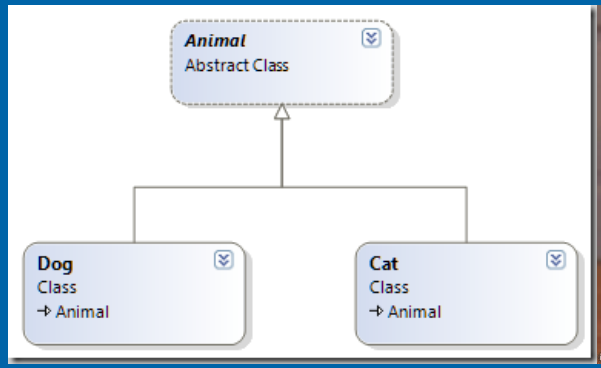
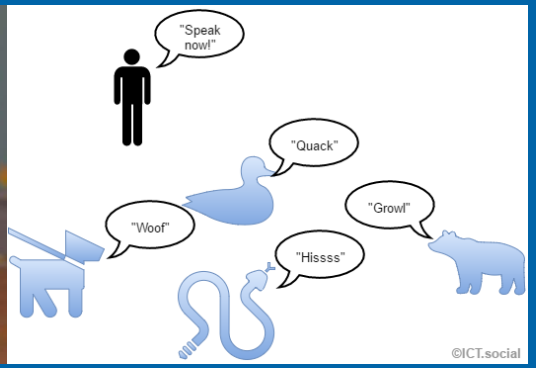
Klassen IV – Vererbung





Vererbung, Klassendiagramme, Konstruktoren-Ketten
Klassen-Kompatibilität

Objektorientierte Programmierung

- Kapselung
- **Vererbung**
- Polymorphie

Motivation

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

    public Car(int max) => MaxSpeed = max;
    public Car() => MaxSpeed = 100;

    public int Speed
    {
        get => CurrentSpeed;
        set
        {
            CurrentSpeed = value;
            if (CurrentSpeed > MaxSpeed)
                CurrentSpeed = MaxSpeed;
        }
    }
}
```

```
static void Main(string[] args)
{
    Car myGolf = new Car(160);
    myGolf.Speed = 80;
    Console.WriteLine(
        $"Geschwindigkeit: {myGolf.Speed}");
}
```

- Erweitern der Klasse bisher nur möglich, wenn die Klasse neu geschrieben wird
- Viele Autos teilen sich gemeinsame Funktionalitäten

Vererbung

```
class Car
{...}

// Minivan "is-a" Car
class MiniVan : Car
{
}
```

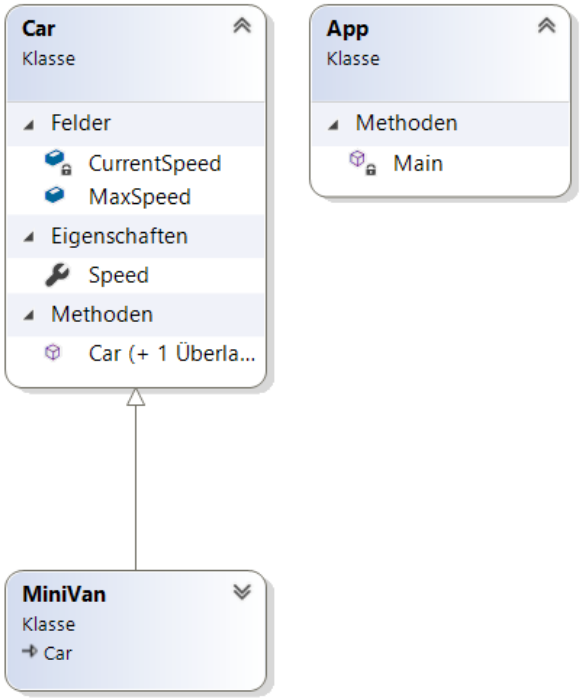
```
MiniVan myVan = new MiniVan();
myVan.Speed = 50;
Console.WriteLine(
    $"Geschwindigkeit: {myVan.Speed}");
```

- **MiniVan** hat alle Member von **Car** geerbt
- der Code der Klasse **Car** wurde wiederverwendet
- Zugriffsmodifizierer bleiben bestehen:

```
// Compiler-Fehler:
myVan.CurrentSpeed = 180;
```

Einschränkungen

- **private** Member können nur aus der eigenen Klasse heraus genutzt werden
- **public** Member sind auch von den abgeleiteten Klassen nutzbar
- Konstruktoren werden, auch wenn sie **public** sind, nicht vererbt; sie erstellen nur die eigenen Objekte
- Anders als in C++ erlaubt C# nur das Erben von **einer** Klasse (*jedoch das Implementieren mehrerer Interfaces*)



- Klassen können nach dem Erben erweitert werden
- von Klassen kann beliebig oft geerbt werden
- alles in C# erbt ursprünglich von **System.Object**
- für Klassenstrukturen gibt es eine einfache Darstellung: UML

Erweitern von Klassen

```
class Employee
{
    public string Name;
    public void Work()
        => Console.WriteLine("Arbeiten");
}

// Boss "is-an" Employee
class Boss : Employee
{
    public void Fire(Employee employee)
        => Console.WriteLine($"{employee.Name} wird gefeuert");
    public void Hire(Employee employee)
        => Console.WriteLine($"{employee.Name} wird eingestellt");
}
```

- Klassen können in ihrer Funktionalität erweitert werden
- abgeleitete Klassen können nicht weniger restriktiv im Zugriff sein; nur gleich oder sogar restriktiver

Beispiel für Nutzung von Fremdcode

```
using System;
using System.Windows.Forms;

class MyForm : Form
{ }

class App
{
    [STAThread]
    public static void Main()
    {
        Application.Run(new MyForm());
    }
}
```

- Fenstermanagement und GUI Entwicklung beruht auf Vererbung
- erste GUI-Applikation: 7 Zeilen

Zusätzliche Zugriffsmodifizierer

- Jede Klasse kann entweder als **public** oder **internal** deklariert sein (Standard: **internal**)

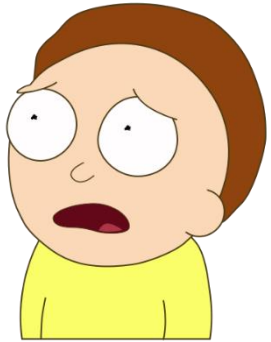
- Klassen können mit **sealed** versiegelt werden. Damit ist das Erben davon ausgeschlossen (Bsp.: **System.String**)

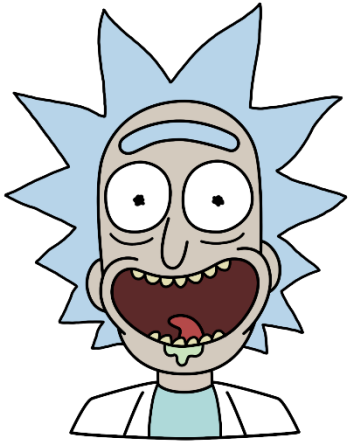
```
// Minivan "is-a" Car
public class MiniVan : Car
{
}
```

Compiler-Fehler

```
sealed class NewCar : Car
{
}

// Compiler-Fehler
class OldCar : NewCar
{
}

// Compiler-Fehler
class MyString : String
{
}
```

Was ist denn jetzt mit den Konstruktoren? Die werden ja nicht mit vererbt. Aber wie kann ich dann private Felder der Basisklasse setzen?
Oder wie kann ich gute Konstruktoren der Basisklasse nutzen?





Konstruktoren werden generell nicht mit vererbt. Aber man kann **alle** eigenen Konstruktoren oder die der Basisklasse nutzen.

Das nennt man „Constructor Chaining“.

Tatsächlich brauchen wir die Konstruktoren der Basisklasse, ob wir wollen oder nicht.

Constructor Chaining

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

    public Car(int maxSpeed, int numberOfSeats)
    {
        Console.WriteLine("2 arg ctor");
        this.MaxSpeed = maxSpeed;
        this.NumberOfSeats = numberOfSeats;
    }
    public Car(int maxSpeed) : this(maxSpeed, 5)
    {
        Console.WriteLine("1 arg ctor");
    }
    public Car() : this(100)
    {
        Console.WriteLine("0 arg ctor");
    }
}
```

- **this** ruft den eigenen Konstruktor der Klasse auf
- erlaubt eine schnelle Überladung von Konstruktoren
- es muss nur eine Implementierung jeweils gepflegt werden

Konstruktorenreihenfolge

```
Car myGolf = new Car();
Car myLancia = new Car(150);
```

```
>inheritanceExample04.exe
2 arg ctor of Car
1 arg ctor of Car
0 arg ctor of Car
2 arg ctor of Car
1 arg ctor of Car
```

- Konstruktoren werden nacheinander aufgerufen
- zuerst springt man in den Konstruktor, welcher mit **this** (oder **base**) deklariert ist
- im Anschluss wird die eigene Logik ausgeführt

Nutzung der Konstruktoren der Basisklasse

```
class MiniVan : Car
{
    public void Sound() => Console.WriteLine("Summ Summ");
    public MiniVan() : this(100)
        => Console.WriteLine("0 arg ctor of MiniVan");
    public MiniVan(int maxSpeed) : base(maxSpeed, 8)
        => Console.WriteLine("1 arg ctor of MiniVan");
    public MiniVan(int maxSpeed, int numberOfSeats) : base(maxSpeed, numberOfSeats)
        => Console.WriteLine("0 arg ctor of MiniVan");
}
```

- Die Konstruktoren der Basisklasse werden mit der gleichen Syntax genutzt

```
>inheritanceExample04.exe
2 arg ctor of Car
1 arg ctor of MiniVan
0 arg ctor of MiniVan
```

Implizite Nutzung der Basis-Konstruktoren

Generell gilt:

- es muss immer ein Konstruktor der Basisklasse aufgerufen werden (Der Standardkonstruktor wird implizit aufgerufen, wenn kein anderer angegeben ist)
- ein Objekt einer Unterklasse ist immer auch ein Objekt einer Basisklasse („is-a“)
- wird kein expliziter Basisklassenkonstruktor verwendet, kommt der Standardkonstruktor zum Tragen

```
class Zug
{
    public Zug()
        => Console.WriteLine("Zug ctor");
    public Zug(int id)
        => Console.WriteLine($"Zug mit der Nummer {id}");
}

class PersonenZug : Zug
{
    public PersonenZug() { }
}

class GueterZug : Zug
{
    public GueterZug(int id) : base(id) { }
}
```

auch ohne seine explizite Nutzung wird der Standardkonstruktor von Zug hier aufgerufen

erbende Klassen sehen, welche Konstruktoren für sie zur Verfügung stehen und müssen davon mindestens einen benutzen

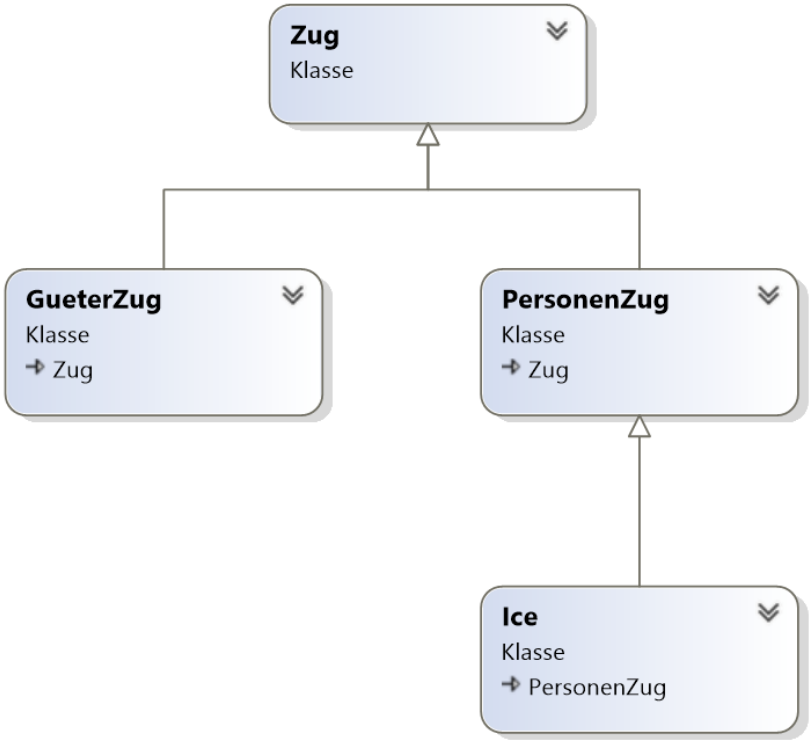
```
Zug zug1 = new Zug();
PersonenZug ice = new PersonenZug();
GueterZug stahl = new GueterZug(12);
```

```
>inheritanceExample06.exe
Zug ctor
Zug ctor
Zug mit der Nummer 12
```

```
class Zug
{
    public Zug() => Console.WriteLine("Zug-ctor");
}

class PersonenZug : Zug
{
    public PersonenZug() : base()
        => Console.WriteLine("PersonenZug-ctor");
}

class Ice : PersonenZug
{
    public Ice()
        => Console.WriteLine("Ice-ctor");
}

class GueterZug : Zug
{
    public GueterZug()
        => Console.WriteLine("GueterZug-ctor");
}
```

```
static void Main(string[] args)
{
    Console.WriteLine("Ice:");
    Ice ice = new Ice();
    Console.WriteLine("GueterZug:");
    GueterZug gueter = new GueterZug();
}
```

```
>inheritanceExample07.exe
Ice:
Zug-ctor
PersonenZug-ctor
Ice-ctor

GueterZug:
Zug-ctor
GueterZug-ctor

>
```




```
>inheritanceExample07.exe
Ice:
Zug-ctor
PersonenZug-ctor
Ice-ctor

GueterZug:
Zug-ctor
GueterZug-ctor

>
```

Vererbung: Kompatibilität zwischen Typen

- ein **Ice** ist immer auch ein **PersonenZug** und auch ein **Zug**
- Objekte haben immer einen statischen Typen und einen dynamischen Typen
- Unterklassen sind Spezialfälle der Oberklassen
- Objekte von Unterklassen können überall verwendet werden, wo Objekte von der Basisklasse erwartet werden

Typen von Klassen

```
class App
{
    static void Main(string[] args)
    {
        Zug zug = new Zug();
        zug = new PersonenZug();
        zug = new Ice();
    }
}
```

```
// PersonenZug "is-a" Zug
class PersonenZug : Zug
{...}
```

```
//Compiler-Fehler:
PersonenZug personen = new Zug();
```

Zuweisung:	statischer Typ von zug	dynamischer Typ von zug
`Zug zug = new Zug();`	Zug	Zug
`zug = new PersonenZug();`	Zug	PersonenZug
`zug = new Ice();`	Zug	Ice

Laufzeitprüfung

- der dynamische Typ einer Klasse kann zur Laufzeit geprüft werden

- Typtest liefert bei **null**-Werten immer **false**

```
Zug zug = new Ice();
Console.WriteLine("zug ist ein Zug? " + (zug is Zug)); // true
Console.WriteLine("zug ist ein PersonenZug? " + (zug is PersonenZug)); // true
Console.WriteLine("zug ist ein Ice? " + (zug is Ice)); // true
zug = null;
Console.WriteLine("zug ist ein Ice? " + (zug is Ice)); // false
```

- **zug** kann nicht ohne Weiteres auf seine **Ice**-Methoden zugreifen

Typumwandlung

- wenn das Objekt **zug** nun vom dynamischen Typ **Ice** ist, kann auch auf die Member von **Ice** zugegriffen werden

- für die Nutzung muss ein **expliziter Cast** erfolgen (*InvalidCastException*)

```
Zug zug1 = new Ice();
PersonenZug zug2 = (PersonenZug)zug1;
Ice zug3 = (Ice)zug1;
zug1.Drive(); // Compiler-Fehler
zug2.Drive(); // Compiler-Fehler
zug3.Drive();
```

```
class Ice : PersonenZug
{
    public Ice() { }
    public void Drive()
        => Console.WriteLine("Brumm brumm!");
}
```

Typumwandlung

Alternative: **as**-Operator

- gleicht einem expliziten Cast, jedoch ohne Exception zur Laufzeit

- bei unerlaubten Casts wird der Wert **null** zugeordnet

```
Zug zug1 = new Ice();
PersonenZug zug2 = zug1 as PersonenZug;
Ice zug3 = zug1 as Ice;
```

```
Zug zug = new Zug();
Ice ice = zug as Ice;
//ice = null
```

- Casts funktionieren in beide Richtungen (solange der dynamische Typ passt)

```
Ice zug1 = new Ice();
Zug zug2 = zug1; // Upcast
Ice zug3 = zug2 as Ice; // Downcast
```

- Die Richtung wird im Klassendiagramm klarer

Zusammenfassung

- Vererbung hilft dabei, Code wiederzuverwenden
- Alles in C# ist ein **System.Object**
- Konstruktoren können horizontal und vertikal verketten
- UML hilft beim Klassendesign und beim Überblick
- Klassentypen können vertikal gecastet werden