Objektorientiertes Programmieren

Grundlagen

11.04.2020

OOP vs. Prozedural

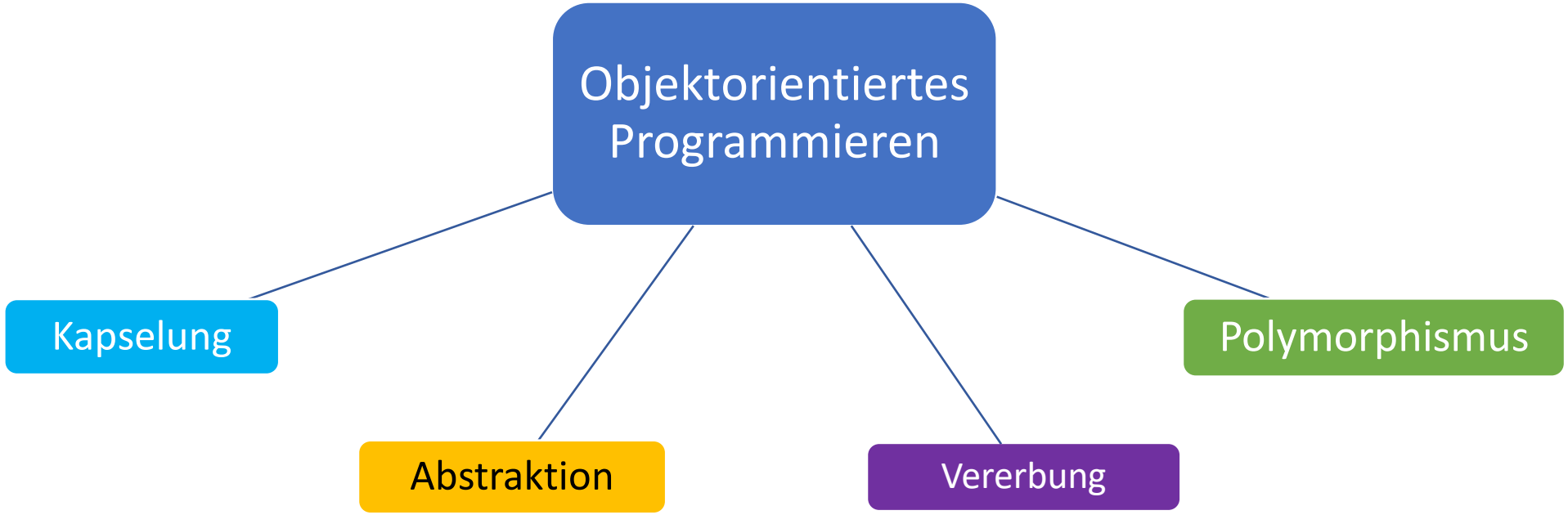
```javascript
let carTank = 50
let consumptionEachKm = 14.1
let pricePerLitre = 1.45

function fillCar(maxTank, pricePerLitre) {
  let cost = maxTank * pricePerLitre
  console.log("You filled your car. Cost: " + cost)
}

function drive(kilometres, consumptionEachKm) {
  let consumedFuel = kilometres * consumptionEachKm
  console.log("You drove " + kilometres + " and used " +
  consumedFuel + "l of fuel.")
}

// ...

fillCar(carTank, pricePerLitre)
drive(50, consumptionEachKm)
```

Ein Beispiel in JavaScript…

```javascript
let car = {
  carTank = 50,
  consumptionEachKm = 14.1,
  pricePerLitre = 1.45,
  fillCar: function () {
    let cost = this.maxTank * this.pricePerLitre
    console.log("You filled your car. Cost: " + cost)
  },
  drive: function(kilometres) {
    let consumedFuel = kilometres * this.consumptionEachKm
    console.log("You drove " + kilometres + " and used " +
    consumedFuel + "l of fuel.")
    this.carTank -= consumedFuel
  }
}
```

Und dann einmal als Objekt

Prinzipien des OOP



Anmerkung

Die objektorientierte Programmierung ist nicht zwingend von einer Programmiersprache abhängig. Sobald man alle Prinzipien versteht, lässt es sich ziemlich leicht auf alle anderen Programmiersprachen übertragen.

Wie sich dieses Paradigma allerdings umsetzen lässt, muss in der jeweiligen Dokumentation der Programmiersprache eingesehen werden

Kapselung

```
let carTank = 50
let consumptionEachKm = 14.1
let pricePerLitre = 1.45

function fillCar(maxTank, pricePerLitre) {
  let cost = maxTank * pricePerLitre
  console.log("You filled your car. Cost: " + cost)
}

function drive(kilometres, consumptionEachKm) {
  let consumedFuel = kilometres * consumptionEachKm
  console.log("You drove " + kilometres + " and used " +
  consumedFuel + "l of fuel.")
}

// ...

fillCar(carTank, pricePerLitre)
drive(50, consumptionEachKm)
```

```
let car = {
  carTank = 50,
  consumptionEachKm = 14.1,
  pricePerLitre = 1.45,
  fillCar: function () {
    let cost = this.maxTank * this.pricePerLitre
    console.log("You filled your car. Cost: " + cost)
  },
  drive: function(kilometres) {
    let consumedFuel = kilometres * this.consumptionEachKm
    console.log("You drove " + kilometres + " and used " +
    consumedFuel + "l of fuel.")
    this.carTank -= consumedFuel
  }
}

// ...
car.fillCar()
car.drive(50)
```

Properties, Attribute, Felder

Methoden, functions

Kapselung

access modifier

„private“ Felder

Nur die Klasse selbst kann direkt auf sie zugreifen.

„public“ Methoden

Jede Klasse/jedes Objekt kann auf diese Methode zugreifen.
In diesem Fall: Indirekter Zugriff auf private Felder von außen.

```
public void Drive(int kilometres)
```

Jede Klasse/jedes Objekt kann auf diese Methode zugreifen.
Logik zur Ausführen der Aktion „Fahren“.

```
namespace ConsoleApp1
{
    0 references
    class Car
    {
        /* Nehmen wir an, da sind gerade 50l im Tank
         * und es passen nur 50l rein.
         */
        private double carTank = 50.0;
        private double consumptionEachKm = 0.141;

        0 references
        public double GetConsumptionEachKm()
        {
            return consumptionEachKm;
        }

        0 references
        public double GetCarTank()
        {
            return carTank;
        }

        0 references
        public void Drive(int kilometres)
        {
            double consumedFuel = kilometres * consumptionEachKm;

            if (consumedFuel > carTank)
            {
                Console.WriteLine($"Not enough fuel available in order to drive {kilometres}km.");
            }
            else
            {
                carTank -= consumedFuel;
                Console.WriteLine($"You drove {kilometres} and used {consumedFuel}l of fuel.");
            }
        }
    }
}
```

Kapselung

Initialisierung eines Objekts der Klasse „Car“

```
Tank: 50l
Fuel consumption: 14,0999999999999998l/100km
You drove 50 and used 7,0499999999999999l of fuel.
Not enough fuel available in order to drive 500000km.
```

```
using System;

namespace ConsoleApp1
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");

            Car honda = new Car();

            Console.WriteLine("Car details:");
            Console.WriteLine($"Tank: {honda.GetCarTank()}l");
            Console.WriteLine($"Fuel consumption: {honda.GetConsumptionEachKm() * 100}l/100km");

            honda.Drive(50);

            honda.Drive(500000);
        }
    }
}
```

Zugriff auf klassendefinierte Methoden

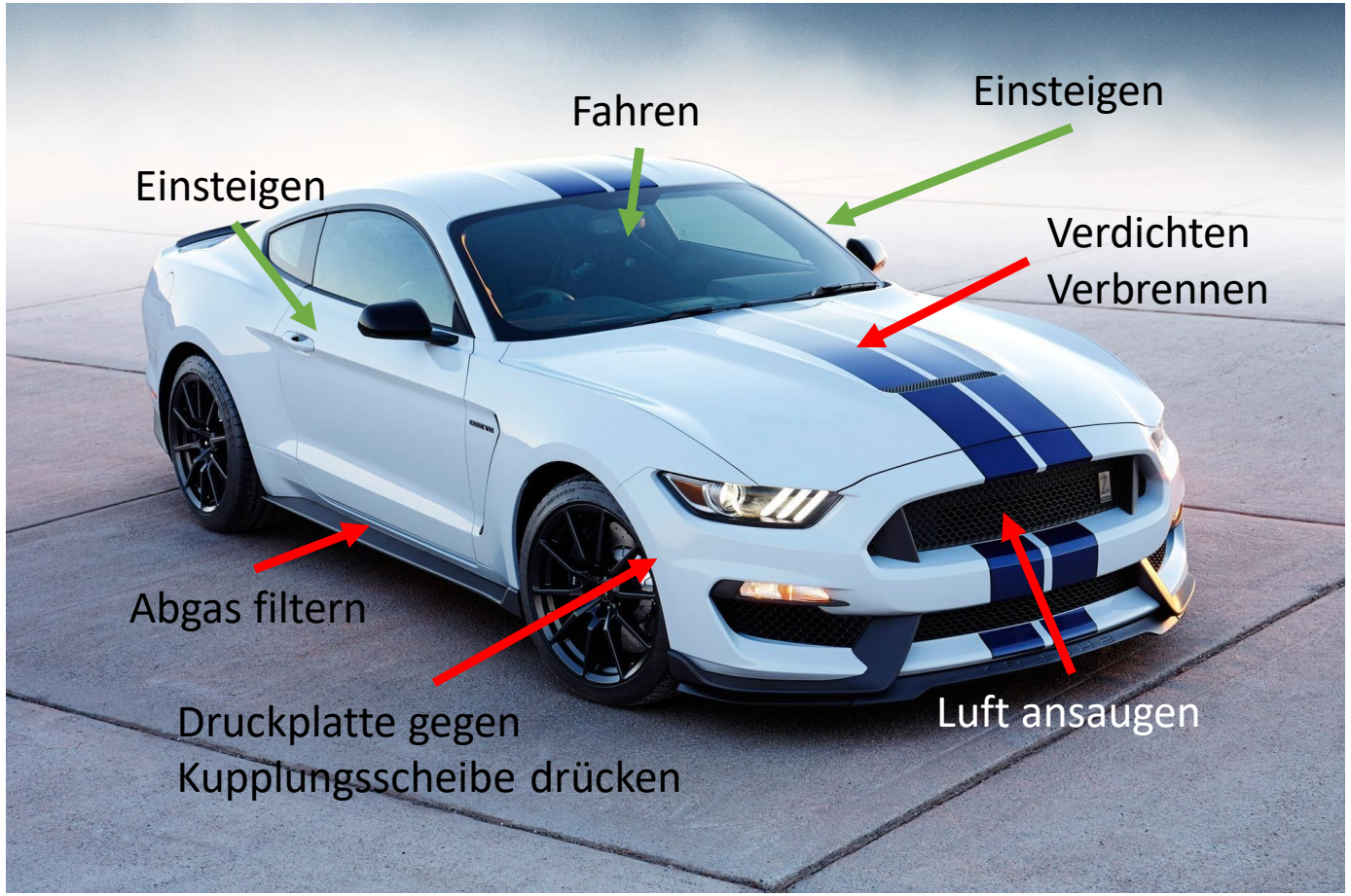
Kapselung

Fazit

- Einheits-/Klasseninterne Properties/Felder bleiben „**private**“.
- Zugriff auf Felder erfolgen nur indirekt über sog. **Getter/Setter-Methoden**
- Durch interne Felder kann bspw. **die Anzahl an Parameter pro Methode verringert** werden → Je weniger, desto besser!

Abstraktion

Ziemlich offensichtlich,
jeder kann das machen
(und sollte das wissen?)



Ziemlich unscheinbar
Interne Prozessabläufe
Schön zu wissen, aber
juckt uns theoretisch
nicht.

Abstraktion

Verbrauchlogik wird als
eine neue „**private**“ Methode extrahiert

```
0 references
public void Drive(int kilometres)
{
    double consumedFuel = kilometres * consumptionEachKm;

    if (consumedFuel > carTank)
    {
        Console.WriteLine($"Not enough fuel available in order to drive {kilometres}km.");
    }
    else
    {
        carTank -= consumedFuel;
        Console.WriteLine($"You drove {kilometres} and used {consumedFuel}l of fuel.");
    }
}
```

```
2 references
public void Drive(int kilometres)
{
    double consumedFuel = kilometres * consumptionEachKm;

    if (ConsumeFuel(consumedFuel))
    {
        Console.WriteLine($"You drove {kilometres} and used {consumedFuel}l of fuel.");
    }
    else
    {
        Console.WriteLine($"Not enough fuel available in order to drive {kilometres}km.");
    }
}

1 reference
private bool ConsumeFuel(double fuelToConsume)
{
    bool canConsume = (fuelToConsume <= carTank);

    if (canConsume)
    {
        carTank -= fuelToConsume;
        return true;
    }
    else
    {
        return false;
    }
}
```

Abstraktion

```
1 reference
private bool ConsumeFuel(double fuelToConsume)
{
    bool canConsume = (fuelToConsume <= carTank);

    if (canConsume)
    {
        carTank -= fuelToConsume;
        return true;
    }
    else
    {
        return false;
    }
}
```

Eine Methode, die nur innerhalb der Klasse `Car` verwendet wird und auch nur verwendet werden soll!

Dafür gibt‘s ja
die Methode `Drive()`
und als Person selbst kann man dem Fahrzeug nicht direkt sagen, dass er Sprit verbrauchen soll

Abstraktion

Fazit

- Mithilfe von **Abstraktion** möchte man ein sauberes Interface zur Interaktion des jeweiligen Objekts geben
- Klasseninterne Methoden, die der „Endnutzer“ nicht benötigt oder nicht verwenden soll, können mit dem *Access Modifier* „**private**“ von außen unzugänglich machen.
- Dient der Übersicht und besseren Gliederung des Codes (Kein Spaghetti-Code!)

Vererbung

Car
- carTank: double - consumptionEachKm: double
+ GetCarTank(): double + GetConsumptionEachKm(): double + Drive(int): void - ConsumeFuel(double): bool

Vererbung

Car
- carTank: double - consumptionEachKm: double
+ GetCarTank(): double + GetConsumptionEachKm(): double + Drive(int): void - ConsumeFuel(double): bool

Viewer does not support full SVG 1.1

Car2
- carTank: double - consumptionEachKm: double
+ GetCarTank(): double + GetConsumptionEachKm(): double + Drive(int): void - ConsumeFuel(double): bool

Viewer does not support full SVG 1.1

Vererbung

Car
- carTank: double - consumptionEachKm: double
+ GetCarTank(): double + GetConsumptionEachKm(): double + Drive(int): void - ConsumeFuel(double): bool

Viewer does not support full SVG 1.1

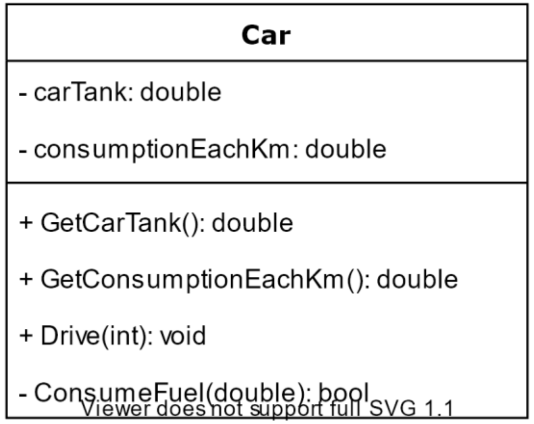
Car2
- carTank: double - consumptionEachKm: double
+ GetCarTank(): double + GetConsumptionEachKm(): double + Drive(int): void - ConsumeFuel(double): bool

Viewer does not support full SVG 1.1

Car3
- carTank: double - consumptionEachKm: double
+ GetCarTank(): double + GetConsumptionEachKm(): double + Drive(int): void - ConsumeFuel(double): bool

Viewer does not support full SVG 1.1

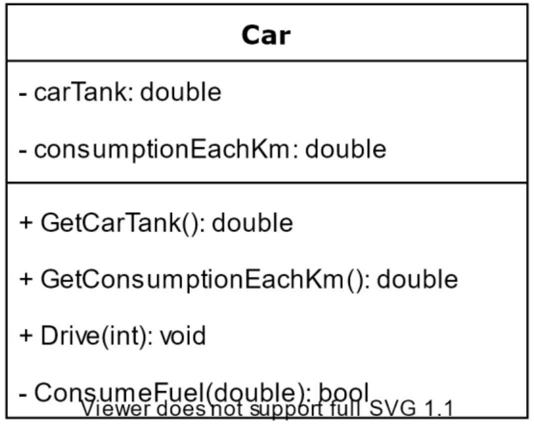
Vererbung


Car

- carTank: double
- consumptionEachKm: double

+ GetCarTank(): double
+ GetConsumptionEachKm(): double
+ Drive(int): void
- ConsumeFuel(double): bool

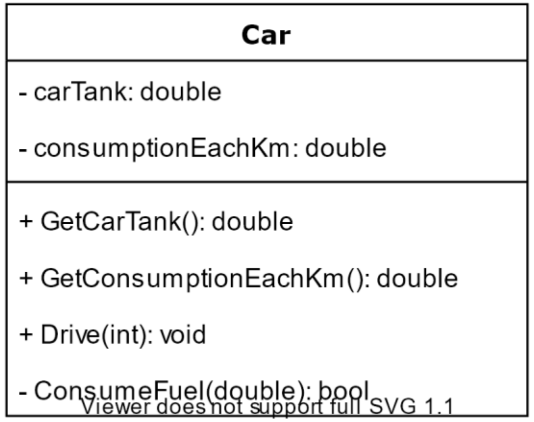
Viewer does not support full SVG 1.1


Car

- carTank: double
- consumptionEachKm: double

+ GetCarTank(): double
+ GetConsumptionEachKm(): double
+ Drive(int): void
- ConsumeFuel(double): bool

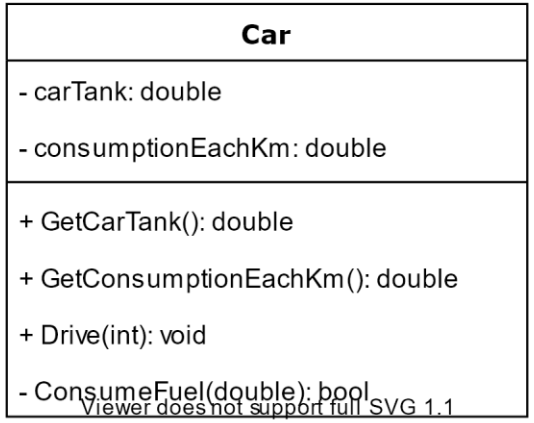
Viewer does not support full SVG 1.1


Car

- carTank: double
- consumptionEachKm: double

+ GetCarTank(): double
+ GetConsumptionEachKm(): double
+ Drive(int): void
- ConsumeFuel(double): bool

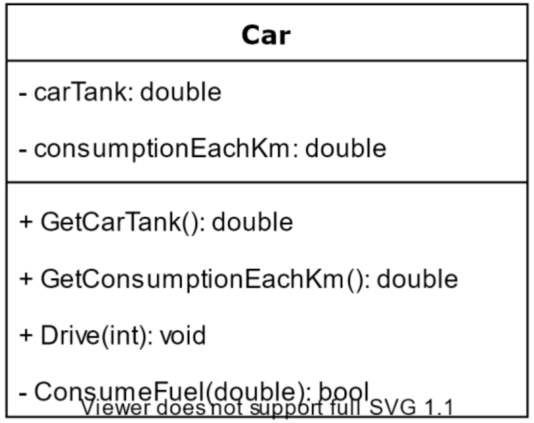
Viewer does not support full SVG 1.1


Car

- carTank: double
- consumptionEachKm: double

+ GetCarTank(): double
+ GetConsumptionEachKm(): double
+ Drive(int): void
- ConsumeFuel(double): bool

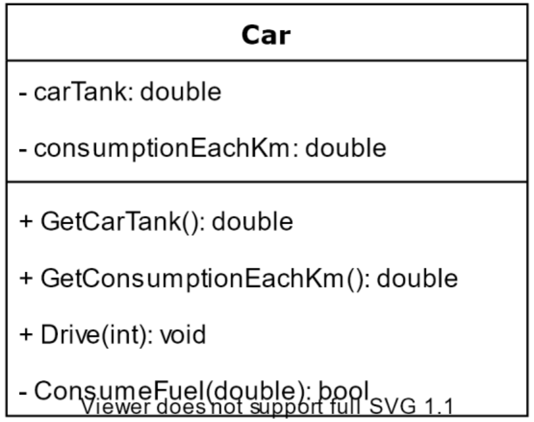
Viewer does not support full SVG 1.1


Car

- carTank: double
- consumptionEachKm: double

+ GetCarTank(): double
+ GetConsumptionEachKm(): double
+ Drive(int): void
- ConsumeFuel(double): bool

Viewer does not support full SVG 1.1


Car

- carTank: double
- consumptionEachKm: double

+ GetCarTank(): double
+ GetConsumptionEachKm(): double
+ Drive(int): void
- ConsumeFuel(double): bool

Viewer does not support full SVG 1.1

Vererbung


Car
- carTank: double
- consumptionEachKm: double
+ GetCarTank(): double
+ GetConsumptionEachKm(): double
+ Drive(int): void
- ConsumeFuel(double): bool
Viewer does not support full SVG 1.1

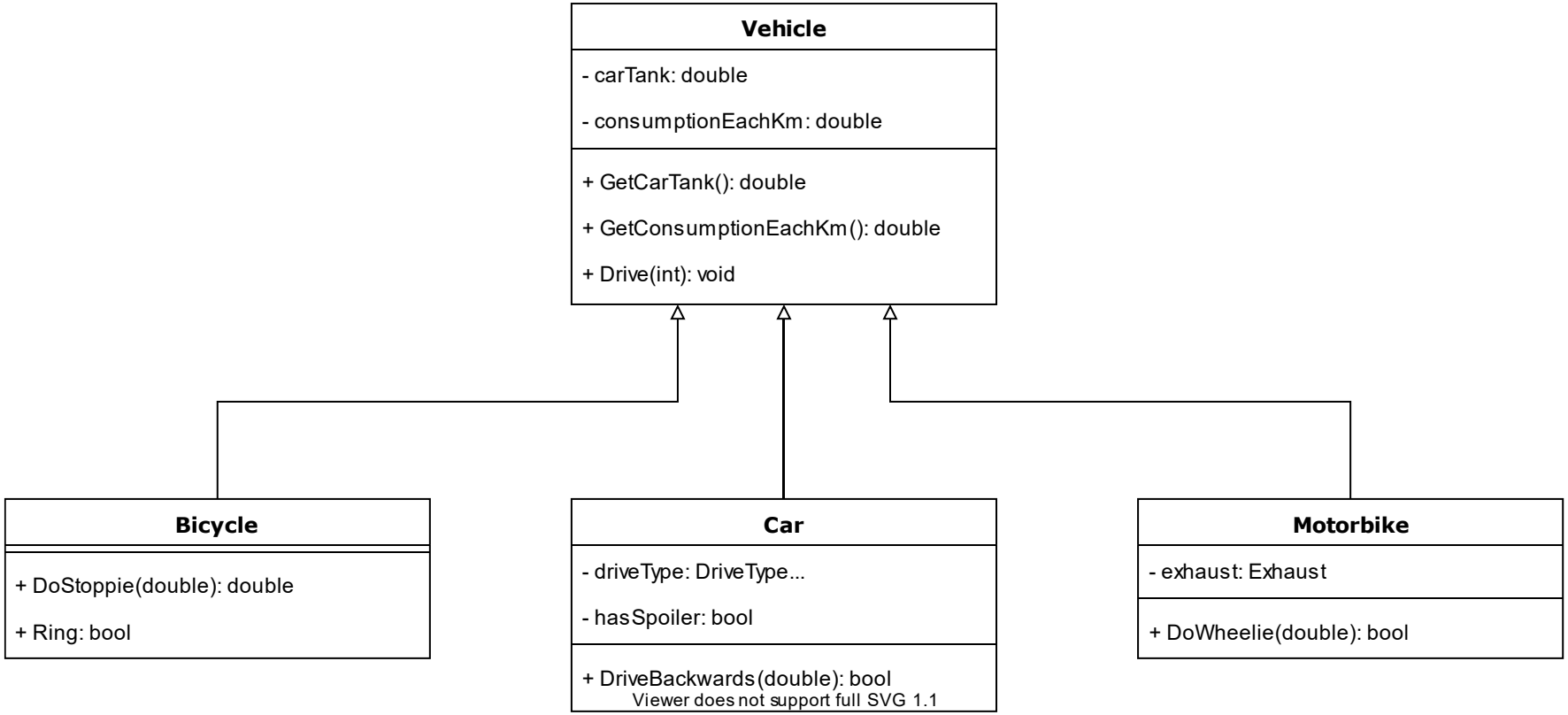
Vererbung


Car
- carTank: double
- consumptionEachKm: double
+ GetCarTank(): double
+ GetConsumptionEachKm(): double
+ Drive(int): void
- Car(int, Car[], int[], int[]): void

Vererbung

Ob du behindert bist?

Vererbung


Vehicle
- carTank: double
- consumptionEachKm: double
+ GetCarTank(): double
+ GetConsumptionEachKm(): double
+ Drive(int): void
Bicycle
+ DoStoppie(double): double
+ Ring: bool
Car
- driveType: DriveType...
- hasSpoiler: bool
+ DriveBackwards(double): bool
Viewer does not support full SVG 1.1
Motorbike
- exhaust: Exhaust
+ DoWheelie(double): bool

Vererbung

```
namespace ConsoleApp1
{
    0 references
    class Car
    {
        /* Nehmen wir an, da sind gerade 50l im Tank
         * und es passen nur 50l rein.
         */
        private double carTank = 50.0;
        private double consumptionEachKm = 0.141;

        0 references
        public double GetConsumptionEachKm()
        {
            return consumptionEachKm;
        }

        0 references
        public double GetCarTank()
        {
            return carTank;
        }

        0 references
        public void Drive(int kilometres)
        {
            double consumedFuel = kilometres * consumptionEachKm;

            if (consumedFuel > carTank)
            {
                Console.WriteLine($"Not enough fuel available in order to drive {kilometres}km.");
            }
            else
            {
                carTank -= consumedFuel;
                Console.WriteLine($"You drove {kilometres} and used {consumedFuel}l of fuel.");
            }
        }
    }
}
```

... umbenannt in...

„private“ zu „protected“

... entfernt und Beschreibung aus Program.cs extrahiert ...

```
2 references
class Vehicle
{
    /* Nehmen wir an, da sind gerade 50l im Tank
     * und es passen nur 50l rein.
     */
    protected double carTank;
    protected double consumptionEach100Km;

    1 reference
    public Vehicle(double _carTank, double _consumptionEach100Km)
    {
        carTank = _carTank;
        consumptionEach100Km = _consumptionEach100Km;
    }

    3 references
    public virtual void Describe()
    {
        Console.WriteLine("Car details:");
        Console.WriteLine($"Tank: {carTank}l");
        Console.WriteLine($"Fuel consumption: {consumptionEach100Km}l/100km");
    }
```

Konstruktor

Dient zur Erzeugung eines Objekts aus der dementsprechenden Klasse

„virtual“

Schlüsselwort muss verwendet werden, wenn erbende Klassen diese Methode überschreiben dürfen

Vererbung

„Neue“ Klasse `Car`

```
namespace ConsoleApp1
{
    3 references
    class Car : Vehicle
    {
        private string model;
        private bool hasSpoiler;

        1 reference
        public Car(
            double _carTank,
            double _consumptionEach100Km,
            string _model,
            bool _hasSpoiler
            ) : base(_carTank, _consumptionEach100Km)
        {
            model = _model;
            hasSpoiler = _hasSpoiler;
        }
    }
}
```

Vererbung

C#:
`class <KLASSE> : <ÜBERKLASSE>`

Java:
`class <KLASSE>`
`extends <ÜBERKLASSE>`

Python:
`class KLASSE(ÜBERKLASSE):`

Konstruktor

Besonderheit bei C#:
„super()“-Aufruf in-line mit Konstruktor

Java:

```
public Car(…) {
  super(…)
}
```

Attribute, Felder

Klasseneigene Felder

Vererbung

```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");

        Car honda = new Car(50.0, 14.1, "Honda Civic", false);

        honda.Describe();

        honda.Drive(50);

        honda.Drive(500000);
    }
}
```

```
Hello World!
Car details:
Tank: 50l
Fuel consumption: 14,1l/100km
You drove 50 and used 7,0499999999999991 of fuel.
Not enough fuel available in order to drive 500000km.
```

Vererbung

Fazit

- Erstellung von einer generalisierten Klasse möglich
- Vererbung überträgt Konstruktor + in der Überklasse definierte Methoden
- Überklasse kann auch eine abstrakte Klasse sein (Nicht zu verwechseln mit Abstraktion! – eine abstrakte Klasse ist eine Klasse, die als „Schablone“ oder Vorschrift an Klassen dient, die davon erben. Man kann davon keine Objekte erzeugen)

Polymorphismus

```
namespace ConsoleApp1
{
    3 references
    class Car : Vehicle
    {
        private string model;
        private bool hasSpoiler;

        1 reference
        public Car(
            double _carTank,
            double _consumptionEach100Km,
            string _model,
            bool _hasSpoiler
            ) : base(_carTank, _consumptionEach100Km)
        {
            model = _model;
            hasSpoiler = _hasSpoiler;
        }
    }
}
```

Haben bisher keine Verwendung…

```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");

        Car honda = new Car(50.0, 14.1, "Honda Civic", false);

        honda.Describe();

        honda.Drive(50);

        honda.Drive(500000);
    }
}
```

```
Hello World!
Car details:
Tank: 50l
Fuel consumption: 14,1l/100km
You drove 50 and used 7,0499999999999999l of fuel.
Not enough fuel available in order to drive 500000km.
```

Polymorphismus

Beispiel: `Describe()`

Klasse `Car`

```
3 references
public override void Describe()
{
    base.Describe();
    Console.WriteLine($"Model name: {model}");
    Console.WriteLine($"Has spoiler?: {(hasSpoiler ? "Yes" : "No")}");
}
```

```
Hello World!
Car details:
Tank: 50l
Fuel consumption: 14,1l/100km
Model name: Honda Civic
Has spoiler?: No
You drove 50 and used 7,0499999999999999l of fuel.
Not enough fuel available in order to drive 500000km.
```

Polymorphismus

Beispiel: `DriveBackwards()`

Klasse `Car`

```
0 references
public bool DriveBackwards(double _distance)
{
    double consumedFuel = _distance * (consumptionEach100Km / 100.0);

    if (base.ConsumeFuel(consumedFuel))
    {
        Console.WriteLine($"You drove {_distance} backwards and used {consumedFuel}l of fuel.");
        return true;
    }
    else
    {
        Console.WriteLine($"Not enough fuel available in order to drive {_distance}km backwards.");
        return false;
    }
}
```

```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");

        Car honda = new Car(50.0, 14.1, "Honda Civic", false);

        honda.Describe();

        honda.DriveBackwards(15.0);
    }
}
```

```
Hello World!
Car details:
Tank: 50l
Fuel consumption: 14,1l/100km
Model name: Honda Civic
Has spoiler?: No
You drove 15 backwards and used 2,11499999999999998l of fuel.
```

Polymorphismus

Beispiel: `DriveBackwards()`

```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");

        Vehicle honda = new Car(50.0, 14.1, "Honda Civic", false);

        honda.Describe();

        honda.DriveBackwards(15.0);
    }
}
```

CS1061	'Vehicle' does not contain a definition for 'DriveBackwards' and no accessible extension method 'DriveBackwards' accepting a first argument of type 'Vehicle' could be found (are you missing a using directive or an assembly reference?)	ConsoleApp1	Program.cs	15	Active

Polymorphismus

Fazit

- Durch Überschreiben von Methoden kann die Logik jeder Methode, die von einer Überklasse geerbt wird, verändert werden
- Vor allem: *Don‘t Repeat Yourself* = Weniger Arbeit