common Intermediate Language* entspricht. Dieser Bytecode ist für alle CLI-konformen Sprachen einheitlich. Er enthält *Metadaten*, das sind selbstbeschreibende Informationen über die im Modul definierten Typen, insbesondere auch die Adressen von Methoden und die Größe des von jeder Methode beanspruchten Speichers.

Bei größeren Software-Projekten wird der Quellcode des Programms auf viele Quelldateien verteilt, die getrennt kompiliert werden können. Zusammengehörige Module werden mit einem *Manifest* versehen. Dieses speichert Informationen über verwendete Ressourcen (zum Beispiel Bilddateien als Bitmaps) und Klassenbibliotheken, aber auch Versionsnummern und Verschlüsselungsinformationen.

Ein oder mehrere Module zusammen mit einem Manifest heißen *Assembly*. Nebenstehend ist die einfache Situation dargestellt, dass das C#-Programm nur aus einer Quelldatei besteht.

Laufzeitumgebung

Auf der Ebene des Betriebssystems ist eine Assembly allein jedoch nicht lauffähig. Sie wird zur Laufzeit von einem *Just-in-Time Compiler* (kurz: JITter) in nativen Code übersetzt. Der JITter nimmt dabei Typüberprüfungen und Optimierungen für den Zielrechner vor. Er gehört zur Laufzeitumgebung, die die Speicherverwaltung (engl. *Garbage Collection*) handhabt, Sicherheits- und Bereichsüberprüfungen vornimmt und außerdem das Exception Handling durchführt.

Programme, die in den Zwischencode der Intermediate Language übersetzt sind und unter Kontrolle der Laufzeitumgebung ablaufen, werden auch als *Managed Code* bezeichnet. Zusätzlich besteht die Möglichkeit, so genannten *Unmanaged Code* zu erstellen, der dann direkt unter Kontrolle des Betriebssystems abläuft.

In einer *integrierten Entwicklungsumgebung* (kurz: IDE) werden Programme von einer Benutzeroberfläche aus editiert, kompiliert und gestartet. Außerdem können weitere Tools, wie Debugger und Designer, gestartet werden.

Ein erstes C#-Programm

Beispielprogramm

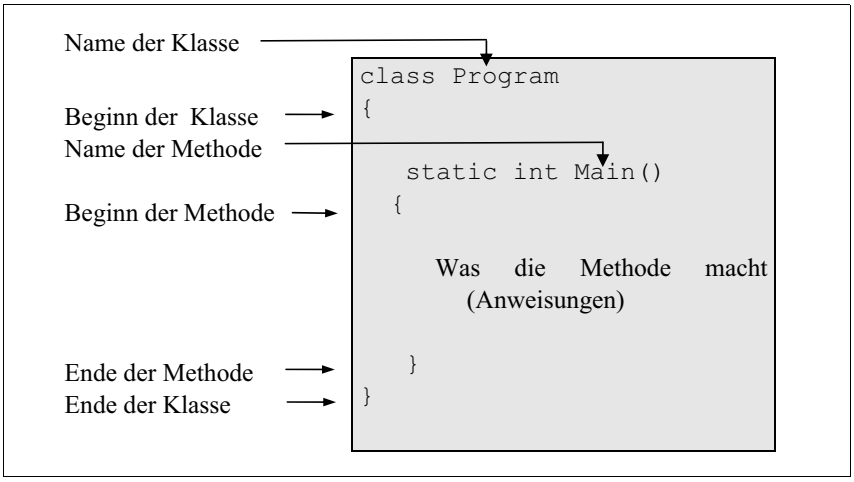
```
using System;

class Program
{
    static int Main(){
      Console.WriteLine("Viel Spaß mit C#!");
      return 0;
    }
}
```

Ausgabe auf dem Bildschirm (Konsole)

```
Viel Spaß mit C#!
```

Struktur einer Klasse



Während die statische Methode `Main()` als Einstiegspunkt des Programms immer denselben Namen hat, ist der Name der Klasse, in der die Main-Methode definiert ist, frei wählbar.

Die Bausteine eines C#-Programms sind Klassen. Jede Klasse beschreibt Daten und Operationen, die mit den Daten ausgeführt werden können. Operationen werden innerhalb einer Klasse als Methoden definiert. Jede Methode ist entweder selbst erstellt oder eine fertige Routine aus einer Klassenbibliothek. Die Methode `Main()` muss dabei immer selbst geschrieben werden und hat eine besondere Rolle: Sie bildet den *Einstiegspunkt* des Programms.

Zum Beispielprogramm

Anhand des nebenstehenden kurzen Beispielprogramms lässt sich bereits die Grundstruktur eines C#-Programms darstellen. Das Programm enthält nur eine Klasse `Program` mit der Methode `Main()` und gibt einen Text auf dem Bildschirm aus.

Die `using`-Direktive in der erste Zeile `using System;` ermöglicht es, Namen aus dem Namensbereich `System` direkt zu verwenden. Im Namensbereich `System` sind alle Namen der CLI-Klassenbibliothek, zu der auch die Klasse `Console` für Ein-/Ausgaben gehört, definiert.

Die Klasse `Program` besitzt eine statische Methode `Main()`. Diese Methode muss in jedem C#-Programm vorhanden sein, denn die Programmausführung beginnt mit der ersten Anweisung in `Main()`. Das Schlüsselwort `static` bedeutet, dass die Methode direkt an die Klasse `Program` gebunden und über sie aufgerufen wird.

Anweisungen

In unserem Beispiel enthält die Methode `Main()` zwei *Anweisungen*. In der ersten Anweisung

```
Console.WriteLine("Viel Spaß mit C#!");
```

wird die Methode `WriteLine()` der Klasse `Console` aufgerufen, die den Text `Viel Spaß mit C#!` auf dem Bildschirm ausgibt.

Die zweite Anweisung

```
return 0;
```

beendet die Methode `Main()` und so auch das Programm. Dabei wird der Wert `0` als *Status-Code* an die Laufzeitumgebung zurückgegeben. Der Status-Code `0` wird üblicherweise verwendet, wenn das Programm korrekt abgelaufen ist.

Jede Anweisung endet mit einem Semikolon `;`. Die kürzeste Anweisung besteht übrigens nur aus einem Semikolon und bewirkt nichts.

Struktur von C#-Programmen

Ein C#-Programm mit mehreren Klassen

```
/* ------------------------------------------------
   Program.cs     Ein Programm mit mehreren Klassen
   ------------------------------------------------ */
using System;

class Program
{
   static void Main()
   {
     Console.WriteLine("Los geht's!");

     Class1.Stars();
     Class2.Message();
     Class1.Stars();

     Console.WriteLine("Servus!");
   }
}

class Class1
{
   public static void Stars(){
      Console.WriteLine("******************");
   }
}
class Class2
{
   public static void Message(){
     Console.WriteLine("In Class2.Message()");
   }
}
```

Die Methode `Main()` kann, muss aber nicht als `public` deklariert werden, da sie der Einstiegspunkt des Programms ist.

Bildschirmausgabe

```
Los geht's!
******************
In Class2.Message()
******************
Servus!
```

Das nebenstehende Beispiel zeigt, wie ein C#-Programm strukturiert ist, das mehrere Klassen enthält. In der Klasse `Program` ist die Methode `Main()` definiert. Sie ruft die Methode `WriteLine()` auf und außerdem die Methoden `Stars()` und `Message()` der Klassen `Class1` bzw. `Class2`. Dabei wird der Klassenname gefolgt von einem Punkt dem Namen der Methode vorangestellt.

Die statischen Methoden `Stars()` bzw. `Message()` sind in den Klassen `Class1` und `Class2` als `public` deklariert. Damit werden die Methoden öffentlich zugänglich gemacht, das heißt, sie können auch außerhalb ihrer Klasse aufgerufen werden. Beide Methoden und auch `Main()` sind als `void` deklariert, da sie keinen Return-Wert besitzen.

Die Reihenfolge, in der Klassen deklariert werden, ist in C# nicht vorgeschrieben. Beispielsweise könnte auch zuerst die Klasse `Class1` und dann die Klasse `Program` angegeben sein. Es werden also Methoden aufgerufen, bevor sie definiert sind. Dies ist möglich, da der Compiler die notwendigen Informationen als Metadaten speichert.

Kommentare

Neu in diesem Beispiel sind auch die *Kommentare*. Das sind Zeichenfolgen, die mit `//` beginnen oder durch `/* ... */` eingeschlossen sind.

Beispiele:
```
// Einzeilen-Kommentar
/* Mehrzeilen-------------------
   --------------------- Kommentar */
```

Beim Einzeilen-Kommentar ignoriert der Compiler ab den Zeichen `//` alle Zeichen bis zum Zeilenende. Kommentare, die sich über mehrere Zeilen erstrecken, sind bei der Fehlersuche nützlich, um ganze Programmteile auszublenden. Jede der beiden Kommentarformen kann auch benutzt werden, um die andere auszukommentieren.

Layout

Der Compiler bearbeitet jede Quelldatei sequenziell und zerlegt den Inhalt in *Token*, das sind zum Beispiel Namen und Operatoren. Token können durch beliebig viele Zwischenraumzeichen (Leer-, Tabulator- oder Newline-Zeichen) getrennt sein. Es kommt also nur auf die Reihenfolge des Quellcodes an, nicht auf ein bestimmtes Layout.

Zur besseren Lesbarkeit von C#-Programmen ist es von Vorteil, einen einheitlichen Stil in der Darstellung beizubehalten. Dabei sollten gemäß der Programmstruktur Einrückungen und Leerzeilen eingefügt werden. Außerdem ist eine großzügige Kommentierung wichtig.

Verwenden von Namensbereichen

Erste Quelldatei

```
// ------------------------------------------
// MyClasses.cs
// Deklariert zwei Klassen Classs1 und Class2
// ------------------------------------------
using System;

namespace MyClasses
{
  public class Class1
  {
     public static void Stars(){
      Console.WriteLine("*******************");
     }
  }

  public class Class2
  {
     public static void Message(){
       Console.WriteLine("In Class2.Message()");
     }
   }
}
```

Zweite Quelldatei

```
// ------------------------------------------------------
// MyClasses_t.cs
// Verwendet die Klassen aus dem Namensbereich MyClasses
// ------------------------------------------------------
using System;
using MyClasses;  // Namensbereich MyClasses importieren

class Program
{
   static void Main(){
     Console.WriteLine("Los geht's!");
     Class1.Stars();
     Class2.Message();
     Class1.Stars();
     Console.WriteLine("Servus!");
   }
}
```

Die bisher betrachteten C#-Programme bestanden nur aus einer Quelldatei, in der lediglich auf die Klasse `Console` der Standardbibliothek Bezug genommen wurde. Größere Software-Systeme setzen jedoch zahlreiche Klassen ein, die von verschiedenen Entwicklern getrennt erstellt und ausgetestet werden.

Namensbereiche

Zur Demonstration ist nebenstehend ein C#-Programm formuliert, das die Klassen `Class1` und `Class2` aus dem Beispiel zuvor in eine separate Quelldatei auslagert. Da sie außerhalb der Quelldatei verfügbar sein sollen, sind beide Klassen als `public` deklariert. Typischerweise sind die Klassen in einem eigenen Namensbereich deklariert.

Namensbereiche unterstützen die interne Organisation von Programmen und verhindern Konflikte bei der Namensvergabe. So kann zum Beispiel der Name `Class1` *innerhalb* des Namensbereichs `MyClasses` direkt verwendet werden, unabhängig davon, ob eine andere Klasse `Class1` außerhalb des Namensbereichs bereits definiert wurde.

In der Quelldatei `MyClasses_t.cs` wird der Namensbereich `MyClasses` mit der `using`-Direktive importiert, so dass auch hier die Klassen `Class1` und `Class2` direkt angesprochen werden können. Die Klasse `Program` ist in keinem Namensbereich deklariert und gehört so zum *globalen Namensraum*.

Konsolenanwendungen

In einer Eingabeaufforderung (engl. *Command Prompt*) sind die Quelldateien mit einem Kommandozeilen-Compiler, zum Beispiel `gmcs` (Mono) oder `csc` (Microsoft), im aktuellen Verzeichnis wie folgt kompilierbar.

```
gmcs MyClasses_t.cs   MyClasses.cs
```

Es entsteht eine Assembly `MyClasses_t.exe`, die mit

```
MyClasses_t   bzw.  mono MyClasses_t.exe
```

gestartet werden kann. Besteht das Programm nur aus einer Quelldatei, so ist diese beim Kompilieren als einzige Datei anzugeben.

Bei Verwendung einer IDE wird in einer Projektmappe zunächst ein neues Projekt als *leere* Konsolenanwendung angelegt. Dann können C#-Quellcodedateien als neue Elemente zum Projekt hinzugefügt und editiert werden. Mit dem »Erstellen« des Projekts, das zuvor als Startprojekt festgelegt wurde, werden die Quelldateien kompiliert. Die dabei erzeugte Assembly kann in der IDE (normalerweise unter dem Menüpunkt DEBUGGEN) gestartet werden.

Übungen

Programm-Listing zur 3. Aufgabe:

```
// ------------------------------------------------
// Pause_t.cs   Demonstriert den Einsatz von Klassen.
// ------------------------------------------------

using System;

class Program
{
    static int Main(){
      Console.WriteLine("Lieber Leser, ") ;
      Console.WriteLine("machen Sie jetzt");

      Pause.Message() ;

      return 0;
     }
}

 class Pause{
     public static void Message()
     {
         Console.WriteLine("eine PAUSE!");
     }
 }
```

1. Aufgabe

Erstellen Sie ein C#-Programm, das den folgenden Text auf dem Bildschirm ausgibt:

```
Mögen täten wir schon wollen,
aber können...
haben wir uns nicht getraut!
```

Falls nach dem Start des Programms das Ausgabefenster sofort wieder geschlossen wird, fügen Sie am Ende der Methode `Main()` folgende Anweisung in den Quelltext ein:

```
Console.ReadLine();
```

Dann wird die Programmausführung so lange angehalten, bis der Anwender die Return-Taste drückt.

2. Aufgabe

Das folgende Programm enthält lauter Fehler:

```
\\ Hier muss man ziemlich genau hinschauen.
using system;
class Brille(
   static void main{}
   {
     Console.Writeline("Wenn dieser Text  "),
     Console.WriteLine("ausgegeben wird, ";
     Console.writeLine('sollten Sie in die ');
     Console.WriteLine("Luft springen!)".
     return 0;
           )
}
```

Korrigieren Sie die Fehler und testen Sie die Lauffähigkeit des Programms.

3. Aufgabe

Was gibt das nebenstehende C#-Programm auf dem Bildschirm aus?

Lösungen

Zur 1. Aufgabe:

```
//------------------------------------------------
// Koennen.cs   Text "Mögen täten wir . . . "
//              auf dem Bildschirm ausgeben.
//------------------------------------------------
using System;
class Program{
  static void Main(){
    Console.WriteLine("Mögen täten wir schon wollen, ");
    Console.WriteLine("aber können...");
    Console.WriteLine("haben wir uns nicht getraut!");
 }
}
```

Zur 2. Aufgabe:

Die korrigierten Stellen sind unterstrichen.

```
\\ Hier muss man ziemlich genau hinschauen.
using system;
class Brille(
   static void main{}{
     Console.Writeline("Wenn dieser Text  "),
     Console.WriteLine("ausgegeben wird, ";
     Console.writeLine('sollten Sie in die ');
     Console.WriteLine("Luft springen!").
             return 0;
           )
}
```

Zur 3. Aufgabe:

Die Bildschirmausgabe des Programms:

```
Lieber Leser,
machen Sie jetzt
eine PAUSE!
```