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 Konzepte I

Parameter	Kursinformationen
Veranstaltung:	`Vorlesung Softwareentwicklung`
Semester	`Sommersemester 2022`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Konzepte OOP Programmierung und Umsetzung in C#`
Link auf den GitHub:	https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/08_OOPGrundlagenII.md
Autoren	@author

Auf Nachfrage ...

Wann machen `private` Klassen oder `structs` Sinn?

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

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

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

  public void AddAnimal(string name, string sound){
    animalList.Add(new Animal(name, sound));
  }

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

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

Wie verhält es sich mit mehreren Dateien in einem Ordner und wie stellt man die Relationen zwischen separaten Assemblies her?

```
dotnet new sln -o assemblies_dotnet
cd assemblies_dotnet
dotnet new console -o MyApp
dotnet new classlib -o MyClass
dotnet sln add MyApp
dotnet sln add MyClass
cd MyApp
dotnet add reference ../MyClass
```

Kopieren Sie noch die Dateien aus dem Mono Verzeichnis in die entsprechenden Ordner.

Starten Sie die Kompilierung, in dem Sie `dotnet run` im Ordner `MyApp` aufrufen. Was beobachten Sie?

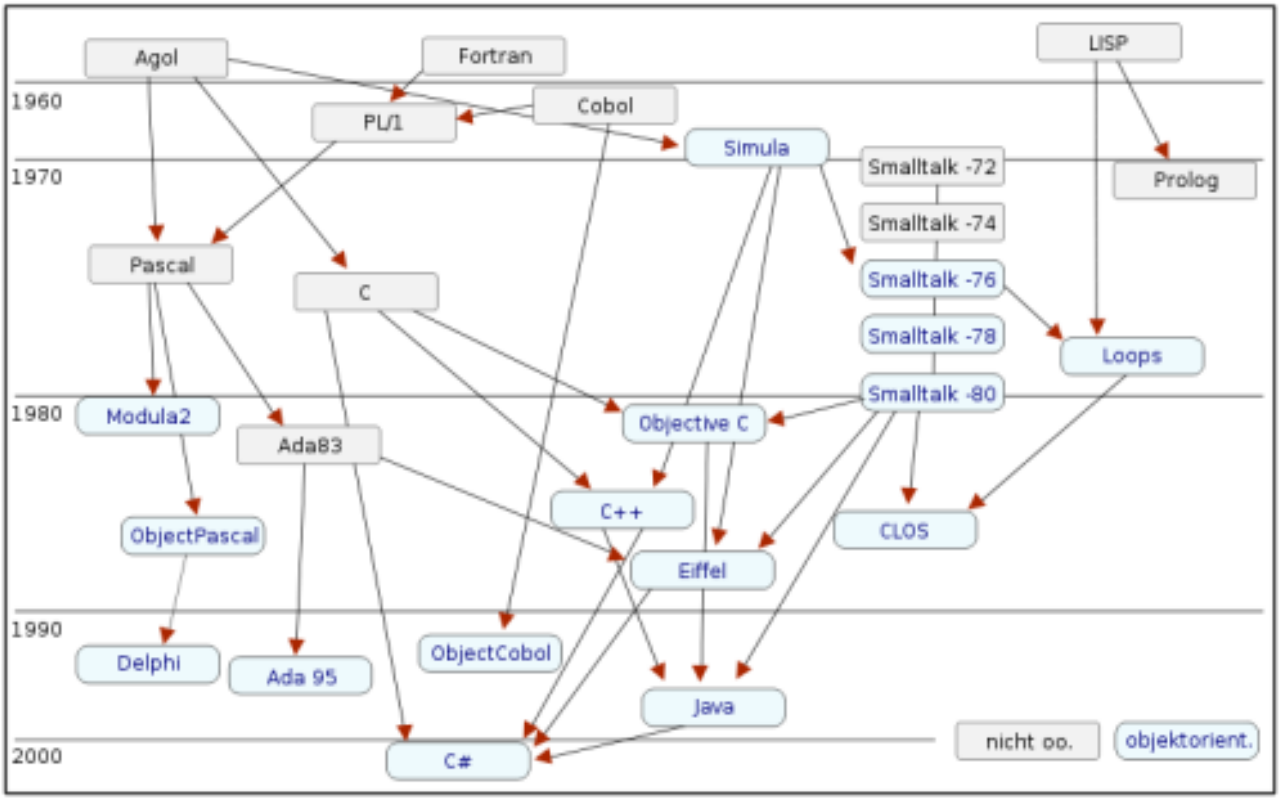
Visionen der Objektorientierung

Ein Objekt ist ein Bestandteil eines Programms, der Zustände enthalten kann. Diese Zustände werden von dem Objekt vor einem Zugriff von außen versteckt und damit geschützt. Außerdem stellt ein Objekt anderen Objekten Operationen zur Verfügung. Von außen kann dabei auf das Objekt ausschließlich zugegriffen werden, indem eine Operation auf dem Objekt aufgerufen wird. Ein Objekt legt dabei selbst fest, wie es auf den Aufruf einer Operation reagiert. Die Reaktion kann in Änderungen des eigenen Zustands oder dem Aufruf von Operationen auf weiteren Objekte bestehen.

Merke *Ein Objekt ist eine zur Ausführungszeit vorhandene und für ihre Member Speicher allozierende Instanz, die sich aus der Spezifikation einer Klasse erschließt.*

Ideen der OOP: * Objekte der *realen Welt* müssen sich in der Programmierung widerspiegeln * Es geht nicht um das Manipulieren von Daten, sondern um Zustandsänderungen von Objekten * Im Zentrum der objektorientierten Programmierung stehen Objekte, die miteinander kommunizieren

Merke Wir haben zwei Herausforderungen zu meistern - Modellierung und Realisierung.



Beispiel - Simulationsumgebung Fußballspiel:

- 1 Objekt vom Typ “Spielsituation”
- 1 Objekt vom Typ “Ball”
- 2 Objekte vom Typ “Trainer”
- 3 Objekte vom Typ “Schiedsrichter”
- 22 Objekte vom Typ “Fußballspieler”

Welche Eigenschaften hat jedes Objekt des Typen “Spieler”, “Trainer” bzw. “Schiedsrichter”?

- Name, Alter, Geschlecht, Gewicht, Größe
- Position (x, y, z),
- im Spiel, Geschwindigkeit
- Mannschaft, Rolle (Stürmer, Tormann, Verteidiger), Nummer
- physischer Zustand (topfit, ausgepowert, verletzt)

Einige der Eigenschaften …

- … ändern sich im Spielkontext, andere bleiben konstant
- … lassen sich durchaus allen Personen zuordnen, anderen nur spezifischen Kategorien von Beteiligten.

Welche Eigenschaften und Methoden (Fähigkeiten) sind für die Instanzen aller Teilnehmer gleich??

- Name, Alter, Geschlecht, Gewicht, Größe
- physischer Zustand (topfit, ausgepowert, verletzt)
- `ändertPosition()`

Welche Eigenschaften und Methoden (Fähigkeiten) sind unterschiedlich??

- Rolle in der Mannschaft und Trikotnummer gibt es nur für Spieler
- Mitglied einer Mannschaft bezieht Spieler und Trainer mit ein
- …

Welche Methoden sollten dem Objekt “Spieler” erlaubt sein und wie verändert dies deren Zustand?

- `FängtDenBall()` -> Wirkt sich auf den Zustand von Ball aus, die Position des Balles ist identisch mit der des Spielers … und es gibt nur einen Ball!
- `WirftDenBall()`

- `Foul(Spieler gefoulterSpieler)` -> Wirkt sich auf die Fitness von `gefoulterSpieler` aus

Welche Schwachstellen sehen Sie bei unserem Modellierungsansatz / der Realisierung?

Kapselung

> Die Verkapselung bezieht sich auf die "Einhüllung" von Daten und Methoden innerhalb einer Struktur (Klasse), die die Objektimplementierung verbirgt und den unmittelbaren Datenzugriff außerhalb vorbestimmter Dienste unterbindet.

Vom Innenleben einer Klasse soll der Verwender – gemeint sind sowohl die Algorithmen, die mit der Klasse arbeiten, als auch der Programmierer, der diese entwickelt – möglichst wenig wissen müssen (Geheimnisprinzip). Durch die Kapselung werden nur Angaben über das „Was“ (Funktionsweise) einer Klasse nach außen sichtbar, nicht aber das „Wie“ (die interne Darstellung).

Standardidentifier für Daten- und Methodenzugriffe sind dabei:

Bezeichner	UML Kürzel	Bedeutung
public	+	Zugreifbar für alle Objekte (auch die anderer Klassen)
private	-	Nur für Objekte der eigenen Klasse zugreifbar
protected	#	Nur für Objekte der eigenen Klasse und von Spezialisierungen derselben zugreifbar
internal		Der Zugriff ist auf die aktuelle Assembly beschränkt

Vorteile

- Da die Implementierung einer Klasse anderen Klassen nicht bekannt ist, kann ihre Implementierung geändert werden, ohne die Zusammenarbeit mit anderen Klassen zu beeinträchtigen.
- Beim Zugriff über eine Zugriffsfunktion spielt es von außen keine Rolle, ob diese Funktion komplett im Inneren der Klasse existiert, das Ergebnis einer Berechnung ist oder möglicherweise aus anderen Quellen (z. B. einer Datei oder Datenbank) stammt.
- Es ergibt sich eine erhöhte Übersichtlichkeit, da nur die öffentliche Schnittstelle einer Klasse betrachtet werden muss.
- Deutlich verbesserte Testbarkeit, Stabilität und Änderbarkeit der Software bzw. deren Teile (Module).

Nachteile

- In Abhängigkeit vom Anwendungsfall Geschwindigkeitseinbußen durch den Aufruf von Zugriffsfunktionen (direkter Zugriff auf die Datenelemente wäre schneller).
- Zusätzlicher Programmieraufwand für die Erstellung von Zugriffsfunktionen.

Beispiel Fußballsimulation

1. Die Position x,y eines jeden Spielers und des Balls sollte nur über entsprechende Zugriffsmethoden manipuliert werden.
2. Die `Foul` kann nur aus dem Spieler heraus aufgerufen werden :-)

```
                                public struct Position{float x; float y};
+---------------------------+
| Spieler                   |       public class Spieler{ // oder struct !!!
+---------------------------+         enum Rolle {Stürmer, Tormann, Verteidiger};
| - name: string            |           private string Name;                |
| - alter: byte             |           private byte? Alter;                |  Geschützte
| - player: Rolle           |           private Rolle player;               |  Felder
| ...                       |           private Position position;          |
+---------------------------+
|   FängtDenBall(): void    |           public get ... set ...              |  Zugriffsmethoden für
|   SchießtDenBall(): Kraft |                                                  Felder
| - Foul()                  |           public void FängtDenBall(Ball);
| ...                       |           public Kraft = SchießtDenBall(Ball);
+---------------------------+           private Foul(SpielerX);              |  Event an SpielerX im
                                      }                                        "Erfolgsfall"
```

Vererbung

> Die Vererbung dient dazu, aufbauend auf existierenden Klassen neue zu schaffen. Aus der Klassenspezifikation einer Klasse wird eine neue Klasse hergeleitet. Diese ist dann entweder eine Erweiterung oder eine Einschränkung der ursprünglichen Klasse.

Die vererbende Klasse wird meist Basisklasse (auch Super-, Ober- oder Elternklasse) genannt, die erbende abgeleitete Klasse (auch Sub-, Unter- oder Kindklasse). Den Vorgang des Erbens nennt man meist Ableitung oder Spezialisierung.

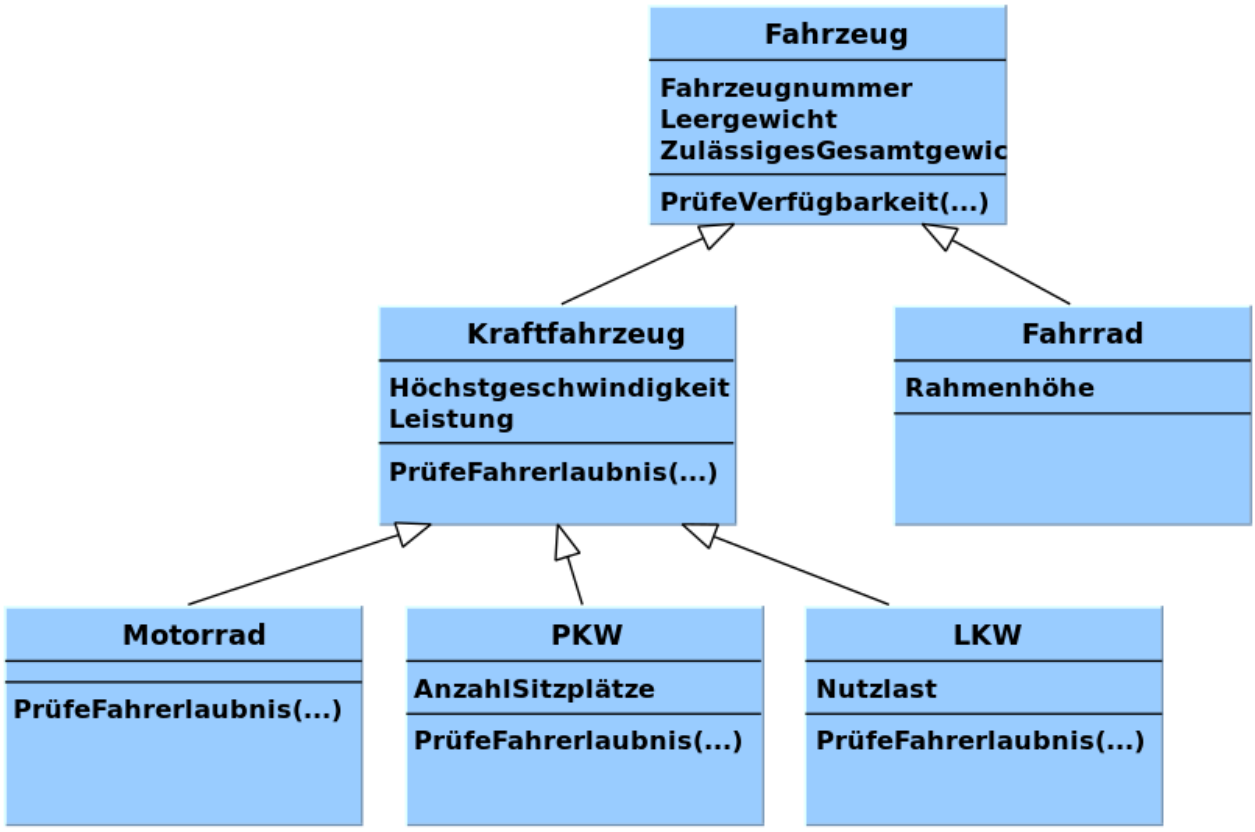


Figure 1: Vererbungsbeispiel

Vorteile

- Abbildung der Eigenschaften und Daten in einem hierarchischen Konzept
- Steigerung der Wartbarkeit und Erweiterbarkeit

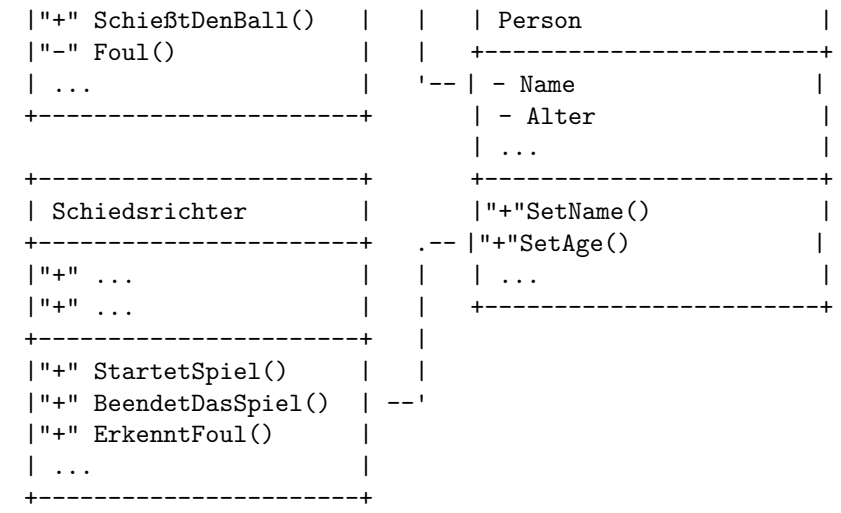
Nachteile

- Eine Klasse, die als Subklasse aus anderen Klassen entsteht, ist kein autonomer Baustein. Bei der Verwendung der Klasse wird es immer wieder zu Rückgriffen auf die Basisklasse(n) kommen.
- Bisweilen schwierige Modellierung
- Schaffung von Abhängigkeitsverhältnissen, die dem Modularisierungsgedanken nicht entsprechen.

Am Beispiel Fußballspiel

```
 Erbende Klasse                  Höherabstrakte Klasse                      .

+-----------------------+
| Spieler               |
+-----------------------+
|"+" Position           |
| ...                   | --.
+-----------------------+   |
|"+" FängtDenBall()     |   |   +-----------------------+
```



Zentrale Objekt-Klasse

https://docs.microsoft.com/de-de/dotnet/api/system.object?view=netframework-4.8

OOP Sprachen verfügen meist über eine zentrale Klasse, von der alle Klassen in letztlich abgeleitet sind. Diese heißt bei diesen Sprachen Object. In Eiffel wird sie mit ANY bezeichnet. Zu den wenigen Ausnahmen, in denen es keine solche Klasse gibt, zählen C++ oder Python.

In den Sprachen mit zentraler Basisklasse erbt eine Klasse, für die keine Basisklasse angegeben wird, implizit von dieser besonderen Klasse. Ein Vorteil davon ist, dass allgemeine Funktionalität, beispielsweise für die Serialisierung, Ausgaben, Hashwerte oder die Typinformation, dort untergebracht werden kann. Weiterhin ermöglicht es die Deklaration von Variablen, denen ein Objekt jeder beliebigen Klasse zugewiesen werden kann. Dies ist besonders hilfreich zur Implementierung von Containerklassen, wenn eine Sprache keine generische Programmierung unterstützt.

```
using System;

public class Program
{
  static void Main(string[] args){
    Console.WriteLine(typeof(int));
    Console.WriteLine(typeof(int).BaseType);
    Console.WriteLine(typeof(int).BaseType.BaseType);
  }
}
```

```
using System;

public class Program
{
  static void Main(string[] args){
    Type t = typeof(Program);
    Console.WriteLine("------ Methods --------");
    System.Reflection.MethodInfo[] methodInfo = t.GetMethods();
    foreach (System.Reflection.MethodInfo mInfo in methodInfo)
      Console.WriteLine(mInfo.ToString());
  }
}
```

Polymorphie

> Polymorphie oder Polymorphismus (griechisch für Vielgestaltigkeit) ermöglicht, dass ein Objekt sich in seiner Funktionalität in Abhängigkeit von den Datentypen verändert.

Die Polymorphie der objektorientierten Programmierung ist eine Eigenschaft, die immer im Zusammenhang mit Vererbung und Schnittstellen (Interfaces) auftritt. Eine Methode ist polymorph, wenn sie in verschiedenen Klassen die gleiche Signatur hat, jedoch erneut implementiert ist.

```
public class Shape
{
    public int X { get; private set; }
    ....
    // Virtual method
    public virtual void Draw()
    {
        Console.WriteLine("Performing base class drawing tasks");
    }
}

class Circle : Shape
{
    public override void Draw()
    {
        // Code to draw a circle...
        Console.WriteLine("Drawing a circle");
        base.Draw();
    }
}

class Square : Shape {}
...
```

In C# ist jeder Typ polymorph, da alle Typen, einschließlich benutzerdefinierten Typen, von `Object` erben.

Beim Vererben erhält die abgeleitete Klasse alle Methoden, Felder, Eigenschaften und Ereignisse der Basisklasse. Dabei gilt es zu entscheiden, welche davon unverändert übernommen und welche auf die spezifischen Anforderungen angepasst werden sollen.

Am Beispiel Fußballspiel

```
public class Person{
  private int alter;
  public virtual void setAge(int alter) {
    this.alter = alter;
  }
}

public class Spieler: Person {
  public override void setAge(int alter) {
      // hier wird noch getestet ob der Spieler älter als 16 ist
      // und überhaupt eingesetzt werden darf
  }
}

public class Zuschauer: Person {
  public override void setAge(int alter) {
      // hier wird noch getestet ob ein Zuschauer jünger als 6 ist und
      // kostenlos ins Stadion darf
  }
}
```

Weitere Beispiele

Beispiel	Kapselung	Vererbung	Polymorphie
Auto	Interne Daten (CAN-Nachrichten zwischen Motor und Getriebe, Zündzeitpunkte, etc.) sind für mich nicht interessant	Kleinwagen und Kombis sind lediglich spezielle Arten von Autos	Ein Maserati Quattroporte und ein C Corsa können beide fahren. Trotzdem sind die Auswirkungen verschieden

Beispiel	Kapselung	Vererbung	Polymorphie
Säugetiere	Die genauen Vorgänge der Verdauung interessieren nicht. Nur das benötigte Futter ist wichtig	Eine Springmaus und eine Hausratte sind beide aus der Ordnung der Nagetiere. Beide Tiere teilen viele gleiche Eigenschaften	Fortbewegen ist eine Eigenschaft jedes Säugetieres. Ein Wal schwimmt jedoch und ein Känguru hüpft. Ein Pferd galoppiert.

Begriffe

Begriff	Bedeutung
Klassen	... sind Vorlagen (Baupläne), aus denen Instanzen Objekte erzeugt werden.
Objekt	... ist ein Element, welches Funktionen, Methoden, Prozeduren, einen inneren Zustand, oder mehrere dieser Dinge besitzt. Es leitet sich von einer Spezifikation ab.
Entität	... ist ein Objekt, welches eine Identität besitzt, welche unveränderlich ist. Beispielsweise kann eine Person ihre Adresse, Telefonnummer oder Namen ändern, ohne zu einer anderen Person zu werden. Eine Person ist also eine Entität.
Eigenschaft	bestimmt den Zustands eines Objekts. Der Zustand des Objektes setzt sich aus seinen Eigenschaften und Verbindungen zu anderen Objekten zusammen.
Prozedur	verändert den Zustand eines Objektes, ohne einen Rückgabewert zu liefern. Eine Prozedur kann andere Objekte als Parameter entgegennehmen.
Funktion	ordnet einer gegebenen Eingabe einen bestimmten Rückgabewert zu. Eine Funktion zeichnet sich insbesondere dadurch aus, dass sie nicht den Zustand eines Objekts verändert.
Methode	ist ein Unterprogramm (Funktion oder Prozedur), welches das Verhalten von Objekten beschreibt und implementiert. Über Methoden können Objekte untereinander in Verbindung treten.

Klassen in C

> Klassen [und Strukturen] sind zwei der grundlegenden Konstrukte des allgemeinen Typsystems in .NET Framework. Bei beiden handelt es sich um eine Datenstruktur, die einen als logische Einheit zusammengehörenden Satz von Daten und Verhalten kapselt.

Entsprechend können Klassenspezifikationen folgende Elemente umfassen:

Member / Elemente	englische Bezeichnung	Funktion
Felder	*fields*	Daten
Konstanten	*constant*	konstante Daten
Eigenschaften	*property*	Daten und Zugriffsmethoden
Methoden	*method*	Funktionen / Prozeduren
Konstruktoren	*constructor*	Instanziierung einer Klasse
Ereignisse	*event*	Informationsaustausch zwischen Klassen
Finalizer	*finalizer*	"Destruktoren"
Indexer	*indexer*	Ähnlich Eigenschaften, Adressierung über Indizes
Operatoren	*operators*	Set von '==', '+' etc. mit eigener Bedeutung
Geschachtelte Typen	*embedded types*	Integrierte Klassen oder Structs, die nur innerhalb einer Klasse/ Structs angewendet werden

```
class Person{
  string name;                    // eine häufige Konvention, kleine Anfangs-
  public int Geburtsjahr;         // buchstaben = privat, groß = public

  public Person(string name, int geburtsjahr){
    this.name = name;
    Geburtsjahr = geburtsjahr;
```

```
  }

  int AktuellesAlter () => DateTime.Today.Year - Geburtsjahr;

  public override string ToString(){
     return name + " ist " + AktuellesAlter().ToString() + "Jahre alt."
  }

  public static bool operator< (Person person1, Person Person2){
    // TODO Hausaufgabe
  }
}
```

Und wie legen wir eine Instanz an? Dazu sind mehrere Schritte notwendig:

```
Person p;  // Generierung einer Referenzvariablen p auf dem Stack
p = new Person();  // Generierung einer Instanz im Heap
// alles zusammen
// Person p = new Person();
```

Als Operanden erwartet der new-Operator einen Klassennamen und eine Parameterliste, die an den entsprechenden Konstruktor übergeben wird.

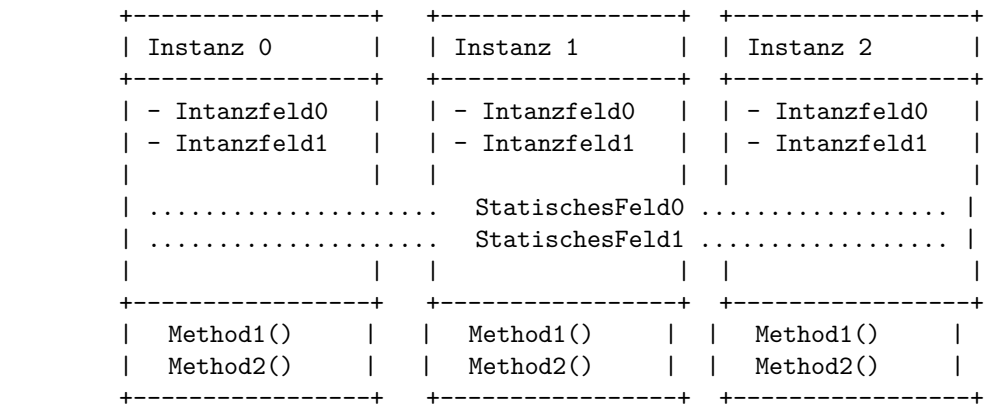
	Fields	Methods
Statisches Attribut	`static`	`static`
Zugriffsattribute	`public`, `internal`, `private`, `protected`	`public`, `internal`, `private`, `protected`
Vererbungsattribut	`new`	`new`, `virtual`, `abstract`, `override`, `sealed`
Unsafe Attribute	`unsafe`	
Attribut Teilimplementierung		`partitial`
Unmanaged Code Attribute		`unsafe extern`
Read-only Attribute	`readonly`	
Threading Attribute	`volatile`	

Felder

Felder sind Variablen eines beliebigen Typs, die einer Klasse unmittelbar zugeordnet sind. In Feldern werden die Daten abgelegt, die übergreifend Verwendung finden.

Der Idee der Kapselung folgend, sollten nur methodenlokal relevante Variablen auch dort deklariert werden.

Eine Klasse oder Struktur kann Instanzenfelder, statische Felder oder beides gemischt verfügen.



Instanzenfelder beziehen sich als Datensatz individuell auf die “eigene” Instanz, statisches Felder gehören zur Klasse selbst und werden von allen Instanzen einer Klasse gemeinsam verwendet. “Lokale” Änderungen, werden

somit übergreifend sichtbar.

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

public class Person{
  string name;
  int index;
  public int Geburtsjahr;
  public static int Count;          // <- Statische Variable Count
  public Person(string name, int geburtsjahr){
    this.name = name;
    Geburtsjahr = geburtsjahr;
    index = Count;
    Count = Count + 1;
  }
  public override string ToString(){
     return name + " ist die " + (index+1).ToString() + " von " + Count.ToString() + " Personen";
  }
}
public class Program
{
  static void Main(string[] args){
    Person Student1 = new Person("Mickey", 1935);
    Console.WriteLine(Student1);
    Person Student2 = new Person("Donald", 1927);
    Console.WriteLine(Student1);
    Console.WriteLine(Student2);
  }
}
```

Felder können mit der Deklaration oder im Konstruktor initialisiert werden. Desweiteren kann mit `readonly` der Wert nach dem Ende des Konstruktorabarbeitung geschützt werden. Eine solche Variable kann als static deklariert werden, um zu vermeiden, dass eine entsprechende Zahl von Kopien erstellt wird.

```
public class Person{
  string name;
  int index = 0;
  readonly string Kategorie = "Student";
  readonly string Hochschule;

  public Person(){
    //...
    Hochschule = "TU Freiberg";
    //...
  }
```

Konstanten

Konstanten sind unveränderliche Datensätze, die zur Kompilierzeit(!) bekannt sind und sich danach nicht mehr verändern lassen. Nur die in C# integrierten Typen - einfache Datentypen und string können als `const` deklariert werden.

Varianten “konstanter” Variablen in C#

	Konstante	Readonly	Readonly statisch
Attribute	`const`	`readonly`	`readonly static`
Veränderbar bis …	Kompilierung	Ende des Konstruktors	Ende des Konstruktors
Statisch	Standard, ja	Nein	Ja
Zugriff	Klasse	Instanz	Instanz

- Versuchen Sie die Variable innnerhalb von Main zu manipulieren
- Wechseln Sie readonly gegen const? Welche Anpassungen müssen Sie vornehmen? –>

```
using System;

public class Person{
  public readonly string name;
  public Person(string name){
    this.name = name;
  }
}
public class Program
{
  static void Main(string[] args){
    Person Student1 = new Person("Mickey");
    Console.WriteLine(Student1.name);
  }
}
```

Konstuktoren

Beim Erzeugen einer Instanz einer `class` oder eines `structs` wird deren Konstruktor aufgerufen. Dieser ist für die Initialisierung der Instanz auf der Zustandsebene verantwortlich. Konstruktoren können überladen werden und verschiedene Signaturen abbilden.

Wenn für eine Klasse kein Konstuktor vorgegeben wird, erstellt der Kompiler standardmäßig einen, der das Objekt instanziiert und Membervariablen auf die Standardwerte festlegt.

```
public class Wine
{
   public decimal Price;
   public int Year;

   // public Wine() // <- Implzit vorhanden, kann aber überschrieben werden
                    // Standardkonstruktor
   public Wine (decimal price){Price = price;}
   public Wine (decimal price, int year) : this (price) {Year = year;}
}
```

Der Standardkonstruktor wird implizit generiert, wenn kein anderer Konstruktor durch den Entwickler spezifiziert wurde. Sofern das geschieht, steht dieser auch nicht mehr bereit.

```
using System;

public class 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);
    Animal tom = new Animal();
```

```
    Console.WriteLine(tom.sound);
  }
}
```

Ein Konstruktor kann einen anderen Konstruktor der gleichen Klasse über das Schlüsselwort `this` aufrufen. Dabei kann der Aufruf mit oder ohne Parameter erfolgen.

```
using System;

class Car
{
  public readonly int NumberOfSeats;
  public readonly int MaxSpeed;
  private int CurrentSpeed;

  public Car(int maxSpeed, int numberOfSeats)
  {
     Console.WriteLine("2 arg ctor");
     this.MaxSpeed = maxSpeed;
     this.NumberOfSeats = numberOfSeats;
  }

  public Car(int maxSpeed) : this(maxSpeed, 5)
  {
     Console.WriteLine("1 arg ctor");
  }

  public Car() : this(100)
  {
     Console.WriteLine("0 arg ctor");
  }
}

public class Program
{
  static void Main(string[] args){
     Car myVehicle = new Car(5);
  }
}
```

Statische Konstruktoren

- … werden verwendet, um static-Daten zu initialisieren oder um eine bestimmte Aktion auszuführen, die nur einmal ausgeführt werden muss.
- … können nicht über Zugriffsmodifizierer oder Parameter verfügen.
- … können nicht vererbt oder überladen werden.
- … werden automatisch vor dem Erzeugen der ersten Instanz ausgeführt und können nicht direkt aufgerufen werden. Damit hat der Nutzer keine Kontrolle, wann der Konstruktor ausgeführt wird.
- … werden kein zweites mal aufgerufen, wenn eine Ausnahme ausgelöst wird.

```
using System;

public class BAFStudent
{
   public static string Universität;
   public string NameStudent;
   static BAFStudent(){
     Console.WriteLine("Universität wird initialisiert");
     Universität = "TU BAF Freiberg";
   }
   public BAFStudent(string name){
```

```
      Console.WriteLine("Name wird initialisiert");
      NameStudent = name;
    }
}

public class Program
{
    static void Main(string[] args){
      BAFStudent student0 = new BAFStudent("Humboldt");
      Console.WriteLine("{0,20} - {1,-10}", BAFStudent.Universität, student0.NameStudent);
      BAFStudent student1 = new BAFStudent("Winkler");
      Console.WriteLine("{0,20} - {1,-10}", BAFStudent.Universität, student1.NameStudent);
    }
}
```

Für die Objektinitialisierung besteht neben den Konstruktoren und dem unmittelbaren Zugriff auf die Membervariablen (vermeiden!) die Möglichkeit direkt nach dem Konstruktoraufruf die Belegung abzubilden.

```
using System;

public class Wine
{
    public decimal Price;
    public int Year;
    public string Vinyard;
    public Wine () {}
    public Wine (decimal price){Price = price;}
    public Wine (decimal price, int year, string vinyard = "Chateau Lafite" ){
      Price = price;
      Year = year;
      Vinyard = vinyard;
    }
    public override string ToString()
    {
      return String.Format("| {0,5} Euro | {1,5} | {2,-18}|", Price, Year, Vinyard );
    }
}

public class Program
{
    static void Main(string[] args){
      // Initalisierung über Standardkonstruktor und direkten Feldzugriff
      Wine bottle0 = new Wine();
      bottle0.Vinyard = "Chateau Latour";
      Console.WriteLine(bottle0);
      // Initialisierung über die Konstruktoren
      Wine bottle1 = new Wine(23);
      Console.WriteLine(bottle1);
      Wine bottle2 = new Wine(3432, 1956);
      Console.WriteLine(bottle2);
      // Initialisierung über Initalizer
      Wine bottle3 = new Wine() {Price = 19, Year = 1910};
      Console.WriteLine(bottle3);
    }
}
```

3 Varianten, und was ist nun besser? Der Aufruf über den Konstruktor ermöglicht die Initialisierung von `readonly` Variablen.

Initalizer werden als atomare Funktion realisiert, sind damit Thread-sicher, sind damit aber auch schwieriger zu debuggen. Zudem können nur `public` Member damit adressiert werden. An dieser Stelle wird deutlich, dass Initializier ggf. beim schnellen Testen Tipparbeit sparen, in realen Anwendungen aber nicht zum Einsatz kommen sollten.

Destruktoren / Finalizer

Mit Finalizern (die auch als Destruktoren bezeichnet werden) werden alle erforderlichen endgültigen Bereinigungen durchgeführt, wenn eine Klasseninstanz vom Garbage Collector gesammelt wird.

```
using System;

public class Person
{
  public string name;
  public Person(string name){this.name = name;}
  ~Person() {Console.WriteLine("The {0} destructor is executing.", ToString());}
}

public class Program
{
  static void Main(string[] args)
  {
    Person Student1 = new Person("Mickey");
    Console.WriteLine(Student1.name);
    Console.WriteLine("Aus die Maus!");
  }
}
```

Der Finalizer ruft implizit die Methode `Finalize` aus der Basisklasse des Typs `Objekt` auf.

Eigenschaften

Eigenschaft (Properties) organisieren den Zugriff auf private Felder über einen flexiblen Mechanismus zum Lesen, Schreiben oder Berechnen des Wertes. Entsprechend können Eigenschaften wie öffentliche Datenmember verwendet werden. Damit wird das Konzept der Kapselung auf effiziente Zugriffsmethoden abgebildet.

Ausgangspunkt:

- Fügen Sie eine Lese / Schreibmethode für die Variable Wochentag ein, die Prüft, ob die Eingabe zwischen Mo = 0 und Freitag = 4 liegt. –>

```
using System;

public class Vorlesung{
  private byte wochentag;
}

public class Program
{
  static void Main(string[] args){
    Vorlesung SoWi = new Vorlesung();
    SoWi.wochentag = 4;
  }
}
```

C# hält, wie andere OOP Sprachen auch dafür eine eigene kompakte Syntax bereit, die Aspekte der Felder und der Methoden kombiniert. Der aufrufende Nutzer sieht eine Feld, der Zugriff kann aber über eine Methode konfiguriert werden. Dabei können durchaus mehrere Eigenschaften auf eine private Variable verweisen.

Für den Benutzer eines Objekts erscheint eine Eigenschaft wie ein Feld; der Zugriff auf die Eigenschaft erfordert dieselbe Syntax.

```
using System;

public class Vorlesung
{
  private byte wochentag;            // Private Variable
  public byte Wochentag              // Öffentliche Variable
  {
```

```
    get { return wochentag; }        // Property accessors
    set {
      if ((value < 7) & (value >= 0))
          wochentag = value;
      else
        Console.WriteLine("Fehlerhafte Eingabe!");
    }
  }
}

public class Program
{
  static void Main(string[] args)
  {
    Vorlesung SoWi = new Vorlesung();
    SoWi.Wochentag = 4;
    Console.WriteLine(SoWi.Wochentag);
  }
}
```

Die Assessoren können beliebig kombiniert werden. Eine Eigenschaft ohne einen `set`-Accessor ist schreibgeschützt. Eine Eigenschaft ohne einen `get`-Accessor ist lesegeschützt.

Zudem lassen sich mit der *Fat Arrow* Notation die Darstellungen wiederum verkürzen. Beispielhaft ist an folgendem Beispiel auch, dass sich die Properties eine vollkommen andere Informationsstruktur bedienen als die eigentlichen privaten Variablen abbilden (= Kapselung).

```
decimal currentPrice, sharesOwned;

public decimal Worth
{
    get => currentPrice * SharesOwned;
    set => sharesOwned = value / currentPrice
}

// Kompakt für

public decimal Worth
{
    get { return currentPrice * SharesOwned; }
    set { sharesOwned = value / currentPrice; }
}
```

`set` verwendet dabei einen impliziten Parameter mit dem Namen `value`, dessen Typ der Typ der Eigenschaft ist.

Und wie sieht es mit dem Zugriffsschutz der Eigenschaften aus? Insbesondere `set` sollte soweit wie möglich eingeschränkt werden. Dafür können `internal`, `private` und `protected` genutzt werden.

Wenn in den Eigenschaftenzugriffsmethoden keine zusätzliche Logik erforderlich ist, bietet sich die Verwendung von automatisch implementierten Eigenschaften.

```
public int CustomerID { get; set; }
```

In diesem Fall erstellt der Compiler ein privates, anonymes, dahinter liegendes Feld, auf das nur über `get` und `set`-Accessoren zugegriffen werden kann.

Indexer

Indexer bilden die Zugriffsmethodik für Arrays `MyArray[3]` auf Klassen ab, um den Zugriff auf Arrays, Listen oder andere Container zu kapseln. Dabei wird folgende Notation benutzt:

```
string [] words = "Das ist ein beispielhafter Text".Split();
//    Typ der Rückgabevariablen
//       |    this Referenz auf das eigene Objekt
```

```
//       |     |    Typ der Indexvariable
//       |     |     |      Bezeichner der Variable
//       v     v     v       v
public string this [int index]{
     get  {return words[index]; }
     set  {words[index] = value; }
}
```

Auch hier wird das Schlüsselwort `value` verwendet, um den Wert zu definieren, der zugewiesen wird.

Indexer müssen nicht durch einen Ganzzahlwert indiziert werden, es können auch andere Typen verwendet werden.

```
using System;


public class Months
{
  string[] months = {"Jan", "Feb", "März", "April", "Mai", "Juni", "Juli",
                     "Aug", "Sep", "Okt", "Nov", "Dez"};

  public string this[byte index]{
    get {return months[index];}
  }
}

public class Program
{
  static void Main(string[] args)
  {
    Months MonthList = new Months();
    Console.WriteLine(MonthList[5]);
  }
}
```

Was ist der Vorteil der Klasse + Indexer Lösung? Wie würden Sie die Indizierung noch absichern?

Operatorenüberladung in C#

Operatoren sind ein Set von Tokens, die grundlegende Operationen für Grunddatentypen beschreiben.

```
int a = 4;
int b = 7;
int c = a + b;                    // + Addition

string s1 = "Hello";
string s2 = "World";
string s3 = s1 + " " + s2;        // + für String Konkatenation
```

Analog zu Methoden werden können Operatoren überladen werden. Entsprechend wird den Operatoren eine spezifische Bedeutung für die Klassen gegeben.

- Operatoren werden in der Klasse überladen
- Operatoren-Überladung ist immer static
- Nutzung des Schlüsselwortes `operator`

Überladbare Operatoren

Opererator	Bedeutug
`+`, `-`, `!`, `~`, `++`, `--`, `true`, `false`	unäre Operatoren, überladbar
`+`, `-`, `\*`, `/`, `%`, `&`, `^`, `<<`, `>>`	binäre Operatoren, überladbar
`==`, `!=`, `<`, `>`, `<=`, `>=`	Vergleichsoperatoren, überladbar
`[]`	nicht überladbar, aber selbe Funktion mit Indexern
`()`	nicht überladbar, aber mittels custom conversion gleiche Funktionalität

Opererator	Bedeutug
+=, -=, *=, /=, %=, &=, ^=, <<=, >>=	Werden durch die zugehörigen binären Operatoren automatisch überladen

Beispiel

```
using System;
using System.Reflection;
using System.ComponentModel.Design;

public class Vector {
  public double X;
  public double Y;

  public Vector (double x, double y){
    this.X = x;
    this.Y = y;
  }

  //public static Vector operator +(Vector p1, Vector p2){
  //  return new Vector(p1.X + p2.X, p1.Y + p2.Y);
  //}

  public static Vector operator -(Vector p1, Vector p2){
    return new Vector(p1.X - p2.X, p1.Y - p2.Y);
  }

  public override string ToString(){
    return "x = " + X.ToString() + ", y = " + Y.ToString();
  }
}

public class Program
{
  static void Main(string[] args)
  {
    Vector a = new Vector (3,4);
    Vector b = new Vector (9,6);
    Console.WriteLine (a+b);
  }
}
```

Die Operatoren += und -= werden dabei automatisch mit überladen.

Merke: Die Typen der Operanden beim Überladen von Operatoren müssen nicht übereinstimmen!

Nehmen wir an, dass wir eine Skalierung r unseres Vektors einfügen wollen und dafür dessen Länge manipulieren.

```
// Es müssen beide Varianten implementiert werden!
public static Point operator *(Point p1, double ratio)
{
  new Point(p1.X * ratio, p1.Y * ratio);
}

public static Point operator *(int ratio, Point p1)
{
  new Point(p1.X * ratio, p1.Y * ratio);
}

static void Main(string[] args)
{
  Point ptOne = new Point(100, 100);
```

```
  Point ptTwo = new Point(40, 40);
}
Console.WriteLine((ptOne * 2.5));
Console.WriteLine((1 * ptOne));
```

Unäre Operatoren (++, –) können in gleicher Art und Weise überschrieben werden.

Wann sind zwei Klasseninstanzen gleich? Müssen alle Inhalte übereinstimmen? Gibt es besondere Felder, deren Übereinstimmung relevanter sind?

class	Felder
Haus	Farbe der Fenster, Markise (ja/nein), Zahl der Räume
Tier	Art, Rasse, Geschlecht
Datei	Typ, Inhalt, Namen

```
using System;
using System.Reflection;
using System.ComponentModel.Design;

public class Vector {
  public double X;
  public double Y;
  public Vector (double x, double y){
    this.X = x;
    this.Y = y;
  }

  public static bool operator ==(Vector p1, Vector p2){
    return (p1.X == p2.X) && (p1.Y == p2.Y);
  }

  public static bool operator !=(Vector p1, Vector p2){
    return (p1.X != p2.X) || (p1.Y != p2.Y);
  }

  //public override bool Equals(object p){
  //  return ???
  //}
}

public class Program
{
  static void Main(string[] args)
  {
    Vector a = new Vector (3,4);
    Vector b = new Vector (9,6);
    Console.WriteLine (a == b);
  }
}
```

Die unären Operatoren `True` und `False` nehmen eine kleine Sonderrolle ein:

```
public static bool operator true(Point p1) => (p1.X>0) && (p1.Y>0);
public static bool operator false(Point p1) => (p1.X < 0) && (p1.Y < 0);

Point pt1 = new Point(10, 10);
if (pt1) Console.WriteLine("true"); // true

// Point is neither true nor false:
Point pt2 = new Point(10, -10);
```

Beispiel der Woche ...

Entwickeln Sie eine Klassenstruktur für die Speicherung der Daten eines Studenten.

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

public class Student
{
  private static int globalerZähler;
  private readonly int uid;
  public string Name { get; set; }
  private bool eingeschrieben;
  private List<string> fächer;

  static Student(){
    globalerZähler = 0;
  }

  public Student(string name)
  {
    Name = name;
    Eingeschrieben = true;
    uid = globalerZähler;
    fächer = new List<string>();
    Console.WriteLine("Der Student {0} (Nr. {1}) ist angelegt!", Name, uid);
    globalerZähler++;
  }

  public bool Eingeschrieben
  {
    get {return eingeschrieben;}
    set
    {
      if (eingeschrieben != value)
        eingeschrieben = value;
      else
      {
        if (value) Console.WriteLine("!Student {0} ist schon eingeschrieben!", Name);
        else Console.WriteLine("!Student {0} ist schon exmatrikuliert!", Name);
      }
     }
  }

  public void addTopic(string Fächername){
    fächer.Add(Fächername);
  }

  public void printTopics(){
    Console.WriteLine("Student {0} hat folgende Fächer absolviert:", Name);
    foreach (string topic in fächer){
      Console.Write(topic + " ");
    }
    Console.WriteLine();
  }
}

public class Program
{
  static void Main(string[] args){
    Student student0 = new Student("Humboldt");
    student0.addTopic("Softwareentwicklung");
```

```
    student0.addTopic("Höhere Mathematik I");
    student0.addTopic("Prozedurale Programmierung");
    student0.printTopics();
    student0.Eingeschrieben = true;
  }
}
```

Aufgaben

- [ ] Setzen Sie sich anhand von Tutorials mit den Konzepten der objektorientierten Programmierung auseinander!

!?alt-text