Objektorientiertes Programmieren (OOP)

Andreas Schärer

EF IF

KSR

Toy-Auftrag

- Schreibe Software für Schule, um sämtliche Personen zu verwalten
- Wird in etwa so gemacht für ISY und EcoReader (Software im Hintergrund)
- Viele Personen an KSR, ca. 645
  - 530 Schüler:Innen
  - 92 Lehrpersonen
  - 32 Mitarbeiter:innen (Mensa, Hausdienst, Front Office, Assistenzen, …)
- Problem: Wie geht man in Code mit so vielen Personen um?

Schlechtes Beispiel 1

- Schlechter Code
  - Umständlich, Fehleranfällig, verliert schnell Überblich
- Probleme:
  - Versch. Infos zu einer Person in verschiedenen Variablen
  - Alle Personen sind praktisch gleich aufgebaut (Name, Alter, …), trotzdem muss gesamter Code für jede Person neugeschrieben werden

```
// LP 1
string scbName = "Bert Schreiner";
string scbAddress = "Hoehenweg 8, 8590 Romanshorn";
string scbBirthday = "07.12.1969";
string scbType = "LP"; // Lehrperson
string scbPos = "HL"; // HauptLehrer

// LP 2
string homName = "Monika Hohler";
string homAddress = "Seeblickstrasse 17, 9306 Freidorf";
string homBirthday = "24.05.1990";
string homType = "LP";
string homPos = "LB1"; // Lehrbeauftragte(r) 1

// LP list with names of all teachers
string[] lpNames = new string[] { scbName, homName };

// STU 1
string verkelleName = "Verena Keller";
string verkelleAddress = "Saentisstrasse 22, 8580 Amriswil";
string verkelleBirthday = "03.02.2007";
string verkelleType = "STU"; // Student
string verkelleClass = "1Fd";

// STU 2
string heimeierName = "Heiri Meier";
string heimeierAddress = "Kreisstrasse 3, 9322 Egnach";
string heimeierBirthday = "11.09.2005";
string heimeierType = "STU"; // Student
string heimeierClass = "3Mfz";

string[] stuNames = new string[] { verkelleName, heimeierName };
```

Schlechtes Beispiel 2

```
string[] Names = new string[] { "Bert Schreiner", "Monika Hohler", "Verena Keller", "Heiri Meier" };
string[] Addresses = new string[] { "Hoehenweg 8, 8590 Romanshorn", "Seeblickstrasse 17, 9306 Freidorf
string[] Ages = new string[] { "07.12.1969", "24.05.1990", "03.02.2007", "11.09.2005" };
string[] Types = new string[] { "LP", "LP", "STU", "STU" };
//string[] Positions = new string[] { }; // ?????
```

- Kompakter, …
- … aber auch nicht viel besser
- Probleme:
  - Nach wie vor: versch. Infos zu einer Person in verschiedenen Variablen
  - Extrem fehleranfällig, da mehrere hundert Personen an Schule
  - Wie umgehen mit Eigenschaften, die nicht für alle Personen gelten?
    - Position der Lehrpersonen
    - Klasse der SuS

I have a dream

- Es wäre doch toll, wenn es eine Möglichkeit gäbe, alles, was zu einer Person gehört, in eine ‘Variable’ packen zu können.

Lösung! OOP

- **Objektorientiertes Programmieren (OOP)!**
- Erstelle eine **Klasse** ‘Person’
- Diese dient dann als **Bauplan** für alle konkreten Personen
- Erstelle in eigenem File
  - File-Name = Name der Klasse
  - In Projektmappe / rechte Maustaste auf Projekt / Hinzufügen / Neue Klasse

```
1   namespace SchoolPersons
2   {
3       public class Person // NAME der Klasse
4       {
5           // ATTRIBUTE / EIGENSCHAFTEN der Klasse
6           public string Name;
7           public string Address;
8           public string Birthday;
9           public string Type;
10
11          // KONSTRUKTOR
12          // wird aufgerufen, wenn ein neues Objekt der Klasse erstellt wird
13          public Person(string _name, string _address, string _birthday, string _type)
14          {
15              Name = _name;
16              Address = _address;
17              Birthday = _birthday;
18              Type = _type;
19          }
20      }
21  }
```

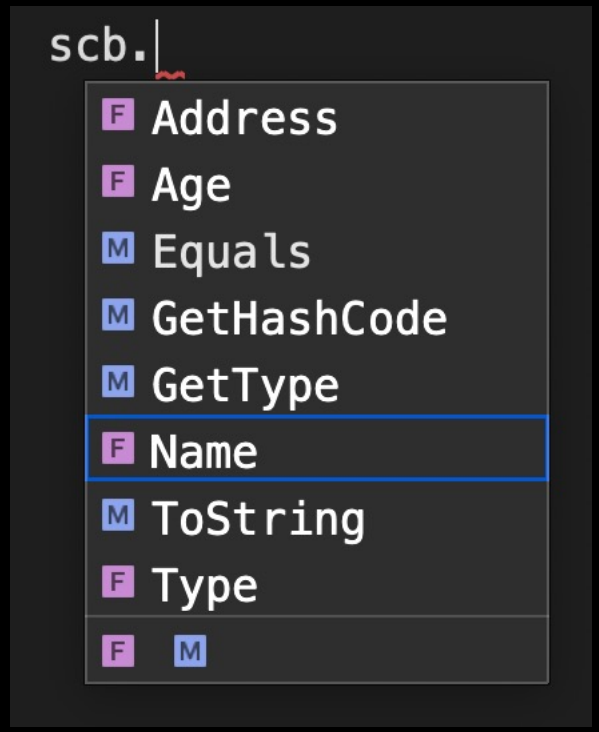
OOP: Neue Instanzen / Objekte

```
public static void Main(string[] args)
{
    Person scb = new Person("Bert Schreiner", "Hoehenweg 8, 8590 Romanshorn TG", "07.12.1969", "LP");
    Person hom = new Person("Monika Hohler", "Seeblickstrasse 17, 9306 Freidorf", "24.05.1990", "LP");
    Person verkelle = new Person("Verena Keller", "Saentisstrasse 22, 8580 Amriswil", "03.02.2007", "STU");
    Person heimeier = new Person("Heiri Meier", "Kreisstrasse 3, 9322 Egnach", "11.09.2005", "STU");
}
```

- Erzeuge dann in Main-Methode **Instanzen der Klasse**, genannt **Objekte**
- Daher: **Objektorientiertes Programmieren**
- Dabei müssen *Argumente* übergeben werden, die vom Konstruktor verlangt werden

OOP: Zugriff auf Eigenschaften

- Zugriff auf Attribute eines Objekts:
  <Name des Objekts>.<Attribute>
- Nutze Autocomplete!
- Beispiel:
  - C#: *Console.WriteLine(scb.Name);*
  - Ausgabe in Konsole: *Bert Schreiner*



OOP: Arrays von Objekten

- Klasse *Person* ist nun wie ein **neuer Datentyp**
- Können auch **Arrays** (und Listen) von diesen *Typ* machen:

```
Person[] PersonsKSR = new Person[] { scb, hom, verkelle, heimeier };
```

- Können so alle Personen der Schule einfach in einem Array speichern
- … und darauf zugreifen:

```
foreach (var p in PersonsKSR)
{
    Console.WriteLine(p.Name);
}
```

- Gibt die Namen sämtlicher Personen im Array an:

```
Bert Schreiner
Monika Hohler
Verena Keller
Heiri Meier
```

Code bisher

Program.cs

```
using System;

namespace SchoolPersons
{
    class MainClass
    {
        public static void Main(string[] args)
        {
            // Create objects / instances of class Person:
            Person scb = new Person("Bert Schreiner", "Hoehenweg 8, 8590 Rom
            Person hom = new Person("Monika Hohler", "Seeblickstrasse 17, 93
            Person verkelle = new Person("Verena Keller", "Saentisstrasse 22
            Person heimeier = new Person("Heiri Meier", "Kreisstrasse 3, 932

            // Save all Person-objects in an array of type Person
            Person[] PersonsKSR = new Person[] { scb, hom, verkelle, heimeie

            // Iterate through all persons in array & print names
            foreach (var p in PersonsKSR)
            {
                Console.WriteLine(p.Name);
            }
        }
    }
}
```

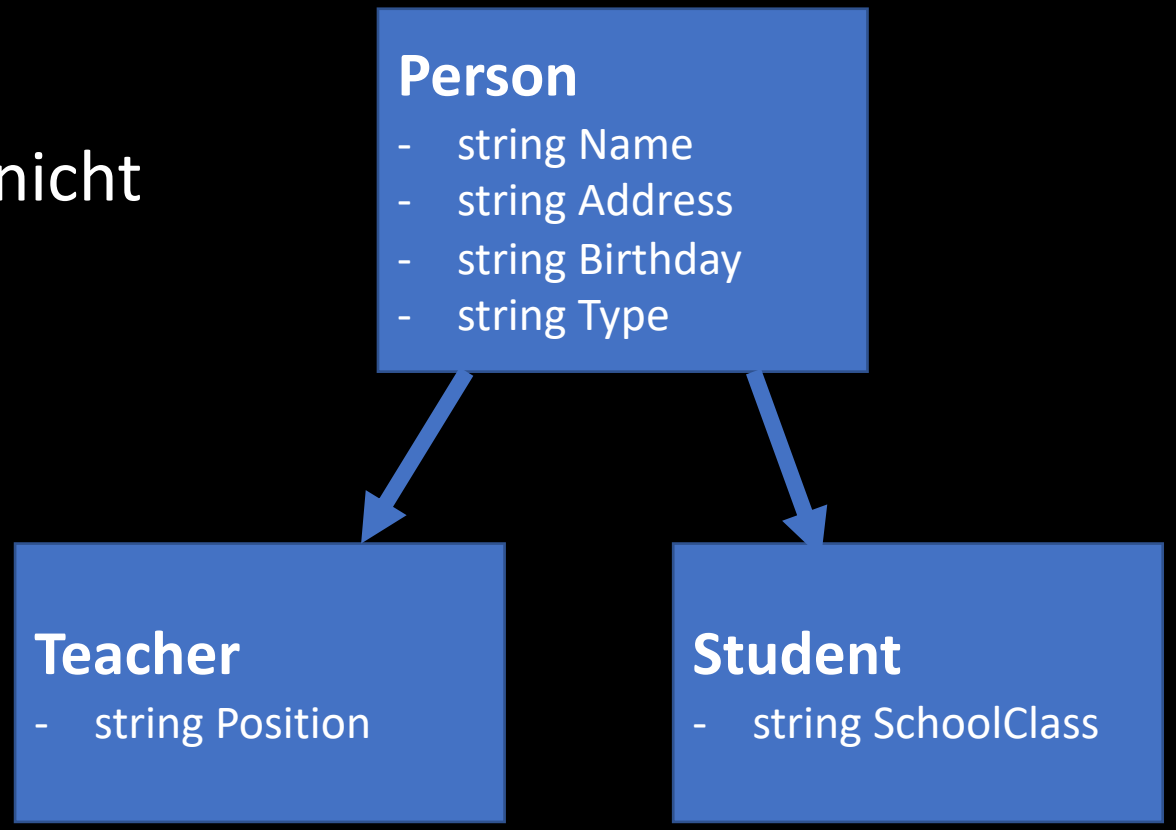
Person.cs

```
namespace SchoolPersons
{
    public class Person // NAME der Klasse
    {
        // ATTRIBUTE / EIGENSCHAFTEN der Klasse
        public string Name;
        public string Address;
        public string Birthday;
        public string Type;

        // KONSTRUKTOR
        // wird aufgerufen, wenn ein neues Objekt der Klasse erstellt wi
        public Person(string _name, string _address, string _birthday, s
        {
            Name = _name;
            Address = _address;
            Birthday = _birthday;
            Type = _type;
        }
    }
}
```

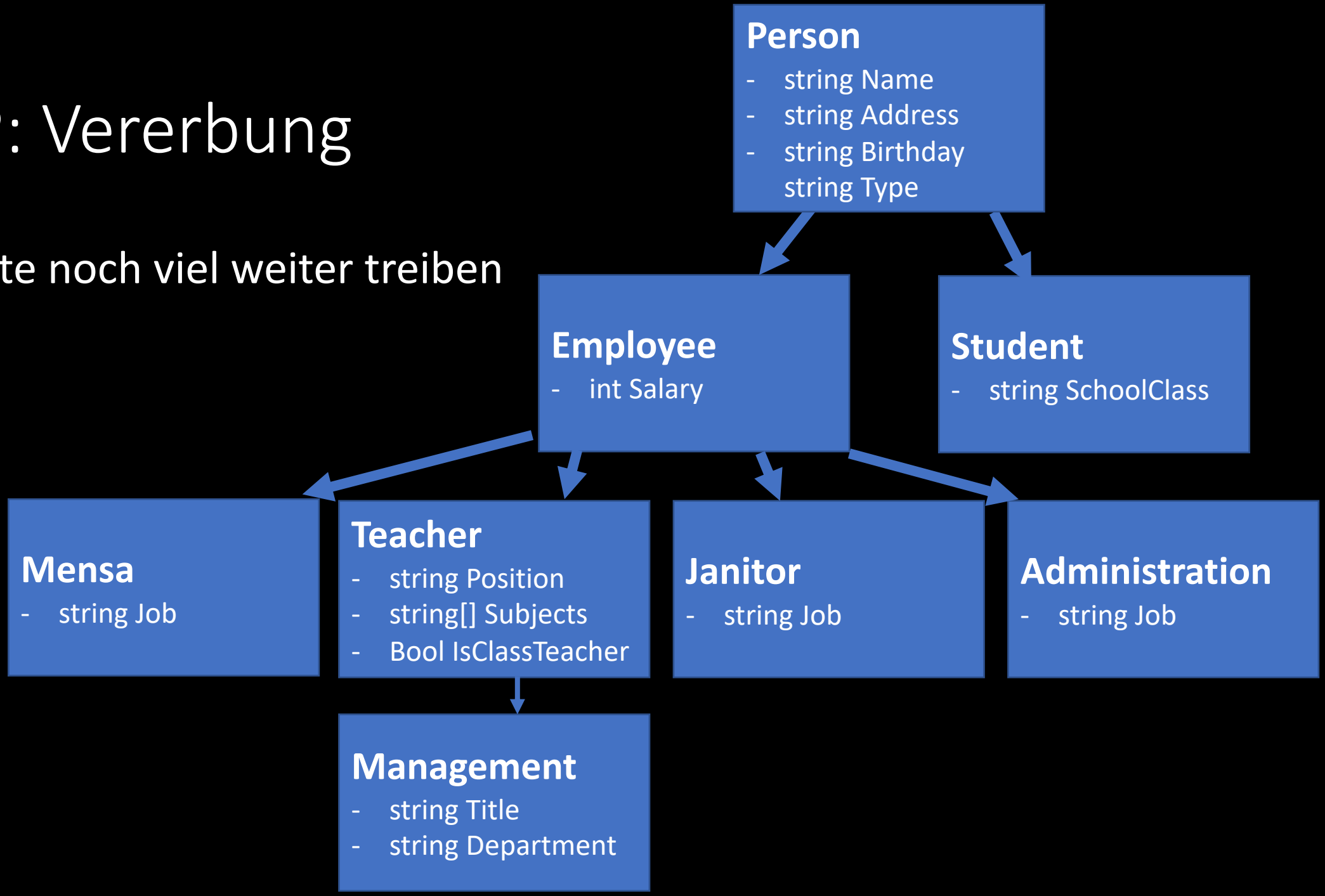
OOP: Vererbung

- Problem: Was ist mit Eigenschaften, die nicht für alle Personen gelten?
  - Position der Lehrpersonen
  - Klasse der SuS
- Lösung: **Vererbung**!
- Erstelle *zwei neue Klassen* **Teacher** und **Student**, die beide *von Person erben*
- Dabei übernehmen sie sämtliche Eigenschaften der Person-Klasse, es können aber neue spezifische hinzugefügt werden



OOP: Vererbung

- Könnte noch viel weiter treiben



OOP: Vererbung

- *zwei neue Klassen* **Teacher** und **Student**, die beide *von Person erben*

```
namespace SchoolPersons
{
    public class Teacher : Person
    {
        public string Position;

        public Teacher(string _name, string _address, string _birthday, string _type, string _position) : base(_name, _address, _birthday, _type)
        {
            Position = _position;
        }
    }
}
```

```
namespace SchoolPersons
{
    public class Student: Person
    {
        public string SchoolClass;

        public Student(string _name, string _address, string _birthday, string _type, string _schoolClass) : base(_name, _address, _birthday, _type)
        {
            SchoolClass = _schoolClass;
        }
    }
}
```

Program.cs

```cs
using System;

namespace SchoolPersons
{
    class MainClass
    {
        public static void Main(string[] args)
        {
            // Create objects / instances of class Person:
            Person scb = new Person("Bert Schreiner", "Hoehenweg 8, 8590 Rom
            Person hom = new Person("Monika Hohler", "Seeblickstrasse 17, 93
            Person verkelle = new Person("Verena Keller", "Saentisstrasse 22
            Person heimeier = new Person("Heiri Meier", "Kreisstrasse 3, 932

            // Save all Person-objects in an array of type Person
            Person[] PersonsKSR = new Person[] { scb, hom, verkelle, heimeie

            // Iterate through all persons in array & print names
            foreach (var p in PersonsKSR)
            {
                Console.WriteLine(p.Name);
            }
        }
    }
}
```

Person.cs

```cs
namespace SchoolPersons
{
    public class Person // NAME der Klasse
    {
        // ATTRIBUTE / EIGENSCHAFTEN der Klasse
        public string Name;
        public string Address;
        public string Birthday;
        public string Type;

        // KONSTRUKTOR
        // wird aufgerufen, wenn ein neues Objekt der Klasse erstellt wird
        public Person(string _name, string _address, string _birthday, string _type)
        {
            Name = _name;
            Address = _address;
            Birthday = _birthday;
            Type = _type;
        }
    }
}
```

erben

Teacher.cs

```cs
namespace SchoolPersons
{
    public class Teacher : Person
    {
        public string Position;

        public Teacher(string _name, string _address, string _birthday, string _type, string _position) : base(_name, _address, _birthday, _type)
        {
            Position = _position;
        }
    }
}
```

Student.cs

```cs
namespace SchoolPersons
{
    public class Student: Person
    {
        public string SchoolClass;

        public Student(string _name, string _address, string _birthday, string _type, string _schoolClass) : base(_name, _address, _birthday, _type)
        {
            SchoolClass = _schoolClass;
        }
    }
}
```

Zusammenfassung bisher

OOP: Methoden

- Bisher nur Klassen mit Eigenschaften
- … können aber auch **Funktionen** (genannt ‘Methoden’) haben
- Beispiel:
  - Von jeder Person ist das Geburtsdatum (string Birthday) bekannt
  - Wäre doch praktisch, wenn man das Alter von allen Personen abfragen könnte
  - Definiere ’Age’ Methode in Klasse Person, die Alter berechnet und zurückgibt
  - Verwende dazu vordefinierte Klasse ‘DateTime’

```
// METHODS
public int Age()
{
    // get age of person
    DateTime bdate = Convert.ToDateTime(Birthday); // convert birthday as string into date format
    int age = DateTime.Now.Subtract(bdate).Days; // get time difference btw today and birthdate in days
    age /= 365; // age in days -> age in years
    return age;
}
```

OOP: Methoden

- Methode im Kontext der Klasse ’Person’

```
1   using System;
2
3   namespace SchoolPersons
4   {
5       public class Person // NAME der Klasse
6       {
7           // ATTRIBUTE / EIGENSCHAFTEN der Klasse
8           public string Name;
9           public string Address;
10          public string Birthday;
11          public string Type;
12
13          // CONSTRUCTOR
14          // gets called, when a new instance of class is created
15          public Person(string _name, string _address, string _birthday, strin
16          {
17              Name = _name;
18              Address = _address;
19              Birthday = _birthday;
20              Type = _type;
21          }
22
23          // METHODS
24          public int Age()
25          {
26              // get age of person
27              DateTime bdate = Convert.ToDateTime(Birthday); // convert birthda
28              int age = DateTime.Now.Subtract(bdate).Days; // get time differe
29              age /= 365; // age in days -> age in years
30              return age;
31          }
32
33      }
34  }
```

OOP: Methoden

- Beispiel:
  - Am Sommerfest darf Alkohol nur an Personen mit Alter 16+ verkauft werden.
  - Erstelle dazu eine Liste mit allen Personen <16
  - Dies wird tatsächlich so gehandhabt
  - Gehe in Main-Methode durch alle Personen in PersonsKSR und gib alle mit Alter < 16 aus:

```
// get all persons with age < 16
foreach (var p in PersonsKSR)
{
    int age = p.Age();
    if (age < 16)
    {
        Console.WriteLine(p.Name);
    }
}
```

```
Verena Keller
```

Properties

- Etwas unschön ist, dass man auf Age über eine Funktion statt ein Attribut (wie z.B. beim Namen) zugreifen muss:

```
int age = p.Age();
string name = p.Name;
```

- Schöner wäre, wenn man auf ‘Age’ wie auf ein Attribut zugreifen könnte (also ohne Klammern) …
- … obwohl es jeweils im Hintergrund berechnet werden muss
- Überraschung: Das geht!

Properties

- Definiere Property *Age*
- getter & setter
- In ***get*** wird festgelegt, was gemacht wird, wenn man Property *Age* abfragt
- Hier: im Hintergrund (in *private* Methode) wird Alter berechnet und ausgegeben
- In ***set*** wird festgelegt, was passiert, wenn man Wert von Age von aussen festlegt. Hier: verboten!

```
public int Age
{
    get
    {
        // determines what user gets when accessing Age, e.g. p.Age
        return convertBirthdayToAge();
    }
    set
    {
        // determines what happens when setting value for p.Age
        // here: nothing, NOT POSSIBLE to set value for Age
        // if want to change Age, have to change Birthday
    }
}
```

```
// METHODS
private int convertBirthdayToAge()
{
    // get age of person
    DateTime bdate = Convert.ToDateTime(Birthday); // convert bi
    int age = DateTime.Now.Subtract(bdate).Days; // get time dif
    age /= 365; // age in days -> age in years
    return age;
}
```

Zugriffsmodifizierer

- Bisher waren alle Attribute & Methoden **public**
- Erste Ausnahme:

```
// METHODS
private int convertBirthdayToAge()
{
    // get age of person
    DateTime bdate = Convert.ToDateTime(Birthday); // convert birthday as string into date format
    int age = DateTime.Now.Subtract(bdate).Days; // get time difference btw today and birthdate in days
    age /= 365; // age in days -> age in years
    return age;
}
```

- Funktion *convertBirthdayToAge* kann **nur innerhalb der Klasse** verwendet werden!
- Auch von einer Child-Class (z.B. Teacher) ist Zugriff nicht möglich. Ausweg: ‘protected'

Zugriffsmodifizierer

Standort des Aufrufers	`public`	`protected internal`	`protected`	`internal`	`private protected`	`private`
Innerhalb der Klasse	✓	✓	✓	✓	✓	✓
Abgeleitete Klasse (selbe Assembly)	✓	✓	✓	✓	✓	✗
Nicht abgeleitete Klasse (selbe Assembly)	✓	✓	✗	✓	✗	✗
Abgeleitete Klasse (andere Assembly)	✓	✓	✓	✗	✗	✗
Nicht abgeleitete Klasse (andere Assembly)	✓	✗	✗	✗	✗	✗

Zugriffsmodifizierer

- **Warum** überhaupt Zugriffsmodifizierer?
- Warum nicht einfach alles ‘public’?
- Funktioniert, macht Code aber *fehleranfällig*
- **Good practice:** Attributen und Methoden immer *möglichst restriktiven Zugriffsmodifizierer* zuweisen

OOP: Abstrakte Klassen

- Sämtliche Personen (Leherpersonen, Schüler:Innen, …) an der Schule erben von der Klasse **Person**
- Es gibt aber niemand, der *nur* vom Typ Person ist, wir schreiben nie *Person per = new Person(".... ") ;*
- Wir können diese Klasse in eine **abstrakte Klasse** umwandeln

```
namespace SchoolPersons
{
    public abstract class Person // NAME der Klasse
    {
        // ATTRIBUTE / EIGENSCHAFTEN der Klasse
        public string Name;
        public string Address;
```

- Damit wird es **verboten (verunmöglicht)**, ein *Objekt* der Klasse Person zu erzeugen
- Von abstrakten Klassen können **keine Objekte erzeugt werden**
- Abstrakte dienen also nur dazu, dass man **von ihnen erben kann**

OOP: Statische Eigenschaften, Methoden & Klassen

- **Statisch** bedeutet, dass etwas **nicht instanziiert** werden kann
- **Statisch Klasse**:
  - Können keine Objekte instanziiert werden.
  - Klasse gibt es also genau 1x – die Klasse selbst
  - Eigenschaften & Methoden einer statischen Klasse müssen statisch sein
  - Beispiel: Klasse *School*

```
1   namespace SchoolPersons
2   {
3       public static class School
4       {
5           public static Person[] All;
6           public static Person[] Teachers;
7           public static Person[] Students;
8           public static string name = "KSR";
9           public static string address = "Weitenzelgstrasse
10      }
11  }
```

```
public static void Main(string[] args)
{
    // Create objects / instances of class Person:
    Person scb = new Person("Bert Schreiner", "Hoehenweg 8,
    Person hom = new Person("Monika Hohler", "Seeblickstras
    Person verkelle = new Person("Verena Keller", "Saentiss
    Person heimeier = new Person("Heiri Meier", "Kreisstras

    School.Students = new Person[] { verkelle, heimeier };
    School.Teachers = new Person[] { scb, hom };
```

Das Gameframework

Auftrag

- Ziel: Sammlung mit Konsolengames erstellen
- Jede Gruppe steuert eine Game dazu bei
- Schwierigkeit: Wie kann man sicherstellen, dass die einzelnen Games zusammengeführt werden können?
- Lösung:
  - Framework wird bereitgestellt (aktuellste Version auf Wiki)
  - Jede Gruppe stellt ihr Game in Form einer Klasse bereit …
  - … die sich an gewisse Vorgaben hält!

Abstrakte Klasse Game

- **Vorgaben**, an die sich jedes Spiel halten muss, werden in **abstrakter Klasse Game** festgelegt
- Jede Spiel-Klasse muss von dieser **erben** …
- … und die dort festgelegten Eigenschaften und Methoden implementieren
- Es können (sollen) **weitere Eigenschaften und Methoden** geschrieben werden, diese müssen aber **private** sein!

```
5   namespace ConsoleGames
6   {
7       public abstract class Game
8       {
9           // Eigenschaften (properties)
10          public abstract string Name { get; } // 'get' means: can be viewed publi
11          public abstract string Description { get; }
12          public abstract string Rules { get; }
13          public abstract int LevelMax { get; }
14          public abstract bool TheHigherTheBetter { get; }
15          public abstract Score HighScore { get; set; } // get; set means: can bot
16
17          // Methoden (methods)
18          public abstract Score Play(int level);
19          /*
20           *  method to play game
21           *  Argument: takes object of class Score that describes previous high s
22           *  Returns: Score object that contains info about score achieved.
23           *  If player is not successful (fail or quit), set score.points=-1.
24           *  If sucessfull, score.points is positive int representing score achie
25           */
26      }
27  }
```

BeiSpiel: GuessNumber

- Jedes Spiel muss vorgegebene Eigenschaften & Methoden implementieren
- Dazu müssen diejenigen der abstrakten Klasse Game *überschrieben* werden
- Deshalb: Keyword **override**
- **Zusätzliche Eigenschaften**:
  - Z.B. int[] levelBoundary
  - Muss **private** sein!

```
3   namespace ConsoleGames.Games
4   {
5       public class GuessNumber : Game
6       {
7           public override string Name => "Guess Number";
8           public override string Description => "Gegeben ist eine Zahl zw
9           public override string Rules => "[HERE EXPLAIN RULES OF GAME]";
10          public override bool TheHigherTheBetter => false;
11          public override int LevelMax => 4;
12
13          public override Score HighScore { get; set; }
14
15          private int[] levelBundary = new int[4] { 51, 101, 151, 201 };
16          private int?[] maxAttempts = new int?[4] { 7, 10, 15, null };
17
18          public override Score Play(int level = 1)
19          {
20              Score score = new Score();
21              score.LevelCompleted = false;
22
23              Random random = new Random();
24              int nrToGuess;
25              int guess = 0;
26              int attempts = 0;
27
28              if (level > LevelMax)
29              {
30                  level = LevelMax;
31              }
32
33              // set number range for levels
34              nrToGuess = random.Next(1, levelBundary[level - 1]);
35
36              Console.Clear();
37
38              if (level <= 3)
39              {
40                  Console.WriteLine("Guess a number! Level: "+level);
```

Eigenes Spiel in Framework einbinden

- Framework herunterladen und öffnen
- Neue Klasse mit Namen des Spiels im Ordner *Game* erstellen
- Diese von abstrakter Klasse Game erben lassen:

```
public class GuessNumber : Game
```

- Vorgegebenen Eigenschaften und Methoden implementieren (mit override-Keyword)
- In Program.cs einbinden und dem *gameArray* hinzufügen

```
10      public static class Program
11      {
12          static Game guessNumber = new Games.GuessNumber();
13
14          static Game[] gameArray = new Game[] { guessNumber };
```

Weitere Infos, Daten, …

- Auf dem Wiki!