Vokabeln

SEW - Softwareentwicklung

htlkrems Bautechnik & IT	**Übungsdatum:** KW /2021 – KW /2021	**Klasse:** 3AHIT	**Name:** Felix Schneider
	Abgabedatum: dd.mm.yyyy	**Gruppe:** SEW	**Note:**
Leitung: Dipl.-Ing.(FH) BIRNZAIN Birgit	**Mitübende:** -		

Übungsbezeichnung:

Vokabelliste

Inhaltsverzeichnis:

1 Theoretische Grundlagen 4
1.1 Deklaration 4
1.2 Initialisierung 4
1.3 Instanziierung 4
1.4 Implementierung 4
1.5 Arrays 4
1.5.1 1-dimensionale Arrays 4
1.5.1.1 Deklaration 4
1.5.1.2 Initialisieren 4
1.5.1.3 Speichern 5
1.5.1.4 Auslesen 5
1.5.2 2-dimensionale Arrays 5
1.5.2.1 Deklaration 5
1.5.2.2 Initialisierung 5
1.5.2.3 Speichern 5
1.5.2.4 Auslesen 5
1.5.3 3-dimensionale Arrays 6
1.5.3.1 Deklaration 6
1.5.3.2 Initialisierung 6
1.5.3.3 Speichern 6
1.5.3.4 Auslesen 6
1.5.4 Jagged Arrays (Verwendung (z.B.): Kalender) 7
1.5.4.1 Deklaration 7
1.5.4.2 Initialisierung 7
1.5.4.3 Speichern 7
1.5.4.4 Auslesen 7
1.6 Enum 7
1.7 Funktion 8
1.8 Klasse vs Instanz vs Object 9
1.9 Vererbung 9
1.10 Polymorphie 10
1.11 Datenkapselung 11
1.11.1 private 11

1.11.2 protected 11
1.11.3 public 11
1.12 Konstruktor 12
1.13 StreamWriter + StreamReader 13
1.14 Windows Forms 14
1.15 Sortieren und Suchen 14
1.16 Indexer 16
1.17 abstrakte Klassen 17
1.18 partial Klassen 19
1.19 Interface 19
1.20 MVC 20

1 Theoretische Grundlagen

1.1 Deklaration

Als **Deklaration** bezeichnet man das Erstellen einer Variablen.

```
string x;
```

1.2 Initialisierung

Als **Initialisierung** bezeichnet man die erstmalige Wertezuweisung einer Variablen.

```
x = "Text";
```

1.3 Instanziierung

Als **Instanziierung** bezeichnet man das Erstellen einer Instanz einer Klasse.

```
Person p = new Person();
```

1.4 Implementierung

Als **Implementierung** bzw. Implementation bezeichnet man das Erstellen von Quellcode bzw. das Umsetzen eines Softwareentwurfs in einem Programm.

```
// Ich will die Variable x in der Console ausgeben.
Console.WriteLine(x); // <- Implementation der Anforderung
```

1.5 Arrays

Arrays anhand von Beispielen erklärt

1.5.1 1-dimensionale Arrays

1.5.1.1 Deklaration

Das Deklarieren eines 1-dimensionalen Arrays

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

1.5.1.2 Initialisieren

Das Initialisieren eines 1-dimensionalen Arrays (Deklaration integriert)

```
int[] myArray = { 62, 54, 14, 86, 38, 54, 49, 8, 35, 99 };
```

1.5.1.3 Speichern

Das Speichern eines Wertes in ein Element eines 1-dimensionalen Arrays

```
myArray[5] = 48;
```

1.5.1.4 Auslesen

Das Auslesen aller Werte eines 1-dimensionalen Arrays in einer Schleife (am besten in einer foreach-Schleife oder einer for-Schleife)

```
foreach (var item in myArray)
{
    Console.Write(item + " ");
}
```

1.5.2 2-dimensionale Arrays

1.5.2.1 Deklaration

Das Deklarieren eines 2-dimensionalen Arrays

```
int[,] myArray = new int[3, 3];
```

1.5.2.2 Initialisierung

Das Initialisieren eines 2-dimensionalen Arrays (Deklaration integriert)

```
int[,] myArray = new int[,] {
    { 1, 2, 3 },
    { 4, 5, 6 },
    { 7, 3, 9 }
};
```

1.5.2.3 Speichern

Das Speichern eines Wertes in ein Element eines 2-dimensionalen Arrays

```
myArray[2, 1] = 8;
```

1.5.2.4 Auslesen

Das Auslesen aller Werte eines 2-dimensionalen Arrays in einer Schleife

```
foreach (var item in myArray)
{
    Console.Write(item + " ");
}
```

1.5.3 3-dimensionale Arrays

1.5.3.1 Deklaration

Das Deklarieren eines 3-dimensionalen Arrays

```
int[,,] myArray = new int[2, 2, 2];
```

1.5.3.2 Initialisierung

Das Initialisieren eines 3-dimensionalen Arrays (Deklaration integriert)

```
int[,,] myArray = new int[,,] {
    {
        { 1, 5 },
        { 3, 4 }
    },
    {
        { 5, 6 },
        { 7, 8 }
    }
};
```

1.5.3.3 Speichern

Das Speichern eines Wertes in ein Element eines 3-dimensionalen Arrays

```
myArray[0, 0, 1] = 2;
```

1.5.3.4 Auslesen

Das Auslesen aller Werte eines 3-dimensionalen Arrays in einer Schleife

```
foreach (var item in myArray)
{
    Console.Write(item + " ");
}
```

1.5.4 Jagged Arrays (Verwendung (z.B.): Kalender)

1.5.4.1 Deklaration

Das Deklarieren eines Jagged Arrays

```
int[][] myArray = new int[3][];
```

1.5.4.2 Initialisierung

Das Initialisieren eines Jagged Arrays (Deklaration integriert)

```
myArray[3] = new int[4];
```

```
int[][] myArray = new int[][]
{
    new int[] { 1, 3, 5, 7, 9 },
    new int[] { 0, 2, 4, 6 },
    new int[] { 11, 22 }
};
```

1.5.4.3 Speichern

Das Speichern eines Wertes in ein Element eines Jagged Arrays

```
myArray[3][4] = 7;
```

1.5.4.4 Auslesen

Das Auslesen aller Werte eines Jagged Arrays in zwei Schleifen

```
foreach (var item in myArray)
{
    foreach (var element in item)
    {
        Console.Write(element + " ");
    }
    Console.WriteLine();
}
```

1.6 Enum

Ein **Enum** ist eine Art boolescher Wert, mit der Ausnahme, dass es nicht nur 2 verschiedene Werte (true/false) annehmen kann, sondern eine bestimmte Anzahl an definierten Werten. Zum Beispiel

kann man ein Enum erstellen, in dem alle 4 Jahreszeiten vorkommen. Eine Instanz dieses Enums kann also nur diesen 4 Werten entsprechen.

```
0 Verweise
class Program
{
    2 Verweise
    enum Jahreszeit
    {
        Frühling, Sommer, Herbst, Winter
    }

    0 Verweise
    static void Main(string[] args)
    {
        Jahreszeit jahreszeit = Jahreszeit.Herbst;
        Console.WriteLine((int)jahreszeit);
        // Ausgabe: 2
    }
}
```

1.7 Funktion

Eine **Funktion** ist eine statische Methode, eine Methode die keine Instanz benötigt, um verwendet werden zu können.

```
0 Verweise
class Program
{
    1 Verweis
    public static double SquareRoot(int number)
    {
        return Math.Sqrt(number);
    }

    0 Verweise
    static void Main(string[] args)
    {
        int number = 16;
        double number_sqrt = SquareRoot(number);
        Console.WriteLine(number_sqrt);
    }
}
```

1.8 Klasse vs Instanz vs Object

Eine **Klasse** ist der Bauplan einer **Instanz**. Sie besteht aus den beiden Teilen Status und Verhalten, die durch Attribute und Eigenschaften und Methoden und Funktionalität umgesetzt werden. Das Objekt ist in der objektorientierten Programmierung die Instanz, die nach dem Bauplan der Klasse *gemalt* wurde.

Instanz / Objekt:

```
Person p = new Person();
```

Klasse:

```
1 Verweis
class Person
{
    public string Name;

    1 Verweis
    public void Hello()
    {
        Console.WriteLine("Hello");
    }
}
```

1.9 Vererbung

Vererbung: Eine Klasse (ChildClass) kann von einer anderen Klasse (BaseClass) erben. Das bedeutet, dass alle Variablen, Methoden und Konstruktoren von der BaseClass auch in allen ChildClasses, GrandChildClass […] verfügbar sind, wenn deren Modifizierer nicht gerade auf *private* gesetzt ist. Eine ChildClass kann nicht von mehreren BaseClasses erben, allerdings von mehreren Interfaces und einer BaseClass.

```
1 Verweis
class Student : Person
{
    1 Verweis
    public string GetName()
    {
        return Name;
    }
}
```

1.10 Polymorphie

Als **Polymorphie** wird folgendes bezeichnet: Wenn die BaseClass eine Methode mit dem Schlüsselwort *virtual* besitzt, so muss in der ChildClass die Methode mit dem Schlüsselwort *override* neu implementiert werden. Falls man nun eine Instanz von der ChildClass bildet und die Methode aufruft, so wird nicht die BaseClass-Methode sondern die ChildClass-Methode aufgerufen, weil die ChildClass-Methode die BaseClass-Methode überschreibt.

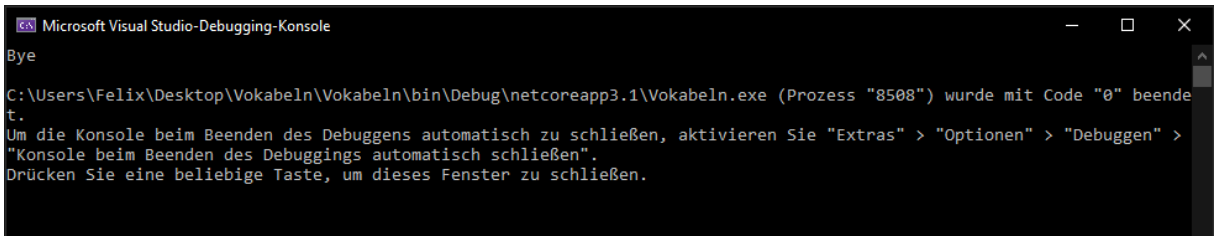
Wenn man statt dem Schlüsselwort *override* das Schlüsselwort *new* verwendet, kommt es zu einem nicht polymorphen Verhalten. Eine Referenz der BaseClass, das auf die ChildClass zeigt, nutzt deshalb nicht die Methode der ChildClass, sondern die Methode der BaseClass. Das Schlüsselwort *virtual* ist in diesem Fall nicht mehr notwendig.

```
0 Verweise
class Program
{
    0 Verweise
    static void Main(string[] args)
    {
        Person st = new Student();

        st.Hello();
    }
}
```

```
2 Verweise
class Person
{
    2 Verweise
    public virtual void Hello()
    {
        Console.WriteLine("Hello");
    }
}
```

```
0 Verweise
class Student : Person
{
    2 Verweise
    public override void Hello() // polymorphes Verhalten
    {
        Console.WriteLine("Bye");
    }
}
```

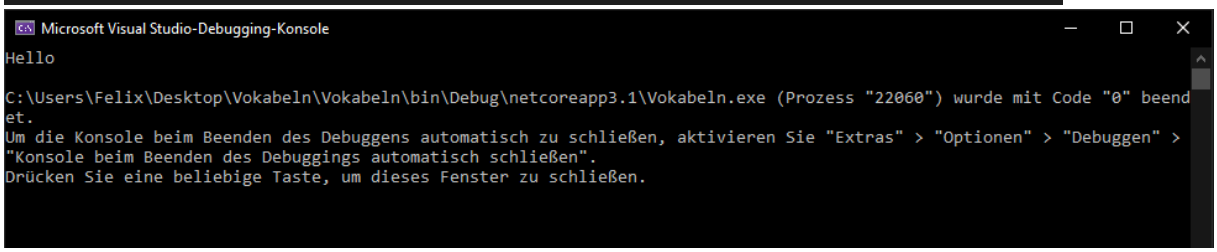


polymorph ↑

vs

nicht polymorph ↓

```
0 Verweise
class Student : Person
{
    0 Verweise
    public new void Hello() // nicht polymorphes Verhalten
    {
        Console.WriteLine("Bye");
    }
}
```



1.11 Datenkapselung

Als **Datenkapselung** bezeichnet man das Verbergen von Variablen vor dem Zugriff von außen.

1.11.1 private

private kapselt die Daten von jeglichem Zugriff außerhalb der Klasse ab

1.11.2 protected

protected kapselt die Daten von Zugriffen aus dem Hauptprogramm ab. ChildClasses von der Klasse haben jedoch Zugriff auf die Daten

1.11.3 public

public kapselt die Daten nicht ab

```
2 Verweise
class Person
{
    private string text1; // nur hier
    protected string text2; // auch in ChildClasses
    public string Text3; // überall
}
```

1.12 Konstruktor

Konstruktoren dienen der bequemen und schnellen Einlesung von Parametern, die beim Instanziieren einer Klasse direkt gespeichert werden können. In einer Klasse können mehrere Konstruktoren verwendet werden, diese dürfen jedoch keine identischen Parameterlisten haben.

Konstruktorketten ermöglichen das Aufrufen von einem Programmcode in einem anderen Konstruktor, wodurch kein Copy&Paste-Code entsteht. Dafür benötigt man das Schlüsselwort *this*.

Will man in einer ChildClass den Konstruktor der BaseClass aufrufen, benötigt man das Schlüsselwort *base*.

```
4 Verweise
class Person
{
    protected string name;

    0 Verweise
    public Person()
    {
        Console.WriteLine("Nothing to do here!");
    }

    0 Verweise
    public Person(string name)
    {
        this.name = name;
    }
}
```

```
5 Verweise
class Person
{
    protected string name;
    protected int age;

    0 Verweise
    public Person():this("unnamed", 0) { }

    0 Verweise
    public Person(string name):this(name, 0) { }

    2 Verweise
    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
}
```

```
2 Verweise
class Student : Person
{
    protected int iq;

    1 Verweis
    public Student(string name, int age):this(name, age, 100) { }

    2 Verweise
    public Student(string name, int age, int iq):base(name, age)
    {
        this.iq = iq;
    }
}
```

1.13 StreamWriter + StreamReader

Um Daten zu speichern (selbst, wenn man das Programm beendet) benötigt man einen **StreamWriter**. Dieser schreibt die Daten, die man speichern will, in eine beliebige Datei.

```
using System.IO;

namespace Vokabeln
{
    0 Verweise
    class Program
    {
        0 Verweise
        static void Main(string[] args)
        {
            string text = "IMPORTANT DATA";

            using(StreamWriter sw = new StreamWriter(@"some/path/on/your/pc/data.csv"))
            {
                sw.WriteLine(text);
            }
        }
    }
}
```

Damit man den StreamWriter nutzen kann, benötigt man außerhalb des namespaces den Befehl **using System.IO;**. Im Main malt man dann den StreamWriter. Das @ Zeichen vor den Anführungszeichen sorgt dafür, dass man keinen Doppelten-Slash beim Angeben des absoluten Pfades benötigt. Anstatt Console.WriteLine(); zu schreiben, schreibt man die Daten dann mithilfe des Names des StreamWriters und WriteLine oder Write in die Datei. CSV-Dateien sind normale Textdateien, die die verschiedenen Inhalte mittels Semikolon trennen.

Um diese wichtigen Informationen beim nächsten Start des Programms wieder aus der Datei herauszulesen benötigt man einen **StreamReader**.

```
using(StreamReader sr = new StreamReader(@"some/path/on/your/pc/data.csv"))
{
    while(sr.Peek() > -1)
    {
        text = sr.ReadLine();
    }
}
```

1.14 Windows Forms

Mithilfe von **WindowsForms** kann man Apps programmieren. Man kann Buttons, Textboxen, Checklisten

1.15 Sortieren und Suchen

Sortieren + Suchen: Es gibt unzählige verschiedene **Sortierverfahren**, um nur ein paar zu nennen: Bubblesort, Insertionsort, Mergesort, Quicksort, Selectionsort und auch das Sortierverfahren, das Microsoft Visual Studio bereits implementiert hat: myArray.Sort(); Letzteres ist das schnellste Sortierverfahren. Es ist schneller als Bubblesort (was nicht wirklich schwierig ist), schneller als Selectionsort und sogar doppelt so schnell wie Mergesort, was eines der schnellsten OpenSource

Sortierverfahren ist. Apropos OpenSource: Microsoft will uns nicht sagen, wie sie es geschafft haben ihr Sortierverfahren so schnell zu machen…

Als Demonstration habe ich einfach einmal **Insertionsort** als Sortierverfahren gewählt, weil es nicht so komplex ist, dass es Schüler nicht mehr verstehen, aber immerhin doppelt so schnell wie Bubblesort ist:

```
0 Verweise
class Program
{
    static Random randy = new Random();
    0 Verweise
    static void Main(string[] args)
    {
        int[] myArray = new int[100];
        for (int i = 0; i < myArray.Length; i++)
        {
            myArray[i] = randy.Next(1, 100);
        }
        Insertionsort(myArray);
        foreach (var item in myArray)
        {
            Console.Write(item + " ");
        }
    }

    1 Verweis
    static void Insertionsort(int[] myArray)
    {
        for (int i = 0; i < myArray.Length; i++)
        {
            int value = myArray[i];
            int j = i;
            while (j > 0 && myArray[j - 1] > value)
            {
                myArray[j] = myArray[j - 1];
                j--;
            }
            myArray[j] = value;
        }
    }
}
```

Insertionsort nimmt immer den nächsten Wert in der unsortierten Liste und sortiert diesen an der richtigen Stelle in der bereits geordneten Liste ein.

Ein sehr beliebtes **Suchverfahren** funktioniert folgendermaßen:
Nehmen wir an, wir suchen aus einer geordneten Liste mit zufälligen Zahlen von 1 bis 1000 die Zahl 423. Zuerst schauen wir nach, ob der Wert, der in der Mitte der geordneten Liste steht größer oder kleiner ist als unsere gesuchte Zahl. Je nachdem ob er größer oder kleiner ist, sehen wir wieder nach ob der Wert in der Mitte der oberen oder unteren Hälfte wieder größer oder kleiner als die gesuchte Zahl ist. Das machen wir solange, bis wir den Wert gefunden haben. Natürlich muss man auch bedenken, dass es die Zahl gar nicht geben könnte.

1.16 Indexer

Verwendet man in einer Klasse ein Array und möchte dieses beim Main direkt mit dem Klassennamen aufrufen können, dann benötigt man dafür einen **Indexer** / **Indizierer**.

```
0 Verweise
class Program
{
    0 Verweise
    static void Main(string[] args)
    {
        Team t = new Team(4);

        t[0] = new Player("David", "Alalba");
        // ohne Indexer (public Player[]): t.myPlayers[0] = ...
        t[1] = new Player("Christiano", "Ronaldo");
        t[2] = new Player("Lionel", "Messi");
        t[3] = new Player("Kylian", "Mbappé");
    }
}
```

```
4 Verweise
class Player
{
    string firstName, lastName;

    4 Verweise
    public Player (string f, string l)
    {
        firstName = f;
        lastName = l;
    }
    0 Verweise
    public override string ToString()
    {
        return firstName + " " + lastName;
    }
}
```

```
2 Verweise
class Team
{
    Player[] myPlayers;

    1 Verweis
    public Team(int size)
    {
        myPlayers = new Player[size];
    }

    // Indexer
    4 Verweise
    public Player this[int idx]
    {
        get
        {
            if (idx <= myPlayers.Length && idx >= 0)
                return myPlayers[idx];
            else throw new Exception("Team doesn't have so many members!");
        }
        set
        {
            if (idx <= myPlayers.Length && idx >= 0)
                myPlayers[idx] = value;
            else throw new Exception("Player doesn't fit into team!");
        }
    }
}
```

1.17 abstrakte Klassen

Es ist möglich einer Klasse das Schlüsselwort *abstract* zu geben. Dies bewirkt, dass keine Instanz dieser BaseClass gebildet werden kann. Es können nur Instanzen der ChildClasses gebildet werden. Ein möglicher kluger Beispielfall wäre hier die BaseClass Tier, von der viele ChildClasses erben, wie zum Beispiel Vogel, Bär oder Hund. Dann kann man nämlich nur Instanzen von spezifischen Tierarten bilden, jedoch nicht von Tier. Abstrakte Klassen beginnen zur leichteren Lesbarkeit mit einem großen A.

```
0 Verweise
class Program
{
    0 Verweise
    static void Main(string[] args)
    {
        Tier t = new Tier();

        Hund h = new Hund();

        Tier k = new Katze();
    }
}
```

```
2 Verweise
abstract class ATier
{
    1 Verweis
    protected string Name { get; set; }
    1 Verweis
    protected int Alter { get; set; }

    5 Verweise
    public override string ToString()
    {
        return Name + ", " + Alter;
    }
}
```

```
1 Verweis
class Hund : ATier
{
    1 Verweis
    public string Hundeart { get; set; }

    5 Verweise
    public override string ToString()
    {
        return base.ToString() + ", " + Hundeart;
    }
}
```

```
0 Verweise
class Katze : ATier
{
    1 Verweis
    public string Fellfarbe { get; set; }

    5 Verweise
    public override string ToString()
    {
        return base.ToString() + ", " + Fellfarbe;
    }
}
```

1.18 partial Klassen

Außerdem kann eine Klasse das Schlüsselwort *partial* bekommen. Dies ermöglicht das Aufteilen einer Klasse in mehrere Dateien.

1.19 Interface

Ein **Interface** ist die Schnittstelle einer Klasse. Es ermöglicht ein leichteres Zusammenarbeiten, da jeder Beteiligte weiß, welche Methoden und Variablen beim jeweiligen Programmteil implementiert werden sollen. Im Gegensatz zu einer *abstrakten* Klasse jedoch werden nicht die Gemeinsamkeiten (zum Beispiel von Fahrzeugen) generalisiert (jedes Fahrzeug hat eine Marke und ein Baujahr und kann „fahren“) sondern das Interface erfasst **Funktionalitäten**, zum Beispiel können sich Fahrzeuge fortbewegen ebenso auch Personen. Deshalb benötigt man ein Interface „**Move**“ und keine *abstrakte* Klasse. Interface-Namen beginnen zur leichteren Lesbarkeit mit einem großen I (kein L).

```
2 Verweise
public interface IMoveable
{
    2 Verweise
    void Move();
}

0 Verweise
public class Car : IMoveable
{
    2 Verweise
    public void Move()
    {
        Console.WriteLine("Das Auto fährt.");
    }
}

0 Verweise
public class Human : IMoveable
{
    2 Verweise
    public void Move()
    {
        Console.WriteLine("Der Mensch geht.");
    }
}
```

1.20 MVC

Das **MVC-Modell** besteht aus den drei Teilen: Model, View und Controller. Mit Model ist das Lesen und Schreiben in Dateien wie zum Beispiel txt oder csv gemeint (mehr dazu in der 4ten Klasse), View ist der Bereich, den der Benutzer sieht, also entweder die Console oder Windows Forms. Der Controller ist der Bereich wo die gesamte Logik stattfindet. Hier befindet sich auch die dll-Datei. Was ist eine dll-Datei? Ganz einfach: Wenn ich die Logik von TicTacToe bereits in einer dll-Datei ausprogrammiert habe, dann muss ich nur noch in Windows Forms den Projektverweis auf diese dll-Datei herstellen und kann schon mit dem nächsten Bereich, dem View fortfahren. Ebenso könnte ich in einer Console diesen Projektverweis herstellen und anschließend den View für die Console programmieren. Das bedeutet in einer dll-Datei steckt rein die Logik einer Erfüllung eines bestimmten Wunsches dahinter. Ob man diese Logik anschließend in Windows Forms, der Console oder HTML einbauen will ist der dll-Datei vollkommen egal. Die Schnittstellen dieser dll-Datei sind Parameter als Eingang und Rückgabewerte als Ausgang.