# ZUSAMMENFASSUNG Objektorientierte Programmierung

## WAS BEDEUTET OBJEKTORIENTIERUNG?

- Die Objektorientierte Programmierung (OOP) ist ein Programmierparadigma (Programmierstil)
- Die Objektorientierung ist dem menschlichen Denken sehr ähnlich
- Alles in der OOP wird durch „Objekte“ beschrieben

## OBJEKTE

- Jedes Objekt wird durch Eigenschaften beschrieben
  **Beispiel**: Farbe, Größe, etc.
- Objekte können Methoden aufrufen
  **Beispiel**: Haus -> Verkaufen(), Hund -> Bellen()

## KLASSEN

- Klassen sind Baupläne für Objekte
- Sie geben die Eigenschaften und Methoden vor, die ein Objekt hat
- Sie beschreiben sozusagen die Art von Objekt, bzw. den Objekttyp
  **Beispiel**: Klasse Haus mit den Eigenschaften Farbe, Baujahr, Besitzer, etc. und den Methoden Verkaufen(), Bauen(), Renovieren()

## VERERBUNG

- In der OOP ist es auch möglich, Klassen zu vererben
- Bei der Vererbung übernimmt eine Klasse alle Eigenschaften und Methoden einer anderen Klasse
- Man kann dadurch eine Klasse erweitern, ohne diese selbst zu verändern
  **Beispiel**: Die Klasse Hund erbt Elemente der Klasse Säugetier

## WIE DEFINIERT MAN EINE KLASSE?

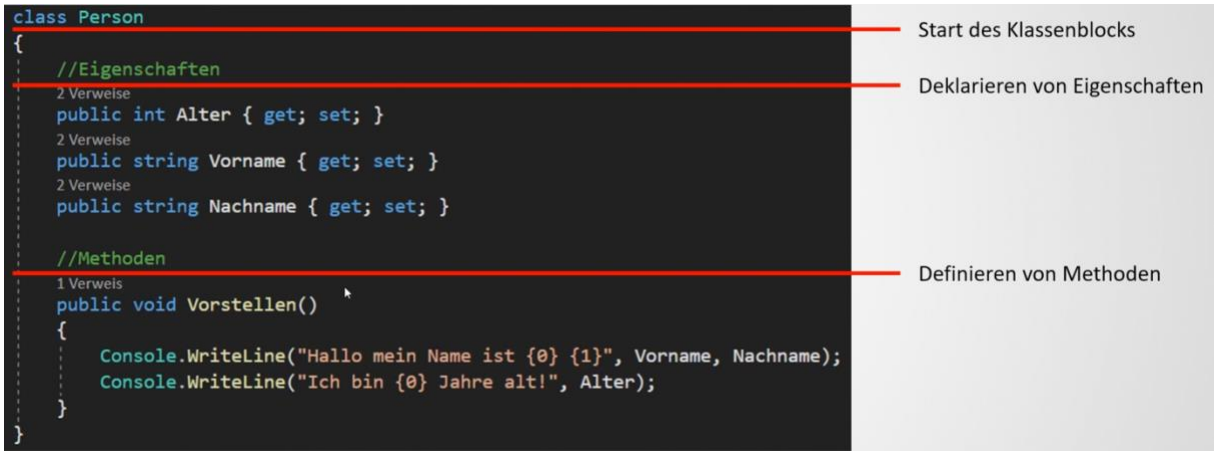


**Abb. 1 Klassendefinition**

## INSTANZIIEREN VON OBJEKTEN

```
static void Main()
{
    Person jens = new Person();
    jens.Alter = 31;
    jens.Vorname = "Jens";
    jens.Nachname = "Berg";

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

**Abb. 2, Instanziieren eines Objekts**

- Ein Objekt wird mit dem **„new“** Schlüsselwort erstellt
- Mit dem Punkt-Operator (.) kann man auf Eigenschaften und Methoden des Objekts zugreifen

## KLASSEN DEFINIEREN UND INSTANZIIEREN IN DER PRAXIS

In unserem Beispiel programmieren wir eine Klasse, die ein simples Bankkonto darstellt. Dieses Bankkonto soll zwei Eigenschaften haben: Eine Eigenschaft für den Kontostand und einen String namens Besitzer. Außerdem werden wir die Methoden Einzahlen() und Auszahlen() dafür definieren. Dafür erstellen wir uns zunächst ein neues Code Dokument.

### Info:

Um ein neues Code Dokument zu erstellen, klicken wir rechts (oder links) in unserer Projektmappe mit einem Rechtsklick auf unser Projekt, wählen im Kontextmenü den Menüpunkt „Hinzufügen“ aus, dann klicken wir auf „Klasse“, lassen Klasse ausgewählt, geben unten den Namen „Bankkonto“ ein und klicken auf „Hinzufügen“.

```
public class Bankkonto
{
    //Eigenschaften

    public decimal Kontostand { get; set; }

    public string Besitzer { get; set; }

    //Methoden

    public void Einzahlen(decimal betrag)
    {
        Kontostand += betrag;
        Console.WriteLine("Es wurden " + betrag + " Euro auf das Konto eingezahlt!");
        Console.WriteLine("Der neue Kontostand beträgt " + Kontostand + " Euro!");
    }


    public void Auszahlen(decimal betrag)
    {
        Kontostand -= betrag;
        Console.WriteLine("Es wurden " + betrag + " Euro vom Konto abgehoben!");
        Console.WriteLine("Der neue Kontostand beträgt " + Kontostand + " Euro!");
    }
}
```

Abb. 3, Definition der Klasse Bankkonto

```
class Program
{
    static void Main(string[] args)
    {
        Bankkonto konto = new Bankkonto();
        konto.Kontostand = 2500;
        konto.Besitzer = "Sandra Müller";

        konto.Einzahlen(250);

        Console.ReadKey();
    }
}
```

Abb. 4, Erstellen einer Variable, die ein Bankkonto Objekt enthält

```
Es wurden 250 Euro auf das Konto eingezahlt!
Der neue Kontostand beträgt 2750 Euro!
```

Abb. 5, Ausgabe

```
class Program
{
    static void Main(string[] args)
    {
        Bankkonto konto = new Bankkonto();
        konto.Kontostand = 2500;
        konto.Besitzer = "Sandra Müller";

        konto.Einzahlen(250);

        Console.WriteLine("Kontostand: " + konto.Kontostand);
        Console.ReadKey();
    }
}
```

Abb. 6, Zusätzliche Ausgabe des Kontostandes

```
Es wurden 250 Euro auf das Konto eingezahlt!
Der neue Kontostand beträgt 2750 Euro!
Kontostand: 2750
```

Abb. 7, Ausgabe

Es besteht auch die Möglichkeit, mehrere Objekte einer Klasse zu erzeugen. Auf unser Beispiel bezogen, könnten wir nun noch weitere Bankkonten hinzufügen:

```
class Program
{
    static void Main(string[] args)
    {
        Bankkonto konto = new Bankkonto();
        Bankkonto konto2 = new Bankkonto();
        konto.Kontostand = 2500;
        konto2.Kontostand = 3000;

        Console.WriteLine("Kontostand 1: " + konto.Kontostand);
        Console.WriteLine("Kontostand 2: " + konto2.Kontostand);

        Console.ReadKey();
    }
}
```

Abb. 8, Hinzufügen eines zweiten Objekts


**Abb. 9, Ausgabe**

### Info:

Eigenschaften und Klassennamen werden in C# groß, Variablennamen hingegen klein geschrieben. Das ist zwar keine Pflicht, aber eine Konvention, die auch von Microsoft so umgesetzt wird.

## WAS SIND EIGENSCHAFTEN?

- Eigenschaften wirken auf den ersten Blick wie normale Variablen einer Klasse. Das sind sie allerdings nicht!
- Eigenschaften sind besondere Variablen, welche mithilfe von zwei Methoden beschrieben und gelesen werden (get-, set-Methoden)
- Eigenschaften verwendet man zum Kapseln von Daten

## WANN VERWENDEN WIR EIGENSCHAFTEN?

- Wenn deren Werte das Objekt wirklich beschreiben und öfter gebraucht werden als nur in einer einzigen Methode (z. B. das Alter eines Menschen, der Name eines Menschen, …)
- Wenn wir in einer Methode Variablen benötigen, die mit dem Objekt an sich nichts zu tun haben, verwenden wir normale Variablen

```
class Person
{
    //Eigenschaften

    public int Alter { get; set; }

    public string Vorname { get; set; }

    public string Nachname { get; set; }


    //Methoden

    public void SageSatz(string satz)
    {
        string ausgabe = Vorname + " sagt: " + satz;
        Console.WriteLine(ausgabe);
    }
}
```

**Abb. 10, Eigenschaften der Klasse Person**

- **Alter**, **Vorname** und **Nachname** sind Eigenschaften, da sie das Objekt beschreiben.
- **ausgabe** ist eine temporäre Variable, da sie nur in diesem einen Methoden-Kontext benötigt wird

## WAS BEDEUTET DATENKAPSELUNG?

- Mithilfe der Datenkapselung werden Daten eines Objekts vor äußeren Zugriffen geschützt. Ermöglicht wird dies mit Eigenschaften
- Eine Eigenschaft ist ein Konstrukt aus einer privaten Variable und zwei Methoden, die dazu verwendet werden, um die Variable zu überschreiben und zu lesen
- Diese beiden Methoden nennt man „Get“ und „Set“, bzw. „Getter“ und „Setter“

## Get und Set

```
class Person
{
    //Eigenschaften
    public string Vorname { get; set; } //Automatisch implementierte Eigenschaft

    public string Nachname { get; set; }

    private int alter; //Private Variable für das Alter

    public int Alter //Nach außen sichtbare Eigenschaft "Alter"
    {
        get
        {
            return alter; //Get wird aufgerufen, wenn man die Eigenschaft lesen will
        }
        set
        {
            alter = value; //Set wird aufgerufen, wenn man die Eigenschaft überschreiben will
        }
    }
}
```

**Abb. 11, automatisch implementierte und ausgeschriebene Eigenschaften**

## EIGENSCHAFTEN UND DATENKAPSELUNG IN DER PRAXIS

Im nachfolgenden Beispiel werden wir zwei Eigenschaften definieren. Eine automatisch implementierte Eigenschaft und eine voll ausgeschriebene Eigenschaft mit ausgeschriebenen Get- und Set-Methoden.

```
class Program
{
    static void Main(string[] args)
    {
        Auto auto = new Auto();
        Console.WriteLine(auto.Hersteller);
        auto.Hersteller = "SuperCar";
    }
}

class Auto
{
    //Eigenschaften

    public string Hersteller { get; set; }
}
```

**Abb. 12, automatisch implementierte Eigenschaft**

```
class Auto
{
    //Eigenschaften

    public string Hersteller { get; set; }

    private int anzahlTüren;

    public int AnzahlTüren
    {
        get
        {
            return anzahlTüren;
        }
        set
        {
            anzahlTüren = value;
        }
    }
}
```

Abb. 13, ausgeschriebene Eigenschaft

## Info:

Wenn wir keinen Setter in unserer Klasse definieren, ist die Eigenschaft darin schreibgeschützt. Darüber hinaus besteht die Möglichkeit, einen **private Setter** (private set;) in einer Klasse zu definieren. Das bedeutet, dass man zwar auch schreibend darauf zugreifen kann, allerdings nur innerhalb der Klasse.

## WAS SIND ZUGRIFFSEBENEN?

- Mit Zugriffsebenen geben wir an, von welchen Stellen im Code auf bereits definierte Elemente (Klassen, Eigenschaften, Methoden, …) zugegriffen werden kann
- Wir verwenden Zugriffsebenen, um den Code überschaubarer und sicherer zu machen

**In C# gibt es folgende Zugriffsebenen:**

| Zugriffsebene | Reichweite |
|---|---|
| Public | Keine Zugriffseinschränkung |
| Private | Zugriff auf die Klasse beschränkt |
| Protected | Zugriff auf Klasse und davon abgeleitete Typen beschränkt |
| Internal | Zugriff auf die Assembly beschränkt |
| Protected Internal | Zugriff auf die aktuelle Assembly oder die von der enthaltenen Klasse abgeleiteten Typen beschränkt |

Abb. 14, Zugriffsebenen in C#

## WERTE- UND REFERENZTYPEN

- Wertetypen sind die primitiven Datentypen (integer, float, …), Structs und Enums
- Referenztypen sind Strings, Arrays und Klassen
- Der Unterschied zwischen Werte- und Referenztypen besteht in der Art und Weise wie die Werte gespeichert werden
- Wertetypen werden auf dem Stack gespeichert
- Referenztypen werden auf dem Heap gespeichert

## STACK UND HEAP

- Der **Stack** ist ein Speicherbereich des RAM (Arbeitsspeicher), welcher direkt vom Prozessor mit einem Stackpointer unterstützt wird
- Die Speicher-Allokation ist effizient und schnell
- Der **Heap** ist ein Speicherbereich des RAM, welcher nicht so gut strukturiert ist wie der Stack. Speicher im Heap muss wieder freigegeben werden!
- Die Speicher-Allokation ist weniger effizient -> Referenztypen belasten den Speicher mehr, da der Speicher vom **Garbage Collector** (räumt automatisiert den Speicher auf) wieder freigegeben werden muss

## UNTERSCHIEDE IM CODE

- Variablen von Wertetypen enthalten immer den tatsächlichen Wert, den wir in sie geschrieben haben
- Variablen von Referenztypen enthalten nur einen Zeiger/Pointer mit der Referenz auf das Objekt (Sie enthalten die Adresse zum Objekt im Speicher)

**Beispiel:**

```
static void Main()
{
    int zahl1 = 10;
    int zahl2 = zahl1; //Kopie von Wert zahl1

    Person person1 = new Person();
    Person person2 = person1; //Exakt dasselbe Objekt wie person1
}
```

**Abb. 15, Beispiel**

**Praxisbeispiel:**

```
static void Main()
{
    Person person1 = new Person();
    Person person2 = person1;

    person1.Vorname = "Jens";
    person1.Nachname = "Berg";

    Console.WriteLine(person1.Vorname + " " + person1.Nachname);
    Console.WriteLine(person2.Vorname + " " + person2.Nachname);
    Console.ReadKey();


}
```

Abb. 16, Praxisbeispiel Referenztypen

Die Ausgabe zeigt, dass beide Variablen (person1 und person2) dieselbe Referenz zum selben Objekt im Speicher enthalten. Bei den Wertetypen ist dies nicht der Fall, da wir dort mit Kopien arbeiten.

```
Jens Berg
Jens Berg
```

Abb. 17, Ausgabe

## DER KONSTRUKTOR

- Der Konstruktor ist die erste Methode, die in einem Objekt aufgerufen wird
- Er wird dazu verwendet, Eigenschaften und Variablen zu initialisieren
- Der Konstruktor-Aufruf wird mit dem „new“ Schlüsselwort eingeleitet

## DER KONSTRUKTOR-AUFRUF

- Der Konstruktor wird bei der Erstellung eines Objekts aufgerufen
- Ihm können auch Parameter übergeben werden

```
Person person1 = new Person();
```

Abb. 18, parameterloser Konstruktor-Aufruf

```
Person person1 = new Person("Sabine", "Müller", 34);
```

Abb. 19, Konstruktor-Aufruf mit Parametern

## DIE KONSTRUKTOR-DEFINITION

- Ein Konstruktor muss nicht selbst definiert werden, wenn man diesen nicht braucht. Es wird ein leerer Konstruktor automatisch erzeugt, sollte kein eigener definiert werden
- Der Konstruktor hat den gleichen Namen wie die Klasse

```
class Person
{
    //Eigenschaften

    public string Vorname { get; set; }

    public string Nachname { get; set; }

    public int Alter { get; set; }


    //Konstruktor

    public Person(string vorname, string nachname, int alter)
    {
        Vorname = vorname;
        Nachname = nachname;
        Alter = alter;
    }
}
```

Abb. 20, Definition eines eigenen Konstruktors

```
static void Main()
{
    Person person1 = new Person("Sabine", "Müller", 34);
}
```

Abb. 21, Aufruf des Konstruktors

## DER KONSTRUKTOR IN DER PRAXIS

```
class Hund
{
    //Eigenschaften

    public int Alter { get; set; }

    public string Name { get; set; }

    public string Herrchen { get; set; }


    //Konstruktor

    public Hund(int alter, string name, string herrchen)
    {
        Alter = alter;
        Name = name;
        Herrchen = herrchen;
    }

}
```

**Abb. 22, Erstellen der Klasse „Hund“ mit Eigenschaften und Konstruktor**

```
class Program
{
    static void Main(string[] args)
    {
        Hund hund1 = new Hund(4, "Pluto", "Peter Müller");
        Console.WriteLine(hund1.Herrchen);
        Console.ReadKey();
    }
}
```

**Abb. 23, Erstellung eines Objekts der Klasse Hund innerhalb der Main Methode**

```
Peter Müller
```

**Abb. 24, Ausgabe**

Info:

Mit dem Schlüsselwort "this" verweist man auf die aktuelle Instanz der Klasse.

```
private int alter;

//Konstruktor

public Hund(int alter, string name, string herrchen)
{
    this.alter = alter;
}
```

Abb. 25, Das Schlüsselwort „this“

## WAS IST EINE STATISCHE KLASSE?

- Statische (static) Klassen sind Klassen, aus welchen man keine Objekte instanziieren kann
- Der Zugriff auf Member erfolgt nicht über Objekte, sondern über den Klassennamen

```
static void Main()
{
    double number = Mathematik.Addition(10, 3);
}
```

Abb. 26, Beispiel einer statischen Klasse

## WANN VERWENDEN WIR STATISCHE KLASSEN?

- Wenn wir keine unterschiedlichen Objekte der Klasse benötigen
  Beispiele: Console Klasse, Math Klasse
- Wenn die Funktionalität der Klasse nicht an die Eigenschaften von Objekten gebunden ist

```
static class Mathematik
{
    public static double Addition(double num1, double num2)
    {
        return num1 + num2;
    }

    public static double Subtraktion(double num1, double num2)...

    public static double Multiplikation(double num1, double num2)...

    public static double Division(double num1, double num2)...

    public static double Modulo(double num1, double num2)...
}
```

Abb. 27, Definition einer statischen Klasse

```
static void Main()
{
    double number = Mathematik.Addition(10, 3);
}
```

Abb. 28, Zugriff auf die statische Klasse

## STATISCHE KLASSEN UND METHODEN IN DER PRAXIS

In unserem Praxisbeispiel schreiben wir eine Methode, die die Fläche eines Rechtecks berechnen soll. Methoden innerhalb statischer Klassen müssen ebenfalls statisch sein.

```
class Program
{
    static void Main(string[] args)
    {
        double fläche = Mathematik.RechteckFläche(10, 15);
        Console.WriteLine(fläche);
        Console.ReadKey();
    }
}

static class Mathematik
{
    public static double RechteckFläche(double breite, double höhe)
    {
        return breite * höhe;
    }
}
```

Abb. 29, Definition einer statischen Klasse und Methode zur Berechnung eines Rechtecks

```
150
```

Abb. 30, Ausgabe

**Beispiel einer Math Klasse:**

```
static void Main(string[] args)
{
    double squareRoot = Math.Sqrt(49);
    Console.WriteLine(squareRoot);
    Console.ReadKey();
}
```

Abb. 31, Math Klasse

```
7
```

Abb. 32, Ausgabe

## WAS IST EIN NAMESPACE?

- Ein Namespace ist ein festgelegter Bereich, der verknüpfte bzw. logisch zusammenhängende Klassen enthält
- Mit dem „using“ Schlüsselwort können wir Namespaces in unser Code-Dokument einbinden

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

Abb. 33, using-Direktive

## ORDNERSTRUKTUREN DURCH NAMESPACES

- Mit Namespaces können wir unseren Code in Ordnern organisieren

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

namespace BeispielProgramm.Geometry
{
    class Rectangle
    {
    }
}
```

Abb. 34, Erstellung der Klasse Rectangle

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


namespace BeispielProgramm
{
    class Program
    {
        static void Main(string[] args)
        {
            Rectangle rect = new Rectangle();
        }
    }
}
```

Abb. 35, Erzeugung eines Objekts der Klasse Rectangle

## WAS IST VERERBUNG?

- Beim Vererben geben wir Member (Methoden, Eigenschaften, …) einer Klasse an eine andere Klasse weiter
- Somit können wir Klassen erstellen, die auf der Funktionalität von anderen Klassen aufbauen

**Beispiel:**

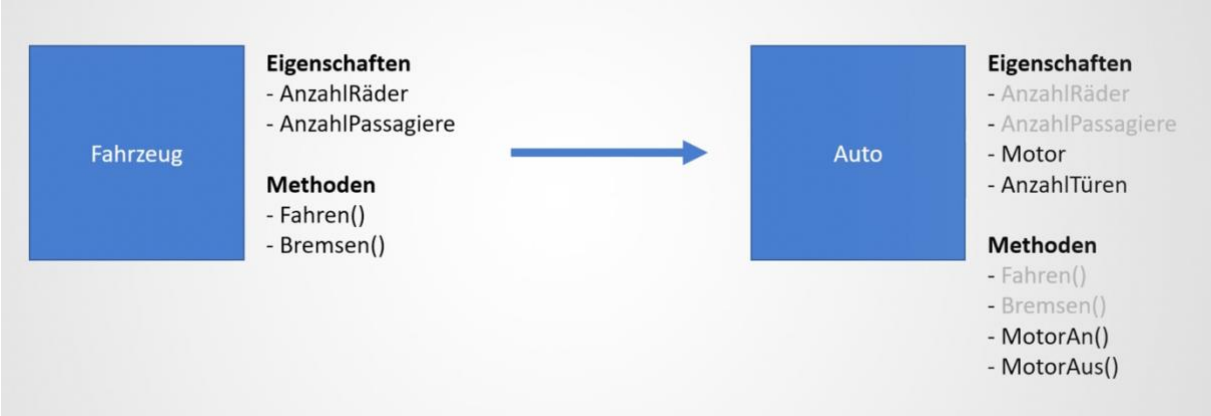


Abb. 36, Die Klasse Fahrzeug vererbt an die Klasse Auto

**Beispiel im Code:**

```
class Fahrzeug
{
    //Eigenschaften

    public int AnzahlRäder { get; set; }

    public int AnzahlPassagiere { get; set; }


    //Methoden

    public void Fahren()...

    public void Bremsen()...
}
```

Abb. 37, Erstellen der Klasse Fahrzeug mit Eigenschaften und Methoden

```
class Auto : Fahrzeug
{
    //Eigenschaften

    public string Motor { get; set; }

    public int AnzahlTüren { get; set; }


    //Methoden

    public void MotorAn()...

    public void MotorAus()...
}
```

Abb. 38, Erstellen der Klasse Auto, die von der Klasse Fahrzeug erbt

**Ein Auto Objekt erstellen:**

```
static void Main(string[] args)
{
    Auto meinAuto = new Auto();

    meinAuto.AnzahlPassagiere = 5;
    meinAuto.AnzahlTüren = 4;
    meinAuto.Motor = "Super Turbo Motor";

    meinAuto.MotorAn();
    meinAuto.Fahren();
    meinAuto.Bremsen();
    meinAuto.MotorAus();
}
```

Abb. 39, Erstellen eines Auto Objekts

## Info:

Die Klasse „Auto“ enthält alle „Eigenschaften“ der Klasse Fahrzeug.

## VERERBUNG IN DER PRAXIS

In unserem Praxisbeispiel werden wir eine Klasse „Hund“ mit verschiedenen Eigenschaften und Methoden erstellen.

```
class Hund
{
    //Eigenschaften

    public string Name { get; set; }

    public int Alter { get; set; }

    public string Rasse { get; set; }


    //Methoden

    public void Fressen()
    {
        Console.WriteLine("Der Hund " + Name + " frisst gerade!");
    }

    public void Bellen()
    {
        Console.WriteLine("Der Hund " + Name + " bellt gerade!");
    }
}
```

**Abb. 40, Die Klasse Hund mit Eigenschaften und Methoden**

```
class Wachhund : Hund
{
    //Methoden

    public void BewacheHaus()
    {
        Console.WriteLine("Der Hund " + Name + " bewacht das Haus!");
    }

}
```

**Abb. 41, Die Klasse Wachhund erbt von der Klasse Hund**

```
static void Main(string[] args)
{
    Hund coco = new Hund();
    coco.Name = "Coco";
    coco.Alter = 11;
    coco.Rasse = "Weißer Schäferhund";
    coco.Fressen();
    coco.Bellen();

    Wachhund jil = new Wachhund();
    jil.Name = "Jil";
    jil.Alter = 9;
    jil.Rasse = "Mischling";
    jil.BewacheHaus();
    jil.Fressen();
    jil.Bellen();
}
```

**Abb. 42, Erzeugung von zwei Objekten der Klassen Hund und Wachhund**

### Wichtig:

In C# kann nicht von **mehreren** Klassen gleichzeitig geerbt werden.

## VERERBTE KONSTRUKTOREN

- Es wird immer erst der Konstruktor aufgerufen, der an eine andere Klasse vererbt, bevor der Konstruktor der erbenden Klasse aufgerufen wird
- Wenn in der vererbenden Klasse ein Konstruktor mit Parametern erstellt wurde, muss dieser so auch in der erbenden Klasse nachgebaut werden
- Nachdem die Parameter an die erbende Klasse übergeben wurden, müssen sie noch an den Konstruktor der vererbenden Klasse übergeben werden

```
class BasisKlasse
{
    public string Name { get; set; }

    public int Age { get; set; }

    public BasisKlasse(string name, int age)
    {
        Name = name;
        Age = age;
        Console.WriteLine("Basisklassenkonstruktor wird ausgeführt...");
    }
}
```

**Abb. 43, Basisklasse (vererbende Klasse)**

```
class ErbendeKlasse : BasisKlasse
{
    public ErbendeKlasse(string name, int age) : base(name, age)
    {
        Console.WriteLine("ErbendeKlasse Konstruktor wird ausgeführt...");
    }

}
```

Abb. 44, ErbendeKlasse, Parameter werden an Basiskonstruktor weitergegeben

```
static void Main(string[] args)
{
    ErbendeKlasse Objekt = new ErbendeKlasse("Peter", 33);
    Console.ReadKey();
}
```

Abb. 45, Objekt der erbenden Klasse mit Parametern

## DIE SCHLÜSSELWÖRTER IS UND AS

Das is-Schlüsselwort verwendet man, um zu prüfen, ob ein Objekt von einer Klasse kompatibel mit einem bestimmten Datentyp ist.

**Beispiel:**

```
Console.WriteLine("Hallo Welt" is string);
```

Abb. 46, Überprüfung, ob „Hallo Welt“ ein String ist

```
Console.WriteLine(person2 is Object);
```

Abb. 47, Überprüfung, ob person2 ein Objekt ist

**Ausgabe beider Zeilen:** True

Mit dem **as**-Operator können wir ein Objekt in einen anderen kompatiblen Typ konvertieren. Im Gegensatz zum is-Operator wird kein True oder False zurückgegeben, sondern das Objekt selbst in umgewandelter Form.

```
static void Main(string[] args)
{
    object[] objects = new object[4];

    objects[0] = "Hallo Welt";
    objects[1] = "Hello World";
    objects[2] = 22;
    objects[3] = false;

    foreach(object o in objects)
    {
        string text = o as string;

        if (text != null)
        {
            Console.WriteLine(text);
        }
        else
        {
            Console.WriteLine("Das Objekt war kein String!");
        }
    }

    Console.ReadKey();
}
```

Abb. 48, Das Schlüsselwort as innerhalb einer Foreach-Schleife

```
Hallo Welt
Hello World
Das Objekt war kein String!
Das Objekt war kein String!
```

Abb. 49, Ausgabe