WBS TRAINING

Programmierung(2)

UNSER ♥ SCHLÄGT FÜR BILDUNG

Agenda

- **Objektorientierte Programmierung (OOP)**
  - Definition
  - Motivation
  - Beispiele für …
    - Kapselung
    - Vererbung
    - Polymorphie
- **Klassendiagramme (UML)**
  - Definition + Motivation
  - Beispiel
- **Fachpraktische Anwendungen**

Objektorientierte Programmierung – Definition

- Die **OOP** ist eine eigenständige Programmier-Philosophie (man spricht auch von einem „**Paradigma**“).
- Sie unterscheidet sich von der klassischen, **prozeduralen** (bzw. „imperativen“) Programmierung, die den eigentlichen Gegenstand dieses Bausteins ausmacht.
- Die Grundidee besteht in dem Ansatz, Programmieraufgaben als Beziehungen zwischen sogenannten „**Objekten**“ zu modellieren.
- Objekte erinnern aus Sicht von ANSI C an **Strukturvariablen**. Aus Sicht von SQL erinnern sie an Datensätze, also an Zusammenfassungen von Informationen, die das Objekt beschreiben.
- Objekte werden als Instanzen von **Klassen** erstellt. Klassen entsprechen aus Sicht von ANSI C einem Strukturtyp. Aus Sicht von SQL entsprechen sie einer Tabellendefinition.
- Im Unterschied zu klassischen Strukturtypen können zu Klassen aber auch **Funktionen** zählen, die in der OOP üblicherweise als „**Methoden**“ bezeichnet werden.

Objektorientierte Programmierung – Motivation

- Mit der Einführung der OOP wurden große Hoffnungen verbunden, die sich allerdings nicht vollständig erfüllten. Dennoch gehört sie neben der klassischen Programmierung zu den **wichtigsten Alternativen**.
- Da in der **IHK-Abschlussprüfung** zumindest elementare Fragen zur OOP auch fachübergreifend gestellt werden, sollten diese auch in diesem Baustein geklärt werden. (Deutlich detailliertere Einsichten werden den angehenden Anwendungsentwicklern im Baustein zu C# vermittelt).
- Die OOP erlaubt das Verfolgen zentraler Ziele, die üblicherweise als „**Säulen der OOP**“ bezeichnet werden. Zu diesen zählen:
  - **Kapselung**
  - **Vererbung**
  - **Polymorphie**

    Einige Autoren nennen auch eine (uneinheitliche) 4. Säule: Generalisierung/Abstraktion(Geheimhaltung)/Relation

- Wir werden uns im Folgenden nur mit den unumstrittenen ersten **3 Säulen** befassen.

  (Zumal die in der 4. Säule angesprochenen Aspekte bereits durch die ersten 3 Säulen aufgegriffen werden)

Objektorientierte Programmierung – Kapselung

- Die Kapselung kann zunächst als ein **Ordnungsprinzip** betrachtet werden: Daten und Methoden, die sich auf die Objekte einer bestimmten Klasse beziehen, können nur von diesen genutzt werden, so dass unzulässige Bezugnahmen vermieden werden.
- Ferner kann die Kapselung auch im Sinne eines „**Geheimnisprinzips**“ genutzt werden, indem dafür gesorgt wird, dass bestimmte Informationen eines Objektes nur diesem Objekt zur Verfügung gestellt werden => „Außerhalb“ der Klasse kann auf diese Informationen nicht zugegriffen werden.

- Dieses Prinzip werden wir uns im Folgenden mit Hilfe eines C#-Codes veranschaulichen:
  - als Beispiel wollen wir eine **Klasse „Ware“** betrachten
  - zu dieser können einzelne **Objekte kreiert** werden
  - deren **Verkaufspreise** sind öffentlich einsehbar
  - Die **Gewinnspanne** ist aber nur für das Objekt selbst verfügbar

    => Das Objekt wird mit dieser Gewinnspanne rechnen können, anschließend aber nur das Ergebnis der Berechnung öffentlich machen, während die Gewinnspanne selbst „**geheim**“ bleibt.

Objektorientierte Programmierung – Kapselung

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    class Ware
    {
        public Ware(int artikelNummer, string warenTyp)
        {
            this.artikelNummer = artikelNummer;
            this.warenTyp = warenTyp;

            if (warenTyp == "Porsche") einkaufspreis = 100000;
            if (warenTyp == "Luftballon") einkaufspreis = 0.2;
            verkaufspreis = einkaufspreis * gewinnProzent;
        }

        public int artikelNummer;
        public string warenTyp;
        public double verkaufspreis;

        private double gewinnProzent = 1.15;
        private double einkaufspreis;

        public void schreibeVerkaufspreis()
        {
            Console.WriteLine("Die Ware mit Artikelnummer "
                        + this.artikelNummer + " ist vom Typ \""
                        + this.warenTyp + "\" und kostet im Verkauf: " + this.verkaufspreis + " €.");
        }
    }
```

WBS TRAINING

Objektorientierte Programmierung – Kapselung

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    class Ware
    {
        public Ware(int artikelNummer, string warenTyp)
        {
            this.artikelNummer = artikelNummer;
            this.warenTyp = warenTyp;

            if (warenTyp == "Porsche") einkaufspreis = 100000;
            if (warenTyp == "Luftballon") einkaufspreis = 0.2;
            verkaufspreis = einkaufspreis * gewinnProzent;
        }

        public int artikelNummer;
        public string warenTyp;
        public double verkaufspreis;

        private double gewinnProzent = 1.15;
        private double einkaufspreis;

        public void schreibeVerkaufspreis()
        {
            Console.WriteLine("Die Ware mit Artikelnummer "
                    + this.artikelNummer + " ist vom Typ \""
                    + this.warenTyp + "\" und kostet im Verkauf: " + this.verkaufspreis + " €.");
        }
    }
```

Hier wird eine **Klasse** (namens „Ware“) **definiert**.
Zwischen den geschwungenen Klammern wird aufgelistet, welche Attribute und Methoden zu dieser Funktion gehören sollen

WBS TRAINING

Objektorientierte Programmierung – Kapselung

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    class Ware
    {
        public Ware(int artikelNummer, string warenTyp)
        {
            this.artikelNummer = artikelNummer;
            this.warenTyp = warenTyp;

            if (warenTyp == "Porsche") einkaufspreis = 100000;
            if (warenTyp == "Luftballon") einkaufspreis = 0.2;
            verkaufspreis = einkaufspreis * gewinnProzent;
        }
```

Die (im Vergleich zum Klassennamen gleichlautende) Funktion Ware() wird als „**Konstruktor**“ bezeichnet.
Sie dient dazu, Objekte vom Typ Ware zu kreieren.

Konstruktoren müssen nicht selbst erstellt werden.
Falls man dies jedoch dennoch tut, so können diese Funktionen z.B. bequeme Wertzuweisungen realisieren.

```
        {
            Console.WriteLine("Die Ware mit Artikelnummer "
                        + this.artikelNummer + " ist vom Typ \""
                        + this.warenTyp + "\" und kostet im Verkauf: " + this.verkaufspreis + " €.");
        }
    }
```

WBS TRAINING

Objektorientierte Programmierung – Kapselung

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    class Ware
    {
        public Ware(int artikelNummer, string warenTyp)
        {
            this.artikelNummer = artikelNummer;
            this.warenTyp = warenTyp;

            if (warenTyp == "Porsche") einkaufspreis = 100000;
            if (warenTyp == "Luftballon") einkaufspreis = 0.2;
            verkaufspreis = einkaufspreis * gewinnProzent;
        }

        public int artikelNummer;
        public string warenTyp;
        public double verkaufspreis;

        private double gewinnProzent = 1.15;
        private double einkaufspreis;

        public void schreibeVerkaufspreis()
        {
            Console.WriteLine("Die Ware mit Artikelnummer "
                        + this.artikelNummer + " ist vom Typ \""
                        + this.warenTyp + "\" und kostet im Verkauf: " + this.verkaufspreis + " €.");
        }
    }
```

Der Zugriffsmodifizierer **public** legt fest, dass die Werte der Variablen auch außerhalb der Klasse ausgelesen werden können.

WBS TRAINING

Objektorientierte Programmierung – Kapselung

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    class Ware
    {
        public Ware(int artikelNummer, string warenTyp)
        {
            this.artikelNummer = artikelNummer;
            this.warenTyp = warenTyp;

            if (warenTyp == "Porsche") einkaufspreis = 100000;
            if (warenTyp == "Luftballon") einkaufspreis = 0.2;
            verkaufspreis = einkaufspreis * gewinnProzent;
        }

        public int artikelNummer;
        public string warenTyp;
        public double verkaufspreis;

        private double gewinnProzent = 1.15;
        private double einkaufspreis;

        public void schreibeVerkaufspreis()
        {
            Console.WriteLine("Die Ware mit Artikelnummer "
                        + this.artikelNummer + " ist vom Typ \""
                        + this.warenTyp + "\" und kostet im Verkauf: " + this.verkaufspreis + " €.");
        }
    }
```

Der Zugriffsmodifizierer **private** legt fest, dass die Werte der Variablen außerhalb der Klasse NICHT ausgelesen werden können.

=> diese Werte sind dann also quasi „**abgekapselt**“

WBS TRAINING

Objektorientierte Programmierung – Kapselung

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    class Ware
    {
        public Ware(int artikelNummer, string warenTyp)
        {
            this.artikelNummer = artikelNummer;
            this.warenTyp = warenTyp;

            if (warenTyp == "Porsche") einkaufspreis = 100000;
            if (warenTyp == "Luftballon") einkaufspreis = 0.2;
            verkaufspreis = einkaufspreis * gewinnProzent;
        }

        public int artikelNummer;
        public string warenTyp;
        public double verkaufspreis;

        private double gewinnProzent = 1.15;
        private double einkaufspreis;

        public void schreibeVerkaufspreis()
        {
            Console.WriteLine("Die Ware mit Artikelnummer "
                        + this.artikelNummer + " ist vom Typ \""
                        + this.warenTyp + "\" und kostet im Verkauf: " + this.verkaufspreis + "
€.");
        }
    }
```

Hier werden 2 **Objekte** vom Typ Ware kreiert

```
class Program
{
    static void Main(string[] args)
    {
        Console.OutputEncoding = Encoding.UTF8;

        Ware w1 = new Ware(1,"Porsche");
        Ware w2 = new Ware(2, "Luftballon");

        w1.schreibeVerkaufspreis();
        w2.schreibeVerkaufspreis();

        // private Werte sind von außen geheim:
        // Console.WriteLine("w1.gewinnProzent="+w1.gewinnProzent);

        Console.ReadKey();
    }
}
```

WBS TRAINING

Objektorientierte Programmierung – Kapselung

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    class Ware
    {
        public Ware(int artikelNummer, string warenTyp)
        {
            this.artikelNummer = artikelNummer;
            this.warenTyp = warenTyp;

            if (warenTyp == "Porsche") einkaufspreis = 100000;
            if (warenTyp == "Luftballon") einkaufspreis = 0.2;
            verkaufspreis = einkaufspreis * gewinnProzent;
        }

        public int artikelNummer;
        public string warenTyp;
        public double verkaufspreis;

        private double gewinnProzent = 1.15;
        private double einkaufspreis;

        public void schreibeVerkaufspreis()
        {
            Console.WriteLine("Die Ware mit Artik
                    + this.artikelNummer + " ist vom Typ \""
                    + this.warenTyp + "\" und kostet im Verkauf: " + this.verkaufspreis + "
€.");
        }
    }
```

Aufruf ist möglich, da die Funktion „schreibeVerkaufspreis“ **public**

```
class Program
{
    static void Main(string[] args)
    {
        Console.OutputEncoding = Encoding.UTF8;

        Ware w1 = new Ware(1,"Porsche");
        Ware w2 = new Ware(2, "Luftballon");

        w1.schreibeVerkaufspreis();
        w2.schreibeVerkaufspreis();

        // private Werte sind von außen geheim:
        // Console.WriteLine("w1.gewinnProzent="+w1.gewinnProzent);

        Console.ReadKey();
    }
}
```

Objektorientierte Programmierung – Kapselung

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    class Ware
    {
        public Ware(int artikelNummer, string warenTyp)
        {
            this.artikelNummer = artikelNummer;
            this.warenTyp = warenTyp;

            if (warenTyp == "Porsche") einkaufspreis = 100000;
            if (warenTyp == "Luftballon") einkaufspreis = 0.2;
            verkaufspreis = einkaufspreis * gewinnProzent;
        }

        public int artikelNummer;
        public string warenTyp;
        public double verkaufspreis;

        private double gewinnProzent = 1.15;
        private double einkaufspreis;

        public void schreibeVerkaufspreis()
        {
            Console.WriteLine("Die Ware mit Artikelnummer "
                        + this.artikelNummer + " ist vom Typ \""
                        + this.warenTyp + "\" und kostet im Verkauf: " + this.verkaufspreis + "
€.");
        }
    }
```

Auskommentiert, da es ansonsten zu einer **Fehlermeldung** käme

```
class Program
{
    static void Main(string[] args)
    {
        Console.OutputEncoding = Encoding.UTF8;

        Ware w1 = new Ware(1,"Porsche");
        Ware w2 = new Ware(2, "Luftballon");

        w1.schreibeVerkaufspreis();
        w2.schreibeVerkaufspreis();

        // private Werte sind von außen geheim:
        // Console.WriteLine("w1.gewinnProzent="+w1.gewinnProzent);

        Console.ReadKey();
    }
}
```

WBS TRAINING

Objektorientierte Programmierung – Vererbung

- Die Vererbung gehört zu den theoretischen Grundpfeilern der OOP, hat aber in der Praxis an Bedeutung verloren (siehe hierzu auch das **Liskovsche-Prinzip**).
- Die Grundidee der Vererbung besteht in der Beobachtung, dass unterschiedliche Klassen gelegentlich als „Sonderfälle“ (**abgeleitete Klassen**, Unterklasse, Subklasse, Kindklasse) einer „allgemeineren Klasse“ (**Basisklasse**, Superklasse, Elternklasse) betrachtet werden können.
- Sofern eine Basisklasse bereits eingeführt wurde, können die abgeleiteten Klassen von dieser „**erben**“, also Code übernehmen, der entsprechend nicht erneut geschrieben werden muss.

- Dieses Prinzip werden wir uns im Folgenden mit Hilfe eines C#-Codes veranschaulichen:
  - als Beispiel wollen wir die Basisklasse **„Tier“** betrachten
  - Von dieser sollen die beiden abgeleiteten Klassen **„Hund“** und „**Schlange“** erben

Objektorientierte Programmierung – Vererbung

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    class Tier
    {
        public Tier(string name)
        {
            this.name = name;
        }

        public string name;
        public int anzahlBeine;

        public void schreibeBeinzahl()
        {
            Console.WriteLine(name + " hat " + anzahlBeine + " Beine.");
        }
    }

    class Schlange : Tier
    {
        public Schlange(string name) : base(name)
        {
            anzahlBeine = 0;
        }
    }

    class Hund : Tier
    {
        public Hund(string name) : base(name)
        {
            anzahlBeine = 4;
        }
    }
```

Objektorientierte Programmierung – Vererbung

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    class Tier
    {
        public Tier(string name)
        {
            this.name = name;
        }

        public string name;
        public int anzahlBeine;

        public void schreibeBeinzahl()
        {
            Console.WriteLine(name + " hat " + anzahlBeine + " Beine.");
        }
    }

    class Schlange : Tier
    {
        public Schlange(string name) : base(name)
        {
            anzahlBeine = 0;
        }
    }

    class Hund : Tier
    {
        public Hund(string name) : base(name)
        {
            anzahlBeine = 4;
        }
    }
```

Die Klasse Tier dient hier als **Basisklasse**:

Alle Tiere haben einen **Namen**.
Alle Tiere haben eine (bestimmte) **Anzahl von Beinen**.
Von allen Tieren wollen wir diese Anzahl **erfahren**.

WBS TRAINING

Objektorientierte Programmierung – Vererbung

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    class Tier
    {
        public Tier(string name)
        {
            this.name = name;
        }

        public string name;
        public int anzahlBeine;

        public void schreibeBeinzahl()
        {
            Console.WriteLine(name + " hat " + anzahlBeine + " Beine.");
        }
    }

    class Schlange : Tier
    {
        public Schlange(string name) : base(name)
        {
            anzahlBeine = 0;
        }
    }

    class Hund : Tier
    {
        public Hund(string name) : base(name)
        {
            anzahlBeine = 4;
        }
    }
```

Die Klassen Schlage und Hund **erben** von Tier.
Beide Klassen **besitzen nun alle Attribute und Methoden** der Basisklasse Tier, ohne dass diese Attribute und Methoden noch einmal codiert werden müssen.

WBS TRAINING

Objektorientierte Programmierung – Vererbung

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    class Tier
    {
        public Tier(string name)
        {
            this.name = name;
        }

        public string name;
        public int anzahlBeine;

        public void schreibeBeinzahl()
        {
            Console.WriteLine(name + " hat " + anzahlBeine + " Beine.");
        }
    }

    class Schlange : Tier
    {
        public Schlange(string name) : base(name)
        {
            anzahlBeine = 0;
        }
    }

    class Hund : Tier
    {
        public Hund(string name) : base(name)
        {
            anzahlBeine = 4;
        }
    }
```

Für Schlangen und Hunde kann die Funktion **schreibeBeinzahl()** aufgerufen werden, obwohl diese für beide Klassen nicht explizit codiert worden war.

```
class Program
    {
        static void Main(string[] args)
        {
            Console.OutputEncoding = Encoding.UTF8;

            Schlange s = new Schlange("Snaky");
            Hund h = new Hund("Wauwi");

            s.schreibeBeinzahl();
            h.schreibeBeinzahl();

            Console.ReadKey();
        }
    }
}
```

WBS TRAINING

Objektorientierte Programmierung – Polymorphie

- Das Konzept der **Polymorphie** (Vielgestaltigkeit) ist eng verbunden mit dem der Vererbung.
- Der Begriff spielt darauf an, dass der selbe Methodenname für **unterschiedliche Funktionalitäten** stehen kann, je nachdem, zu welchem Objekt er aufgerufen wird.

- Auch dieses Prinzip werden wir uns im Folgenden mit Hilfe eines C#-Codes veranschaulichen:
  - als Beispiel wollen wir die (abstrakte) Basisklasse **„Fahrzeug“** betrachten

    (der Begriff „**abstrakt**“ soll in diesem Zusammenhang deutlich machen, dass es keine Objekte geben soll, die ausschließlich vom Typ „Fahrzeug“ sind)

  - Von der Basisklasse Fahrzeug sollen die beiden abgeleiteten Klassen **„Auto“** und **„Zug“** erben

    (diese Klassen sind **nicht abstrakt** => es gibt also Objekte, die vom Typ „Auto“ oder vom Typ „Zug“ sind. Auf Grund der Vererbungsbeziehung ist jedes Auto und jeder Zug somit auch vom Typ „Fahrzeug“, aber kein Fahrzeug ist dies ausschließlich, sondern ist zugleich immer auch ein Auto oder Zug)

  - In der Klasse Fahrzeug ist die (abstrakte) Methode „**mach Geräusch**“ implementiert. Sie wird an die beiden abgeleiteten Klassen vererbt. Diese Methode soll aber bei Autos und Zügen eine **unterschiedliche Funktionalität** aufweisen

    (der Begriff „abstrakt“ soll in diesem Zusammenhang deutlich machen, dass diese Methode nur als **Prototyp** eingeführt wird, also keine Funktionalität besitzt)

Objektorientierte Programmierung – Polymorphie

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    abstract class Fahrzeug
    {
        abstract public void machGeräusch();
    }

    class Auto : Fahrzeug
    {
        override public void machGeräusch()
        {
            Console.WriteLine("Brumm Brumm");
        }
    }

    class Zug : Fahrzeug
    {
        override public void machGeräusch()
        {
            Console.WriteLine("Ratter Ratter");
        }
    }
```

WBS TRAINING

Objektorientierte Programmierung – Polymorphie

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    abstract class Fahrzeug
    {
        abstract public void machGeräusch();
    }

    class Auto : Fahrzeug
    {
        override public void machGeräusch()
        {
            Console.WriteLine("Brumm Brumm");
        }
    }

    class Zug : Fahrzeug
    {
        override public void machGeräusch()
        {
            Console.WriteLine("Ratter Ratter");
        }
    }
```

Die **(abstrakte) Klasse** Fahrzeug wird hier als Basisklasse fungieren.
Sie besitzt den Funktionsprototypen **machGeräusch()**.
Dieser Prototyp besitzt natürlich keine Funktionalität, kann aber vererbt werden.

Objektorientierte Programmierung – Polymorphie

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    abstract class Fahrzeug
    {
        abstract public void machGeräusch();
    }

    class Auto : Fahrzeug
    {
        override public void machGeräusch()
        {
            Console.WriteLine("Brumm Brumm");
        }
    }

    class Zug : Fahrzeug
    {
        override public void machGeräusch()
        {
            Console.WriteLine("Ratter Ratter");
        }
    }
```

Die Klassen Auto und Zug **erben** beide von der Klasse Fahrzeug.

WBS TRAINING

Objektorientierte Programmierung – Polymorphie

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    abstract class Fahrzeug
    {
        abstract public void machGeräusch();
    }

    class Auto : Fahrzeug
    {
        override public void machGeräusch()
        {
            Console.WriteLine("Brumm Brumm");
        }
    }

    class Zug : Fahrzeug
    {
        override public void machGeräusch()
        {
            Console.WriteLine("Ratter Ratter");
        }
    }
```

Für beide Klassen wird aber jeweils eine **individuelle Funktionalität** festgelegt.

WBS TRAINING

Objektorientierte Programmierung – Polymorphie

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    abstract class Fahrzeug
    {
        abstract public void machGeräusch();
    }

    class Auto : Fahrzeug
    {
        override public void machGeräusch()
        {
            Console.WriteLine("Brumm Brumm");
        }
    }

    class Zug : Fahrzeug
    {
        override public void machGeräusch()
        {
            Console.WriteLine("Ratter Ratter");
        }
    }
```

```
    class Program
    {
        static void Main(string[] args)
        {
            Console.OutputEncoding = Encoding.UTF8;

            Auto a = new Auto();
            Zug z = new Zug();

            Fahrzeug[] array = new Fahrzeug[2];
            array[0] = a;
            array[1] = z;

            for(int i=0;i<2;i++)
            {
                array[i].machGeräusch();
            }


            Console.ReadKey();
        }
    }
}
```

WBS TRAINING

Objektorientierte Programmierung – Polymorphie

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    abstract class Fahrzeug
    {
        abstract public void machG
    }

    class Auto : Fahrzeug
    {
        override public void machGeräusch()
        {
            Console.WriteLine("Brumm Brumm");
        }
    }

    class Zug : Fahrzeug
    {
        override public void machGeräusch()
        {
            Console.WriteLine("Ratter Ratter");
        }
    }
```

Zunächst wird ein **Objekt** vom Typ Auto und ein **Objekt** vom Typ Zug kreiert.

```
    class Program
    {
        static void Main(string[] args)
        {
            Console.OutputEncoding = Encoding.UTF8;

            Auto a = new Auto();
            Zug z = new Zug();

            Fahrzeug[] array = new Fahrzeug[2];
            array[0] = a;
            array[1] = z;

            for(int i=0;i<2;i++)
            {
                array[i].machGeräusch();
            }


            Console.ReadKey();
        }
    }
}
```

WBS TRAINING

Objektorientierte Programmierung – Polymorphie

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    abstract class Fahrzeug
    {
        abstract public void machGeräusch();
    }

    class Auto : Fahrzeu
    {
        override public vo
        {
            Console.WriteLine("Brumm Brumm");
        }
    }

    class Zug : Fahrzeug
    {
        override public void machGeräusch()
        {
            Console.WriteLine("Ratter Ratter");
        }
    }
```

Daraufhin wird ein **Array** vom Typ Fahrzeug eingeführt

```
    class Program
    {
        static void Main(string[] args)
        {
            Console.OutputEncoding = Encoding.UTF8;

            Auto a = new Auto();
            Zug z = new Zug();

            Fahrzeug[] array = new Fahrzeug[2];
            array[0] = a;
            array[1] = z;

            for(int i=0;i<2;i++)
            {
                array[i].machGeräusch();
            }


            Console.ReadKey();
        }
    }
}
```

Objektorientierte Programmierung – Polymorphie

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    abstract class Fahrzeug
    {
        abstract public void machGeräusch();
    }

    class Auto : Fahrzeug
    {
        override public void machGeräusch()
        {
            Console.W
        }
    }

    class Zug : Fahrzeug
    {
        override public void machGeräusch()
        {
            Console.WriteLine("Ratter Ratter");
        }
    }
```

In array[0] wird das Objekt **a** (vom Typ Auto) abgespeichert
In array[1] wird das Objekt **z** (vom Typ Zug) abgespeichert

```
class Program
{
    static void Main(string[] args)
    {
        Console.OutputEncoding = Encoding.UTF8;

        Auto a = new Auto();
        Zug z = new Zug();

        Fahrzeug[] array = new Fahrzeug[2];
        array[0] = a;
        array[1] = z;

        for(int i=0;i<2;i++)
        {
            array[i].machGeräusch();
        }


        Console.ReadKey();
    }
}
}
```

WBS TRAINING

Objektorientierte Programmierung – Polymorphie

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    abstract class Fahrzeug
    {
        abstract public void machGeräusch();
    }

    class Auto : Fahrzeug
    {
        override public void machGeräusch()
        {
```

Das in einer Schleife Feld für Feld abgearbeitete Array vom Typ Fahrzeug hat also ein Auto und einen Zug abgespeichert.

Die **Polymorphie** erlaubt nun, dass zwar stets die selbe Funktion machGeräusch() aufgerufen wird, diese aber nicht stets die selbe Ausgabe liefert, sondern dies jeweils passend zum Typ Auto oder Zug variiert.

```
        }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Console.OutputEncoding = Encoding.UTF8;

        Auto a = new Auto();
        Zug z = new Zug();

        Fahrzeug[] array = new Fahrzeug[2];
        array[0] = a;
        array[1] = z;

        for(int i=0;i<2;i++)
        {
            array[i].machGeräusch();
        }


        Console.ReadKey();
    }
}
}
```

Klassendiagramm – Definition + Motivation

- Der Aufbau eines objektorientierten Programmes kann recht anspruchsvoll sein, daher bietet es sich an, diesen **graphisch zu veranschaulichen**. Hierzu dienen Klassendiagramme, die üblicherweise aus der **UML** (Unified Modelling Language) stammen.
- Im Folgenden wollen wir uns die zu einem Klassendiagramm gehörenden Elemente und deren Aufbau an einfachen Beispielen verdeutlichen, hierzu zählen:
  - **Klasse**
    - Allgemeine Klasse
    - Basisklasse (eventuell abstrakt)
    - Abgeleitete Klasse
  - **Interface** (abgeschwächte Form einer abstrakten Klasse, besitzt nur Methoden-Prototypen)
  - **Assoziation(Relation) und Kardinalität**
    - Vererbung
    - Implementierung
    - Aggregation
    - Komposition

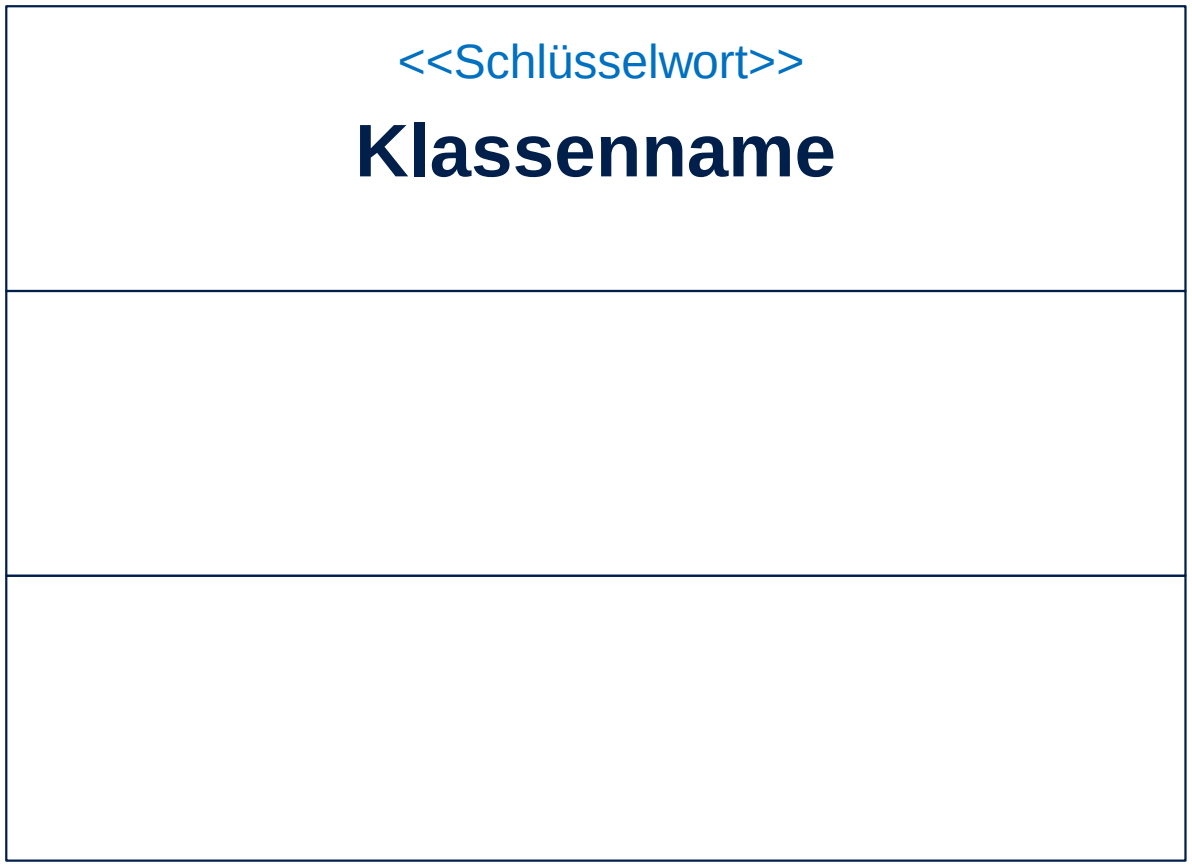
Klassendiagramm – Klasse (detaillierte Darstellung)

Klassen werden in
Rechtecken
dargestellt, die in 3 Abschnitte
unterteilt werden

Klassendiagramm – Klasse (detaillierte Darstellung)

Im oberen Abschnitt wird der **Name** der Klasse eingetragen

Klassendiagramm – Klasse (detaillierte Darstellung)

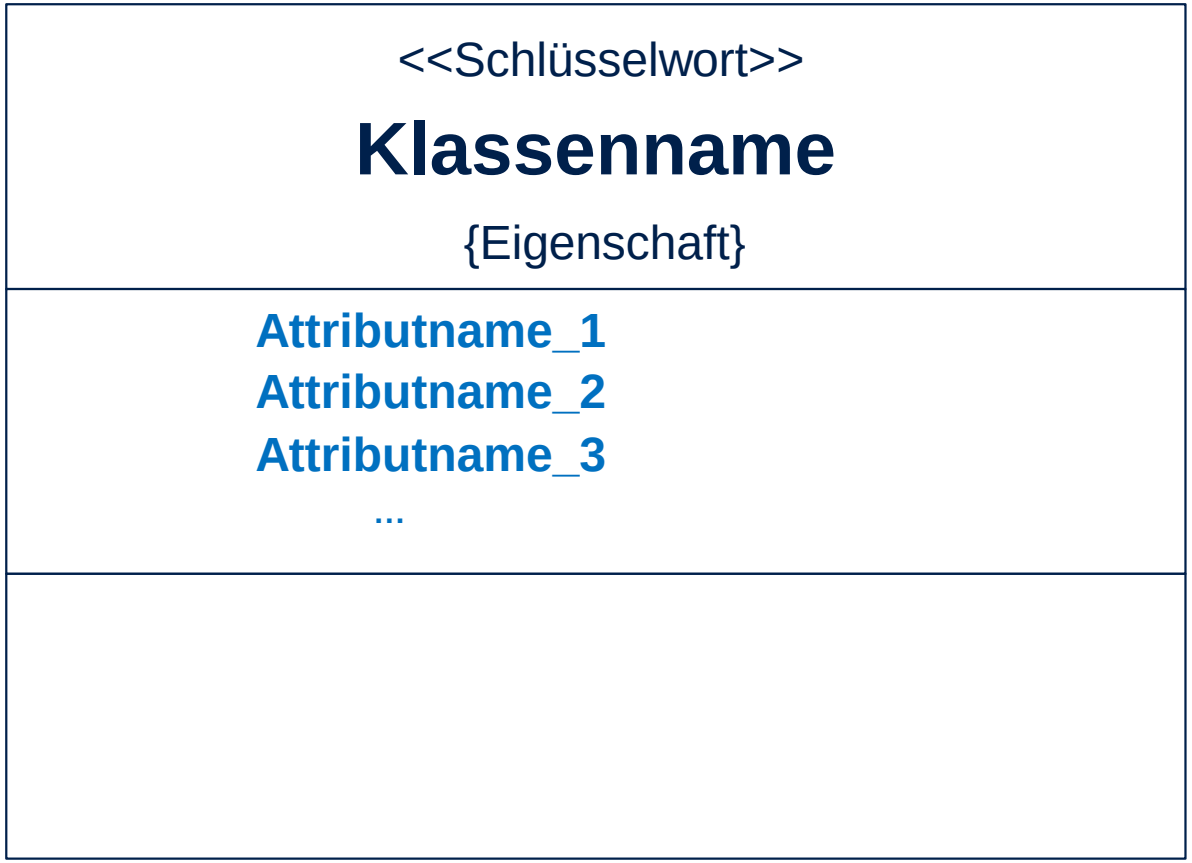


In doppelte spitzen Klammern kann ein **Schlüsselwort** eingetragen werden.
(In dieser elementaren Einführung werden wir dies nur bei Interfaces nutzen)

Klassendiagramm – Klasse (detaillierte Darstellung)

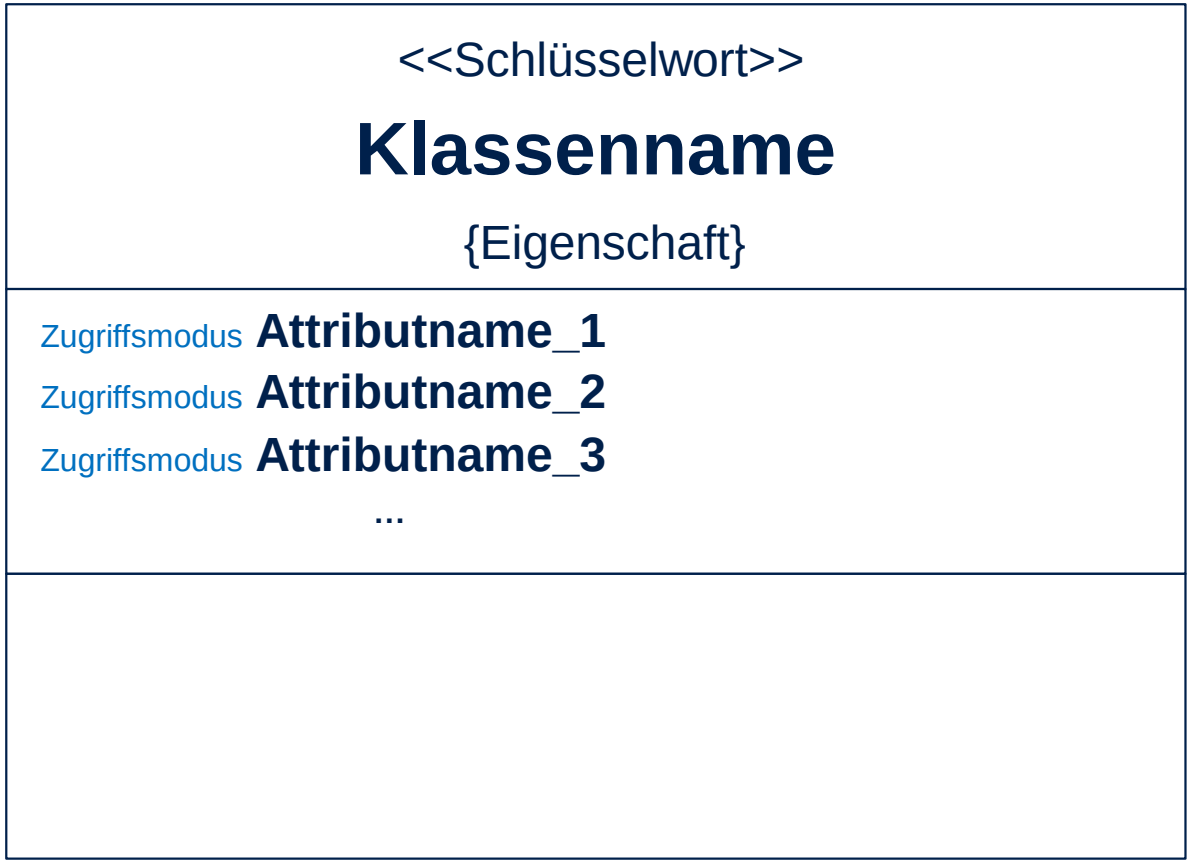
In geschwungenen Klammern kann eine **Eigenschaft** eingetragen werden.
(In dieser elementaren Einführung wird dies nur die Eigenschaft „abstract“ sein)

Klassendiagramm – Klasse (detaillierte Darstellung)



Im mittleren Abschnitt werden die Attributnamen eingetragen

Klassendiagramm – Klasse (detaillierte Darstellung)

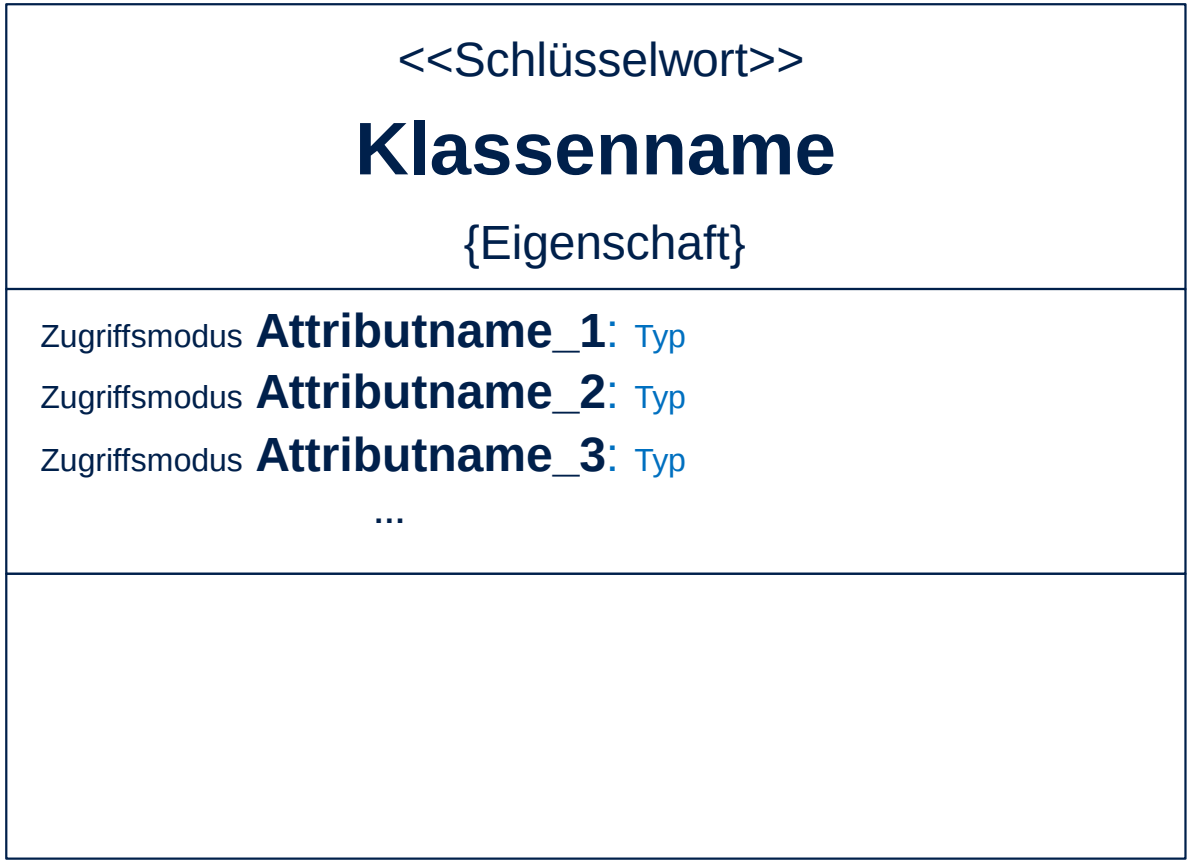


Die Attributnamen können durch den Zugriffsmodus ergänzt werden.

In dieser elementaren Einführung unterscheiden wir nur zwischen

public (+)
private (-)

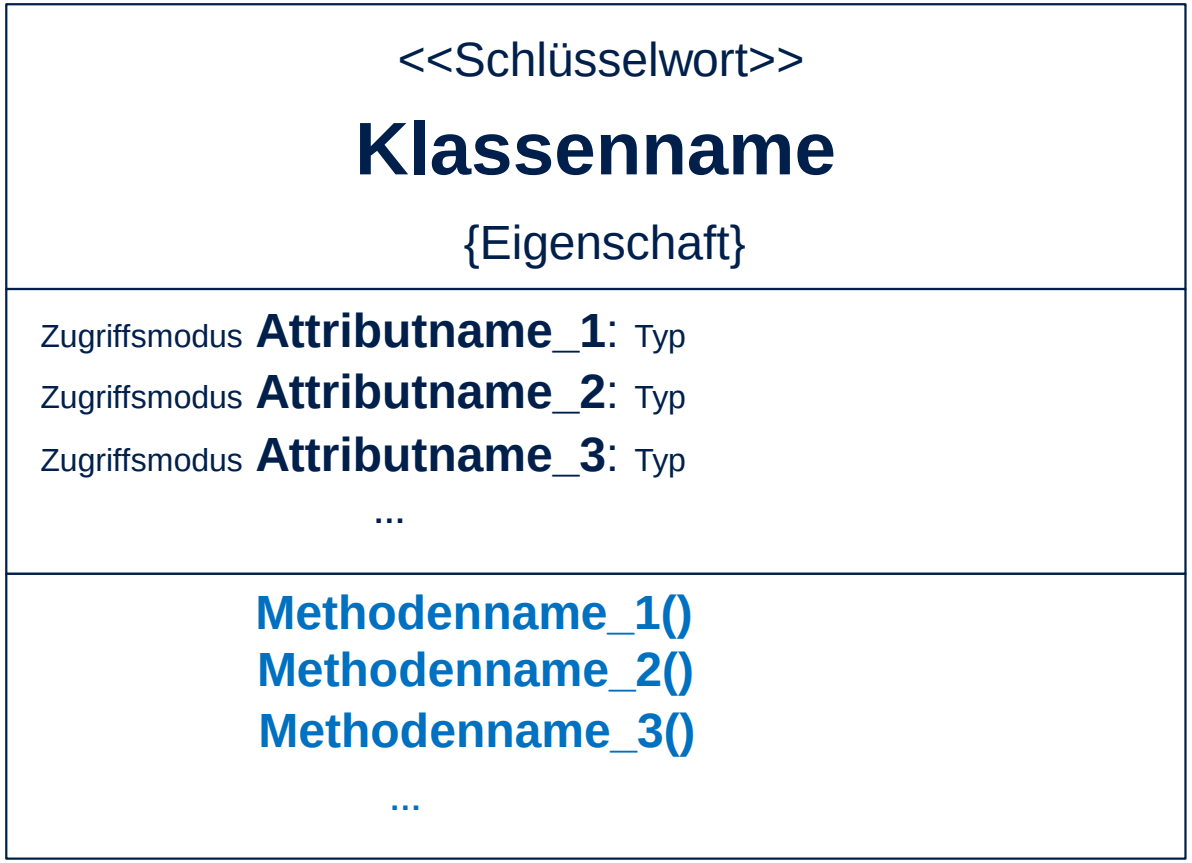
Klassendiagramm – Klasse (detaillierte Darstellung)



Ferner kann der **Typ** jedes Attributs angegeben werden, der durch eine Doppelpunkt getrennt hinter dem Namen des Attributs eingetragen wird.

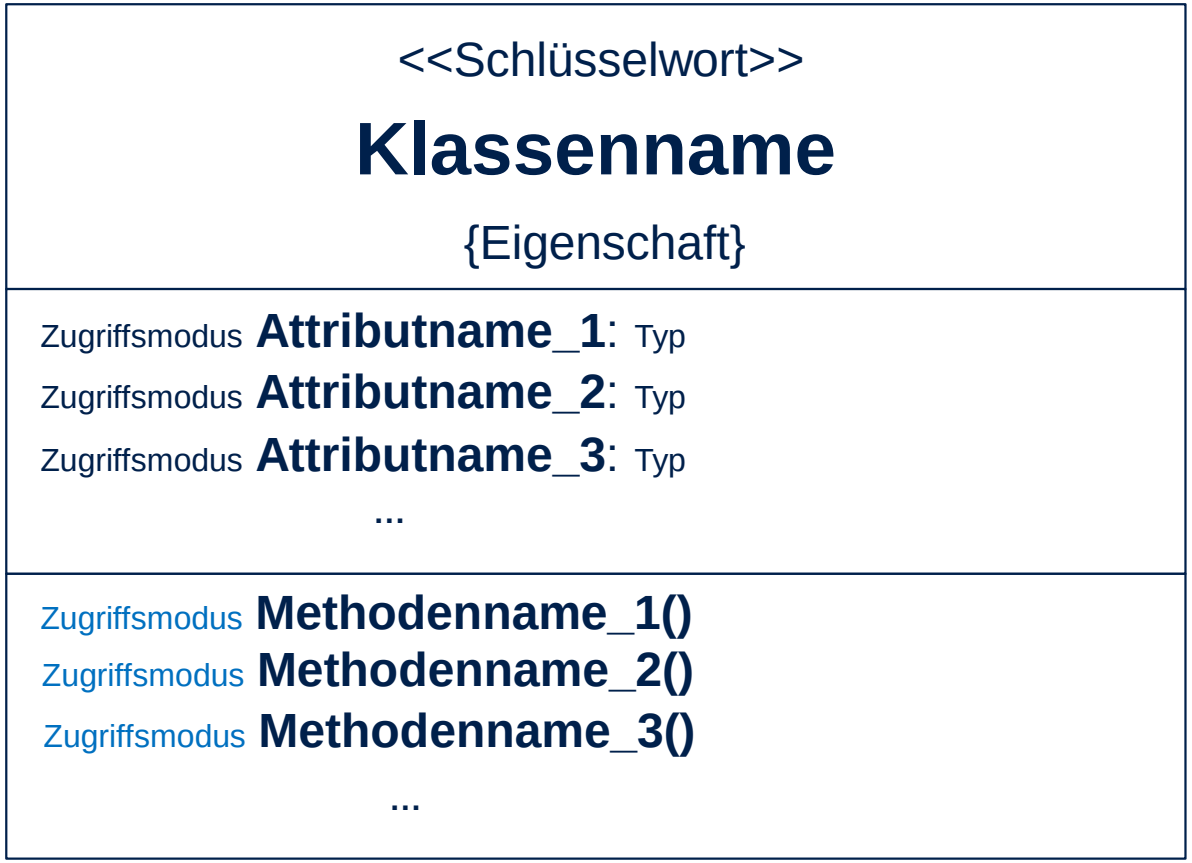
Neben den uns bekannten Typen wie char, int, float … könnte dies auch **boolean** sein

Klassendiagramm – Klasse (detaillierte Darstellung)



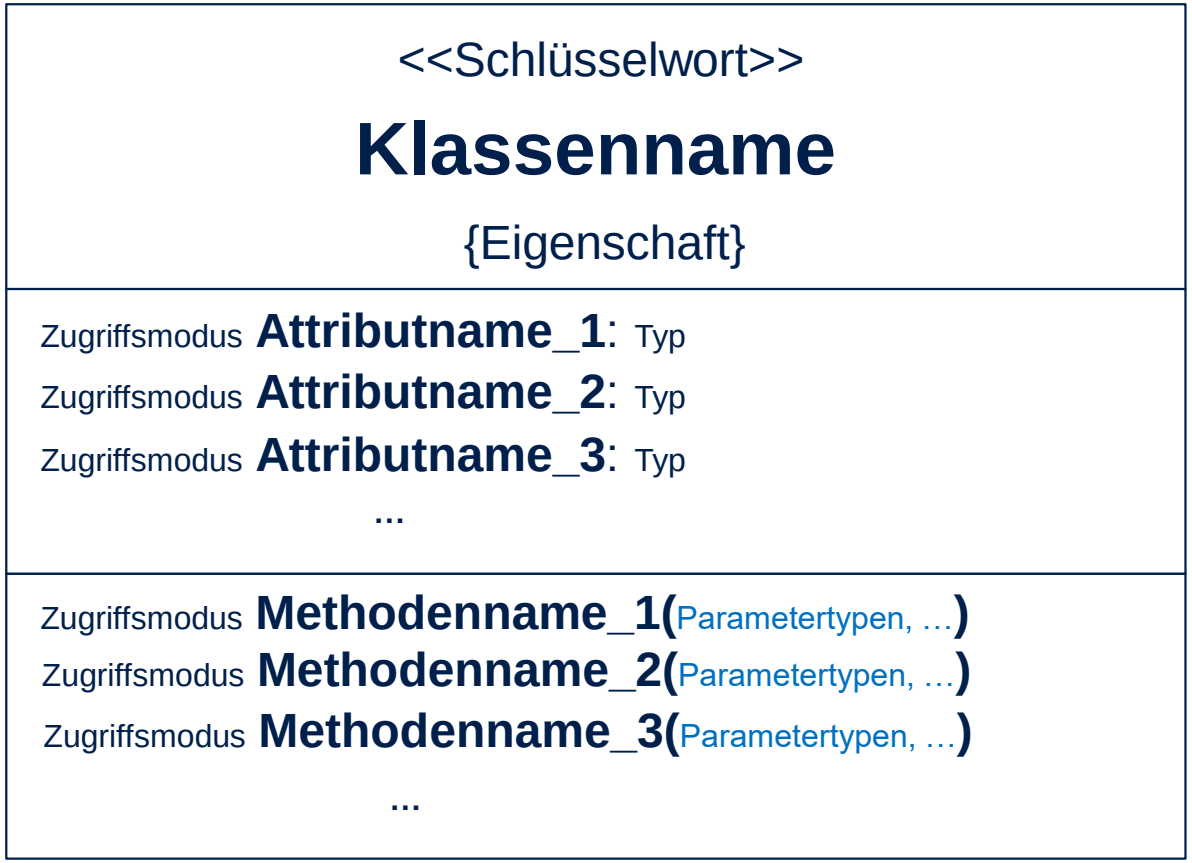
Im unteren Abschnitt werden die **Methodennamen** eingetragen

Klassendiagramm – Klasse (detaillierte Darstellung)



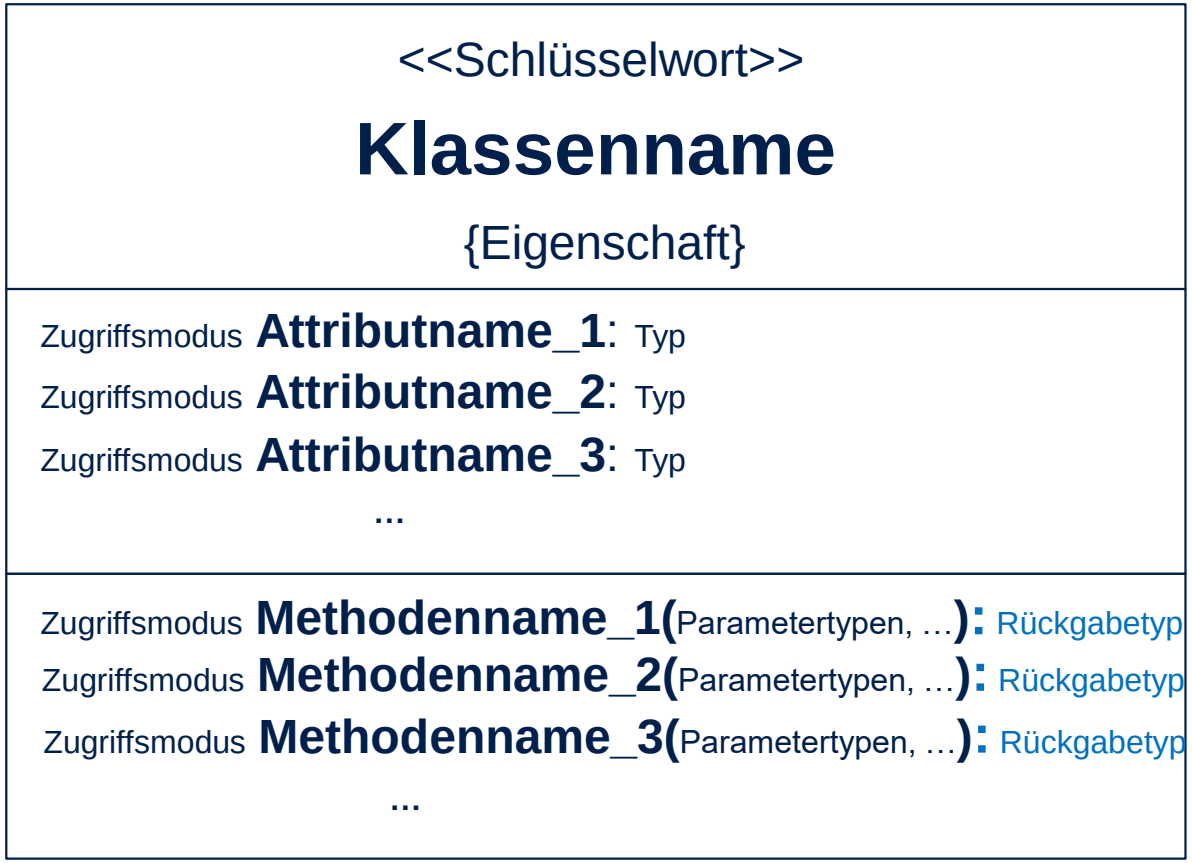
Auch bei Methoden kann der Zugriffsmodus ergänzt werden

Klassendiagramm – Klasse (detaillierte Darstellung)



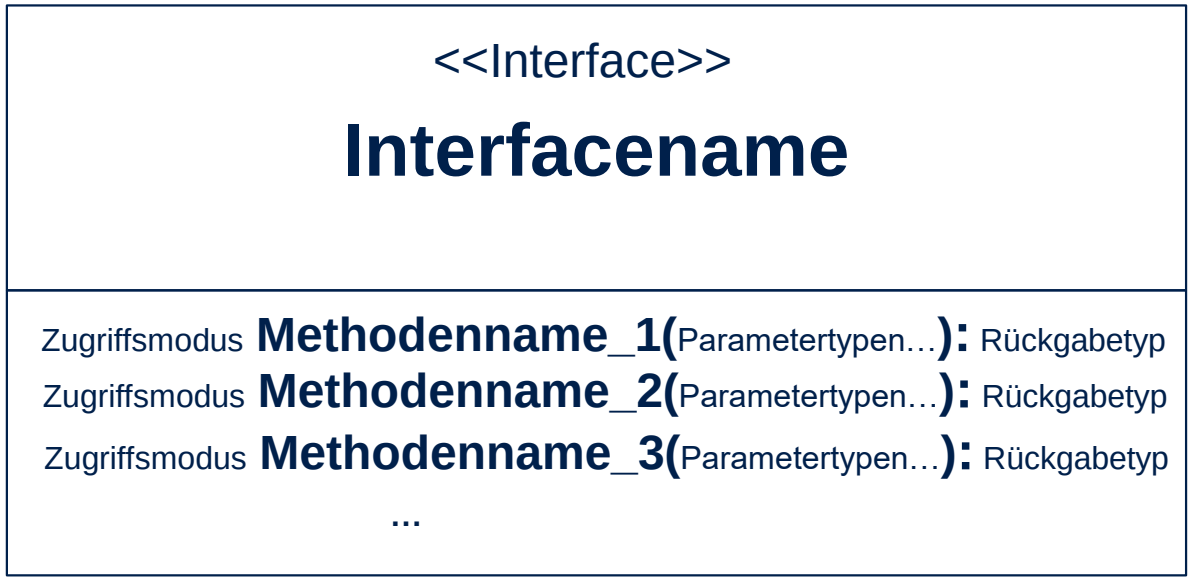
Bei Bedarf kann auch der **Typ** der einzelnen Parameters eingetragen werden

Klassendiagramm – Klasse (detaillierte Darstellung)



Ebenso kann natürlich auch der **Typ** des Rückgabewertes angegeben werden

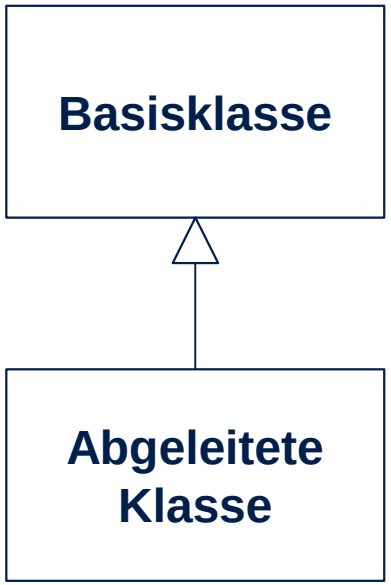
Klassendiagramm – Interface (detaillierte Darstellung)



Klassischerweise besitzen Interface keine Attribute sondern nur Methoden-Prototypen.

Seit UML 2.0 sind zwar auch Attribute zulässig, dies wird aber von C# nicht unterstützt.

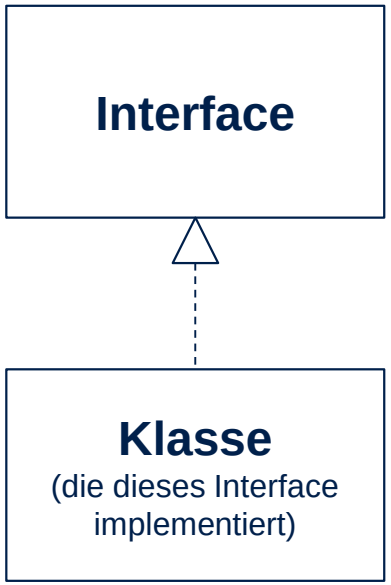
Klassendiagramm – Vererbung



Der **Generalisierungspfeil** der UML wird in Klassendiagrammen genutzt, um eine Vererbungs-Relation darzustellen.

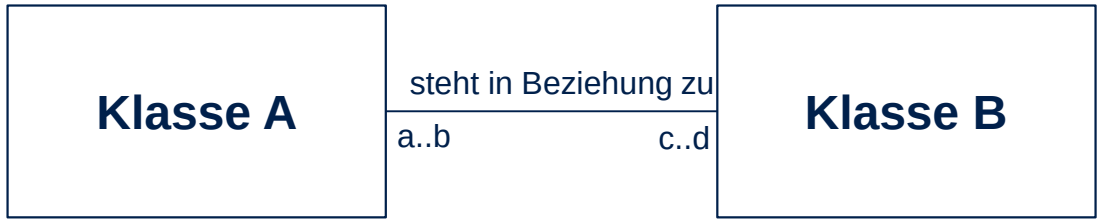
Der Pfeil startet bei der abgeleiteten Klasse, die Spitze zeigt auf die Basisklasse.

Klassendiagramm – Implementierung



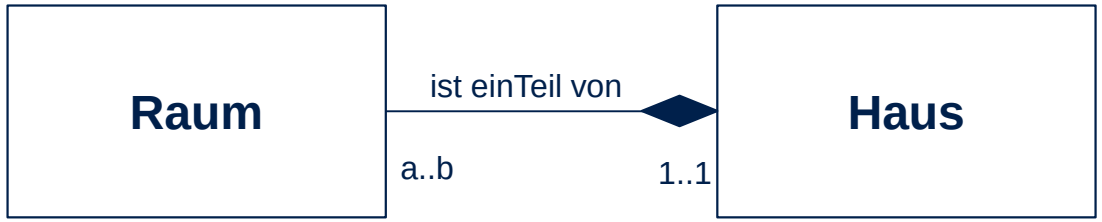
Ein Pfeil, der im Wesentlichen wie ein Generalisierungspfeil dargestellt wird, dessen **Linie** aber **gestrichelt** ist, zeigt an, welche Klasse welches Interface implementiert.

Klassendiagramm – Assoziation



(Allgemeine) Relationen zwischen Klassen werden durch **Assoziationen** dargestellt.
Wie bei Datenbankmodellen können diese durch Kardinalitäten spezifiziert werden.

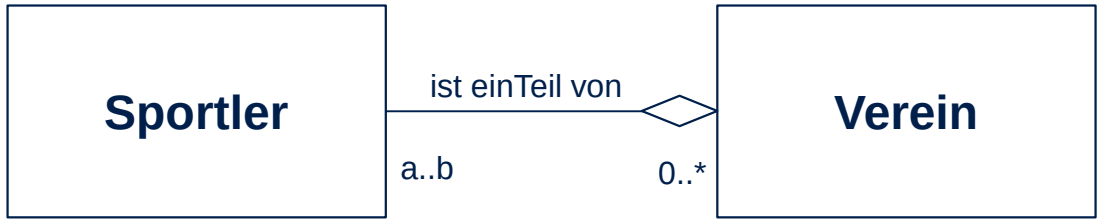
Klassendiagramm – Komposition



Bei der **Komposition** handelt es sich um eine spezielle Assoziation. Es ist eine „ist-Teil-von“-Beziehung, die sich dadurch auszeichnet, dass der **Teil nicht ohne das Ganze existieren** kann.

Kompositionen werden durch eine **gefüllte Raute** dargestellt. Diese befindet sich an der Klasse, die „das Ganze“ beschreibt. Deren Kardinalität ist bei Kompositionen notwendig eine 1 (bzw. 1..1).

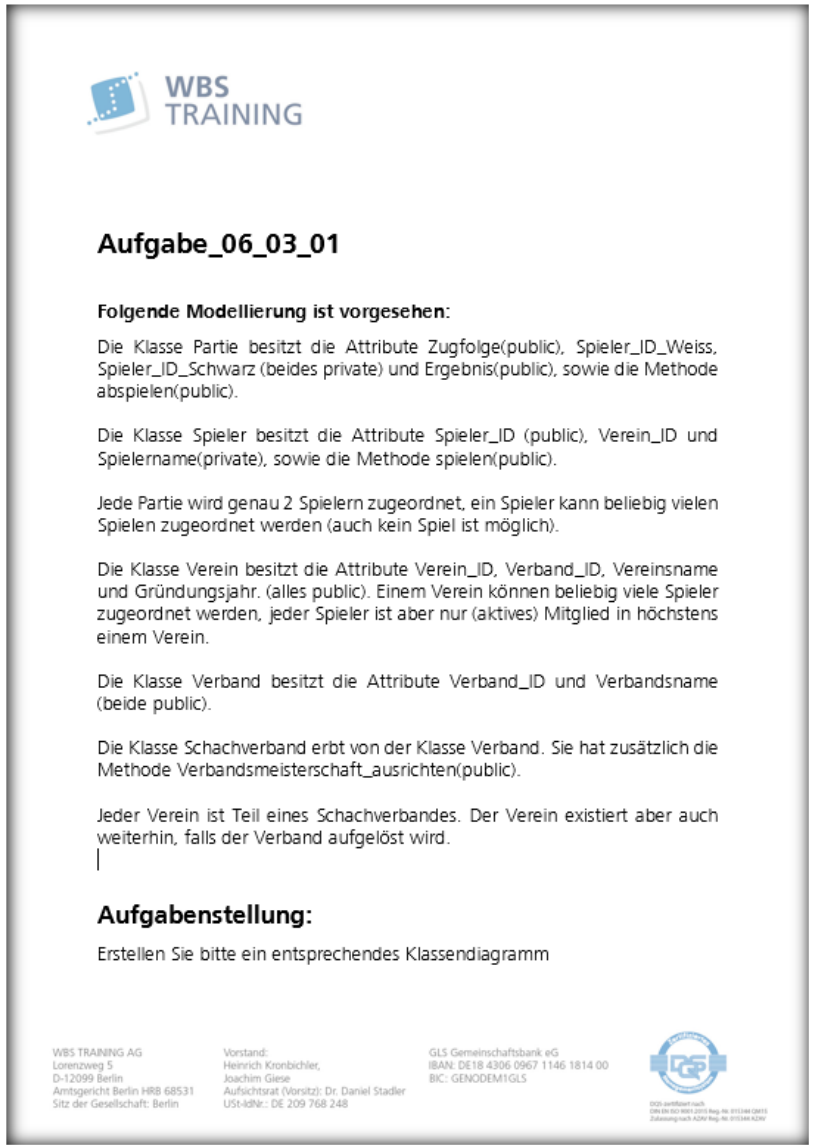
Klassendiagramm – Aggregation



Auch bei der **Aggregation** handelt es sich um eine „ist-Teil-von“-Beziehung, die sich allerdings dadurch auszeichnet, dass der **Teil auch ohne das Ganze existieren** kann.

Aggregationen werden durch eine **nicht-ausgefüllte Raute** dargestellt. Diese befindet sich an der Klasse, die „das Ganze“ beschreibt. Deren Kardinalität kann 0 oder mehr (also 0..*) sein.

Klassendiagramm – Übung A_06_03_01



WBS TRAINING

Aufgabe_06_03_01

Folgende Modellierung ist vorgesehen:

Die Klasse Partie besitzt die Attribute Zugfolge(public), Spieler_ID_Weiss, Spieler_ID_Schwarz (beides private) und Ergebnis(public), sowie die Methode abspielen(public).

Die Klasse Spieler besitzt die Attribute Spieler_ID (public), Verein_ID und Spielername(private), sowie die Methode spielen(public).

Jede Partie wird genau 2 Spielern zugeordnet, ein Spieler kann beliebig vielen Spielen zugeordnet werden (auch kein Spiel ist möglich).

Die Klasse Verein besitzt die Attribute Verein_ID, Verband_ID, Vereinsname und Gründungsjahr. (alles public). Einem Verein können beliebig viele Spieler zugeordnet werden, jeder Spieler ist aber nur (aktives) Mitglied in höchstens einem Verein.

Die Klasse Verband besitzt die Attribute Verband_ID und Verbandsname (beide public).

Die Klasse Schachverband erbt von der Klasse Verband. Sie hat zusätzlich die Methode Verbandsmeisterschaft_ausrichten(public).

Jeder Verein ist Teil eines Schachverbandes. Der Verein existiert aber auch weiterhin, falls der Verband aufgelöst wird.

Aufgabenstellung:

Erstellen Sie bitte ein entsprechendes Klassendiagramm

WBS TRAINING AG
Lorenzweg 5
D-12099 Berlin
Amtsgericht Berlin HRB 68531
Sitz der Gesellschaft: Berlin

Vorstand:
Heinrich Kronbichler,
Joachim Giese
Aufsichtsrat (Vorsitz): Dr. Daniel Stadler
USt-IdNr.: DE 209 768 248

GLS Gemeinschaftsbank eG
IBAN: DE18 4306 0967 1146 1814 00
BIC: GENODEM1GLS

WBS TRAINING

VIELEN DANK FÜR IHRE AUFMERKSAMKEIT!