Vorlesung Softwareentwicklung 2021

https://github.com/SebastianZug/CsharpCourse

André Dietrich Christoph Pooch Fabian Bär Fritz Apelt Galina Rudolf JohannaKlinke Jonas Treumer KoKoKotlin Lesestein LinaTeumer MMachel Sebastian Zug Snikker123 Yannik Höll Florian2501 DEVensiv fb89zila

OOP Motivation

Parameter	Kursinformationen
Veranstaltung:	`Vorlesung Softwareentwicklung`
Semester	`Sommersemester 2022`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Einführung der Konzepte OOP am Beispiel von Structs`
Link auf den GitHub:	https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/07_OOPGrundlagenI.md
Autoren	@author

Structs

Ein struct-Typ ist ein ein Werttyp, der in der Regel verwendet wird, um eine Gruppe verwandter Variablen zusammenzufassen. Beispiele dafür können sein:

- kartesische Koordinaten (x, y, z)
- Merkmale eines Produktes (Größe, Name, Preis)
- Charakteristik einer Datei (Speicherort, Name, Größe, Rechteinformationen)

Ausgangspunkt für die weiteren Überlegungen ist die Konfiguration von `structs` in C.

> Machen Sie sich, wenn Ihnen hier die Grundidee noch unklar ist mit den `structs` anhand weiterer Materialien vertraut.

!?Einführung in Structs

```
#include <stdio.h>
#include <stdlib.h>

typedef struct RectangleStructure{
    double length;
    double width;
    double area;
} Rectangle;

int main(int argc, char const *argv[])
{
    Rectangle rect1, rect2, rect3;
    // Initialisierung:
    rect1.length = 2.5; rect1.width = 5;
    rect2.length = 4; rect2.width = 8;
```

```
    rect3.length = 1.5; rect3.width = 2.1;
    // Berechnung:
    rect1.area = rect1.length * rect1.width;
    rect2.area = rect2.length * rect2.width;
    rect3.area = rect3.length * rect3.width;
    // Ausgabe:
    printf("Rectangle 1 has an area of %5.1f\n",rect1.area);
    printf("Rectangle 2 has an area of %5.1f\n",rect2.area);
    printf("Rectangle 3 has an area of %5.1f\n",rect3.area);
    return 0;
}
```

Wir können also Daten unterschiedlichen Typs situationsspezifisch zusammenfassen ... Was fehlt uns?

Richtig, ein Set zugehöriger Funktionen!

```
#include <stdio.h>
#include <stdlib.h>

typedef struct RectangleStructure{
    double length;
    double width;
    double area;
} Rectangle;

void initializeRectangle(Rectangle *actualRect, double length,
                         double width){
 actualRect->length = length;
 actualRect->width = width;
}

void computeArea(Rectangle *actualRect){
 actualRect->area = actualRect->length * actualRect->width;
}

void printRectangleArea(Rectangle *actualRect){
 printf("The Rectangle has an area of %f\n",actualRect->area);
}

int main(int argc, char const *argv[])
{
    Rectangle rect1;
    // Initialisierung:
    initializeRectangle(&rect1, 30, 40);
    computeArea(&rect1);
    printRectangleArea(&rect1);
    return 0;
}
```

C sieht keine Möglichkeit vor Funktion und Daten "dichter" zusammenzubringen, wenn man von Funktionspointern im `struct` absieht.

Structs in C

Wir fokussieren uns zunächst auf `structs` und übertragen das dabei gewonnene Verständnis dann auf die Klassen. Dabei gilt es einige Unterschiede zu beachten!

Und in C#? Die Sprache erweitert das Konzept in Richtung der Konzepte der objektorientierten Programmierung und integriert Operationen über den Daten in das `struct`-Konzept. Dabei können sie folgende Elemente umfassen:

- Felder und Konstanten
- Methoden

- Konstruktoren und Destruktoren
- Eigenschaften(Properties)
- Indexer
- Events
- überladene Operatoren
- geschachtelte Typen

Konzentrieren wir im ersten Beispiel auf die Felder und die Methoden. Wie sieht eine entsprechende Definition des Bauplanes aller Instanzen von `Animal` aus?

```
public struct Animal
{
  public string name;               // Felder / Konstanten
  public string sound;              //
  public void MakeNoise() {         // Methode
    Console.WriteLine($"{name} makes {sound}");
  }
} // <- Keine Semikolon liebe C++ Programmierer!
```

Sowohl die ganze Struktur als auch die einzelnen Felder sind mit entsprechenden Sichtbarkeitsattributen versehen. Hier wurde explizit `public` vorgesehen, damit ist Animal aber auch alle Elemente uneingeschränkt "sichbar". In der Regel ist das aber nicht gewünscht.

Wie legen wir nun ein Objekt entsprechend der Spezifikation an? Wir haben mit dem `struct` einen neuen Typ definiert. Der Aufruf kann (!) anhand des Formats `<datentyp> Variablenname` erfolgen.

```
using System;

public struct Animal
{
  public string name;               // Felder / Konstanten
  public string sound;              //
  public void MakeNoise() {         // Methode
    Console.WriteLine($"{name} makes {sound}");
  }
}
public class Program
{
  static void Main(string[] args){
    Animal kitty;
    kitty.name = "Kitty";
    kitty.sound = "Miau";
    kitty.MakeNoise();
  }
}
```

Erstellen Sie eine weiteres Objekt vom Typ `Animal` und erstellen Sie eine Kopie von Kitty.

Herausforderung `this`

Warum nutzen einige Beispiele das Schlüsselwort `this`? Es wird verwendet, um auf die aktuelle Instanz der Klasse zu verweisen. Obiges Beispiel kann also auch wie folgt geschrieben werden.

```
using System;

public struct Animal
{
  public string name;
  public string sound;
  public void MakeNoise() {
    Console.WriteLine($"{this.name} makes {this.sound}");
  }

  public void setName(string name) {
```

```
    this.name = name;
    this.MakeNoise();
  }
}

public class Program
{
  static void Main(string[] args){
    Animal kitty;
    kitty.name = "Kitty";
    kitty.sound = "Miau";
    kitty.MakeNoise();
    kitty.setName("Tom");
  }
}
```

Das Verständnis von `this` macht die grundlegende Idee deutlich. Das `struct Animal` ist ein genereller Bauplan. Die Instanzen davon haben ein "Selbstverständnis" in Form der `this` Referenz.

Konstruktoren

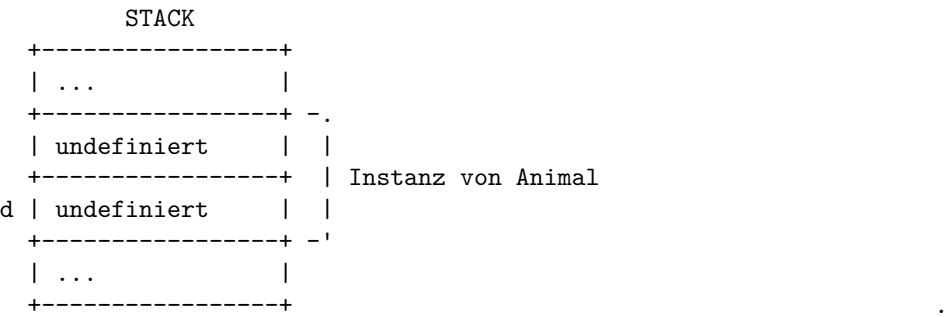
Nun umfasst unsere Datenstruktur möglicherweise - anders als unser Beispiel - eine große Zahl von individuellen Variablen.

```
using System;

public struct Animal
{
  public string name;
  public string sound;
}

public class Program
{
  static void Main(string[] args){
    Animal kitty;
    //kitty.name = "Kitty";   // <- fehlende "manuelle" Initalisierung
    //kitty.sound = "Miau";
    Console.WriteLine(kitty.name);
  }
}
```

Wie sieht unser Speicher nach Zeile 12 aus?



Wie können wir also sicherstellen, dass die Initialisierung vollständig vorgenommen wurde?

Konstruktoren sind spezielle Methoden für die Initialisierung eines Objektes. Sie werden einmalig aufgerufen.

Und wie erfolgt der Aufruf des Konstruktors, einer Funktion, die auf einer Datenstruktur wirkt, die es noch gar nicht gibt? Das Schlüsselwort `new` übernimmt diese Aufgabe für uns.

```
<Type> <Variablenname> = new <Konstruktorsignatur>;
```

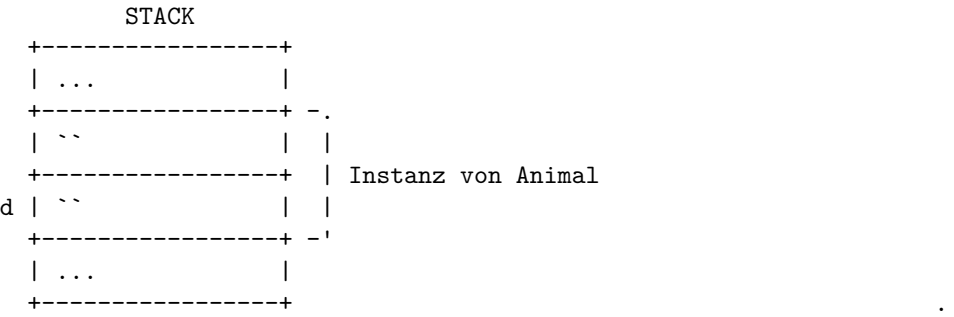
```
using System;

public struct Animal
{
  public string name;
  public string sound;

  public void MakeNoise() {
    Console.WriteLine($"->{this.name}<- makes ->{this.sound}<-");
  }
}

public class Program
{
  static void Main(string[] args){
    Animal cat = new Animal();
    cat.MakeNoise();
  }
}
```

Nach dem Aufruf besteht eine mit den Standardwerten der Datentypen vorinitialisierte Instanz auf dem Stack.



In `structs` dürfen allerdings (im Unterschied zu Klassen) keine parameterlosen Methoden sein. Der Compiler erzeugt diese automatisch, die Methode beschreibt alle Felder mit den datentypspezifischen Nullwerten.

Der Standardkonstruktor hilft aber nur bei der Vermeidung von uninitalisierten Werten. In der Regel wollen wir den Feldern aber konkrete Werte zuordnen.

```
   + Deklaration eines parameterlosen Konstruktors
   + Deklaration eines Konstruktors mit einzelnem Parameter
   + Überladen des Konstruktors
```

–>

```
using System;

public struct Animal
{
  public string name;
  public string sound;

  public Animal(string name, string sound) {
    this.name = name;
    this.sound = sound;
  }

  public void MakeNoise() {
    Console.WriteLine($"{this.name} makes {this.sound}");
  }
}

public class Program
{
```

```
  static void Main(string[] args){
    Animal dog = new Animal("Roger", "Wuff");
    dog.MakeNoise();
  }
}
```

Anmerkung: Konstruktoren sind Methoden und folglich steht das gesamte Spektrum der Variabilität bei deren Definition zur Verfügung (Überladen, vordefinierte Variablen, Parameterlisten, usw.)

Konstruktor	Aufruf
Aufruf des (impliziten) Standardkonstruktors	`Animal kitty = new Animal()`
`public Animal(name, sound)`	`Animal kitty = new Animal("kitty","Miau")`
`public Animal(name, sound = "Miau")`	`Animal kitty = new Animal("kitty")`

Im Hinblick auf die Verwendung des Schlüsselwortes `new` und die Konsequenzen in Bezug auf die Nutzung des Speichers, sei auf den exzellenten Beitrag von Clark Kromenaker verwiesen.

In Ergänzung sei auch noch auf die kompakte *Fat Arrow* Darstellung im Zusammenhang mit Konstruktoren, die ja Funktionen wie alle anderen sind verwiesen. Wenn nur eine Anweisung ausgeführt wird kann dies in einer Zeile realisiert werden.

```
using System;

public struct Animal
{
  public string name;
  public Animal(string name) => this.name = name;
  public void MakeNoise() {
    Console.WriteLine("{0} makes Miau", name);
  }
}
public class Program
{
  static void Main(string[] args){
    Animal cat = new Animal("Kitty");
    cat.MakeNoise();
  }
}
```

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
</Project>
```

Ein alternatives Vorgehen bietet die sogenannte Object Initialization Syntax aus C# 3.0 nutzen. Der Compiler generiert den zugehörigen Code für die spezifische Initialisierung.

Dies kann zum Beispiel ein Array mit allen Tieren unseres virtuellen Bauernhofes sein. Wie werden diese dann initialisiert?

```
using System;

public struct Animal
{
  public string name;
  public string sound;
  public void MakeNoise() {
    Console.WriteLine("{0} makes {1}", name, sound);
  }
}
public class Program
```

```
{
  static void Main(string[] args){
     Animal[] myAnimals = new Animal[]{
      new Animal{ name = "Kitty", sound = "Miau"},     // Object Initialization Syntax
      new Animal{ name = "Wally", sound = "Wuff"},
      new Animal{ name = "Berta", sound = "Muuuh"}
    };
    foreach(Animal animal in myAnimals){
      animal.MakeNoise();
    }
  }
```

Aufgabe: Versuchen Sie das Beispiel um einen Konstruktor und einen zugehörigen Aufruf für die Initialisierung zu ergänzen!

Veränderliche und nicht veränderliche Felder

Aspekt	`readonly` Felder	`const` Felder
Zweck	… wird verwendet, um ein schreibgeschütztes Feld zu erstellen.	… wird verwendet, um konstante Felder zu erstellen.
Typ	… ist eine zur Laufzeit definierte Konstante.	… wird verwendet, um eine Konstante zur Kompilierzeit zu erstellen.
Wert	… kann nach der Deklaration geändert werden.	… kann nach der Deklaration nicht geändert werden.
Wertzuweisung	… werden als Instanzvariable deklariert und im Konstruktor mit Werten belegt.	… sind zum Zeitpunkt der Deklaration zuzuweisen.
Einbettung in Methoden	… können nicht innerhalb einer Methode definiert werden.	… können innerhalb einer Methode deklariert werden.

```
using System;

public class Animal
{
  //public const string name = "Kitty";
  public string name = "Kitty";
  public readonly int[] legCount = new int[1];  // Referenzdatentyp (etwas konstruiert)
  public readonly int age;                      // Wertdatentyp
  //public const long[] hairCount = new int[1]; //geht nicht, da const nicht auf Refernztyp verweisen d

  public Animal(string name, int age, int legs) {
    this.name = name;
    this.age = age;
    this.legCount[0]= legs;
  }
}
public class Program
{
  static void Main(string[] args){
    Animal kitty = new Animal("Miau", 5, 4);
    kitty.legCount[0]=5;
    Console.WriteLine(kitty.legCount[0]);
  }
}
```

Achtung: *Der readonly-Modifizierer verhindert, dass das Feld durch eine andere Instanz des Verweistyps ersetzt wird. Der Modifizierer verhindert jedoch nicht, dass die Instanzdaten des Felds durch das schreibgeschützte Feld geändert werden.*

Achtung: *Verwendet man* `const` *ist dies jedoch nicht möglich, da eine Konstante nur ein numerischer Typ, ein Boolean, ein String oder ein Enum sein kann.* `const` *kann also nicht auf einen Referenztyp verwendet werden (außer string -> **Ausnahme**) bzw. wenn doch, muss es dann immer*

***null** sein.*

C# 9 geht einen Schritt weiter und ermöglicht die pauschale Deklaration von `readonly structs`. Damit wird der Entwickler beauftragt JEDES Feld entsprechen mit `readonly` zu schützen.

```
using System;

public readonly struct Animal
{
  public string name;
  public string sound;
  public int age;

  public Animal(string name, string sound, int age) {
    this.name = name;
    this.sound = sound;
    this.age = age;
  }
}

public class Program
{
  static void Main(string[] args){
    Animal kitty = new Animal("Kitty", "Miau", 5);
    Console.WriteLine(kitty.sound);
  }
}
```

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
</Project>
```

In der nächsten Vorlesung werden die sogenannten Eigenschaften genauer untersucht. Diese eröffnen nochmals weitere Möglichkeiten der Kapselung.

Sichtbarkeitsattribute

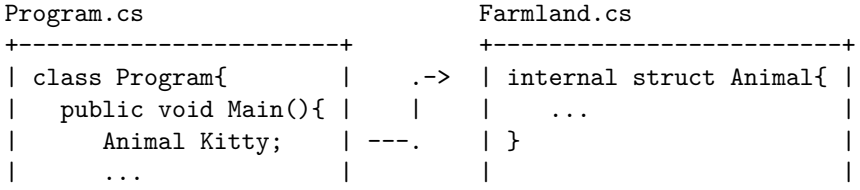
Strukturen und Klassen abstrahieren Daten und verbergen konkrete Realisierungen von Programmteilen. Um zu steuern, welche Elemente eines Programms aus welchem Kontext heraus sichtbar sind, werden diese mit Attributen versehen.

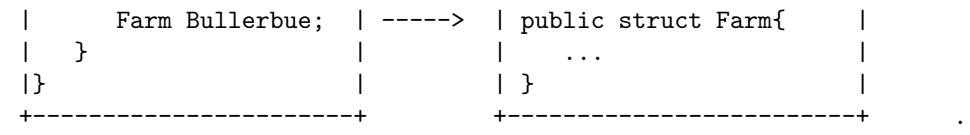
Sichtbarkeit der Stuktur

Strukturen, die innerhalb eines *Namespace* (mit anderen Worten, die nicht in anderen Klassen oder Strukturen geschachtelt sind) direkt deklariert werden, können entweder `private`, `public` oder `internal` sein.

Bezeichner	Konsequenz
`public`	Keine Einschränkungen für den Zugriff
`internal`	Der Zugriff ist auf die aktuelle *Assembly* beschränkt.
`private`	Der Zugriff kann nur aus dem Code der gleichen Struktur erfolgen.

Wenn kein Modifizierer angegeben ist, wird standardmäßig `internal` verwendet.






```
Schritt 1                                   mcs -target:library Farmland.cs
Schritt 2   mcs -reference:Farmland.dll Program.cs
```

Das struct “Animal” soll in einem anderen Assembly nicht aufrufbar sein. Wir wollen die Implementierung kapseln und verbergen. Folglich generiert der entsprechende Aufruf einen Compiler-Fehler. Zugehörige Dateien sind unter GitHub zu finden.

Variante 1: Compilieren von Farmland und Programm in ein Assembly

```
mcs Program.cs Farmland.cs
My name ist Kitty.
```

Variante 2: Compilieren von Farmland als externe Bibliothek

```
mcs -target:library Farmland.cs
mcs -reference:Farmland.dll Program.cs
Programm.cs(8,13): error CS0122: `Farm.Animal' is inaccessible due to its protection level
Programm.cs(9,26): error CS0841: A local variable `cat' cannot be used before it is declared
```

Sichtbarkeit der Felder und Member einer Stuktur

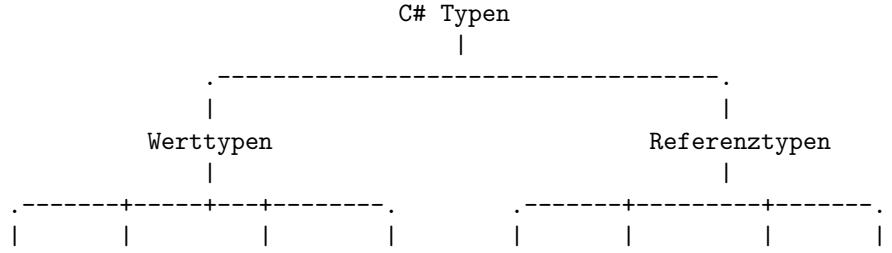
Für die Sichtbarkeit auf der Ebene der Felder und Member können für structs drei Attribute verwendet werden `private`, `internal` und `public`. Standard ist `private`.

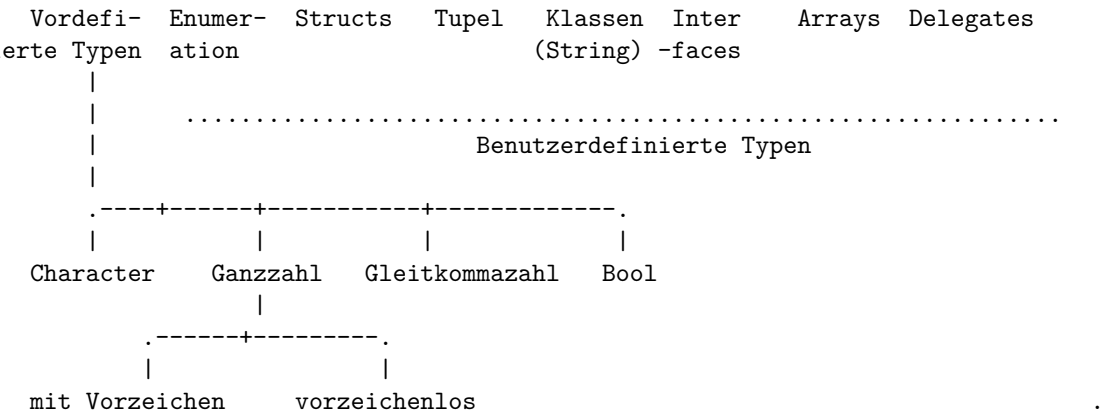
```
using System;

public struct Animal
{
  public string name;
  internal string sound;
  private byte age;
  public Animal(string name, uint born, string sound = "Miau"){
    this.name = name;
    this.sound = sound;
    age = (byte) (2019 - born);
  }
  public void MakeNoise() {
    Console.WriteLine("{0} ({1} years old) makes {2}", name, age, sound);
  }
}

public class Program
{
  static void Main(string[] args){
    Animal Wally = new Animal ("Wally", 2014, "Wau");
    Wally.MakeNoise();
    Wally.age = 5;
  }
}
```

Vergleich mit Klassen





Das vorangegangene Beispiel als Klassenimplementierung:

```
using System;

//public struct Animal
public class Animal
{
  public string name;
  private string sound;
  private byte age;
  public Animal(string name, uint born, string sound = "Miau"){
    this.name = name;
    this.sound = sound;
    age = (byte) (2019 - born);
  }
  public void MakeNoise() {
    Console.WriteLine("{0} ({1} years old) makes {2}", name, age, sound);
  }
}

public class Program
{
  static void Main(string[] args){
    Animal Wally = new Animal ("Wally", 2014, "Wau");
    Wally.MakeNoise();
  }
}
```

Structs	Klassen
Werttyp (Variablen enthalten das Objektes)	Referenzdatentyp
Abgelegt auf dem Stack	Gespeichert auf dem Heap
Unterstützen keine Vererbung	Unterstützen Vererbung
können Interfaces implementieren	können Interfaces implementieren
`internal` und `public` als Struct-Attribute	`private`, `internal` und `public` als Klassenattribute
`private`, `internal` und `public` als Feld und Member-Attribute	analog plus 3 weitere Attribute
keine parameterlosen Konstruktoren deklarierbar	

Welche Konsequenzen hat das zum Beispiel?

Merke: Strukturen sind Wertdatentypen!

```
using System;

public class Animal
{
```

```
  public string name;
  ...
  }
}

public struct Human
{
  public string name;
  ...
  }
}

public class Program
{
  static void Main(string[] args){
    Animal kitty = new Aninmal();  // <- Referenzvariable
    Human farmer = new Farmer();   // <- Wertvariable
  }
}
```

Beispiel der Woche

Im folgenden Beispiel werden Instanzen des Structs “Animal” von einem anderen Struct “Farm” genutzt und dort in einer (generischen) Liste gespeichert.

```
using System;
using System.Collections;
using System.Collections.Generic;

public struct Animal
{
  public string name;
  public string sound;

  public Animal(string name, string sound = "Miau"){
    this.name = name;
    this.sound = sound;
  }

  public void MakeNoise() {
    Console.WriteLine("{0} makes {1}", name, sound);
  }
}

public struct Farm{
  public string adress;
  public List<Animal> animalList;

  public Farm(string adress) {
    animalList = new List<Animal>();
    this.adress = adress;
  }

  public void AddAnimal(Animal newanimal){
    animalList.Add(newanimal);
  }

  public void PrintAnimals(){
    foreach (Animal pet in animalList){
      pet.MakeNoise();
    }
```

```
  }
}

public class Program
{
  static void Main(string[] args){
    Animal Wally = new Animal ("Wally", "Wau");
    Animal Kitty = new Animal ("Kitty", "Miau");
    Farm myFarm = new Farm("Biobauernhof Freiberg");
    myFarm.AddAnimal(Wally);
    myFarm.AddAnimal(Kitty);
    myFarm.PrintAnimals();
  }
}
```

Aufgaben

Vollziehen Sie die Überlegungen rund um `structs` nach! Das ist Elementar für das weitere Vorgehen.

!?Konstruktoren

!?Klassen

Für diejenigen, die bei der Befragung beim letzten mal “Langweilig” ausgewählt hatten, eine Anregung für die Verwendung von C# …

!?C# und Unity