Klassen

Ein *struct* ist eine besondere (nämlich nicht ableitbare) Klasse.

Daten und zugehörige Funktionen sollten in Klassen statt *structs* gruppiert werden.

```
class CMyClass
{
        float data1;  // private, Daten ->Attribute
        double data2;
    public void doSomethingWithData() { data2 += data1; } // Funktionen -> Methoden
};
```

Daten einer Klasse werden in C# *Felder* genannt, **Funktionen** einer Klasse werden *Methoden* genannt.

Klassen und Objekte

Klasse:

- Bauplan für Objekte
- “Idee” der Objekte
- Definition aller Attribute und Methoden Klasse allein tut noch nichts

Objekt

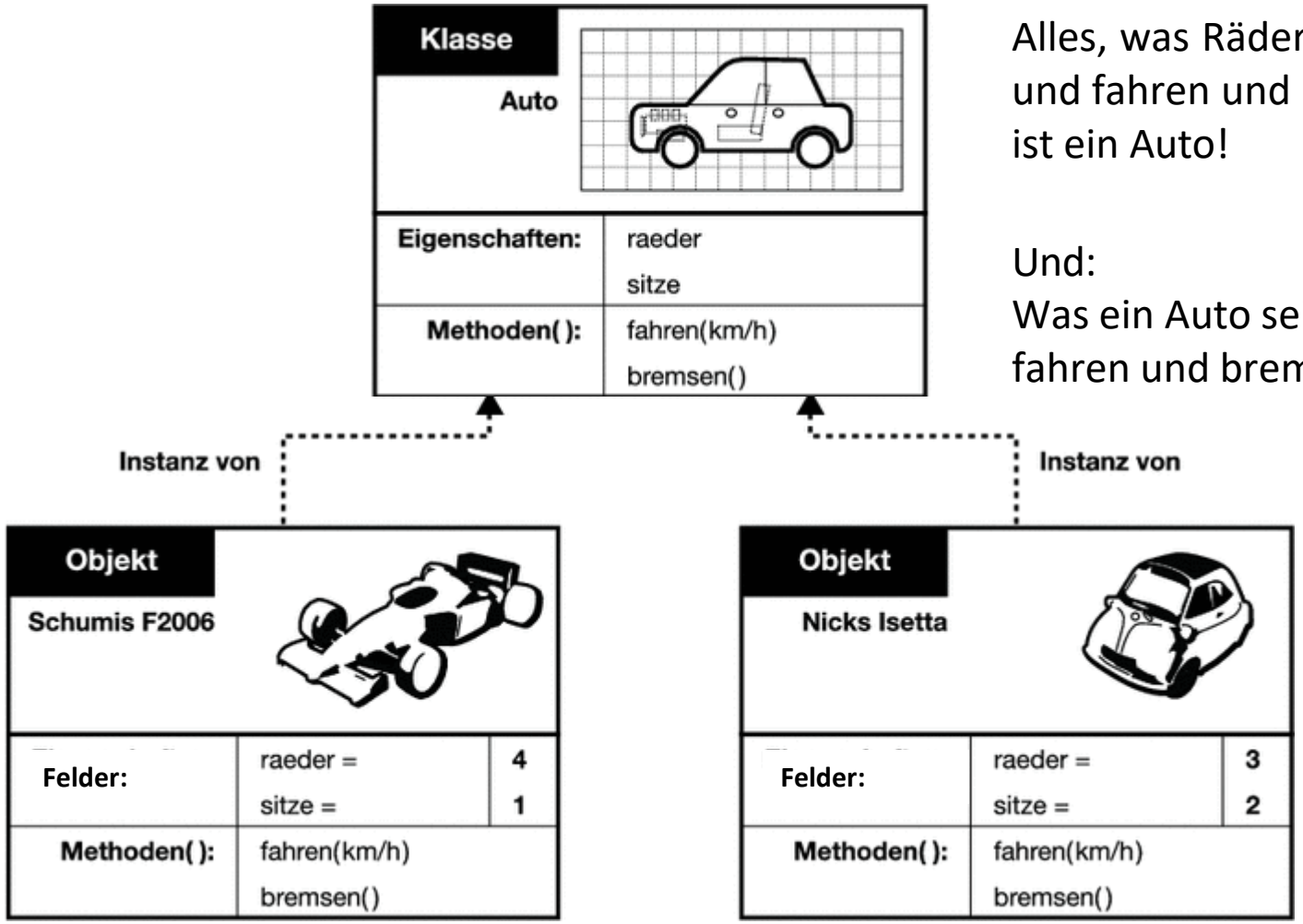
- ist ein konkretes Element (Instanz) dieser Klasse

Objekte

- Gegenstand, auf den sich jemand bezieht, auf den das Denken oder Handeln ausgerichtet ist
- Sind überall und werden als solche wahrgenommen.

Objekte der realen Welt haben:	**Objekte der OOP haben:**
Zustand	Feldwerte
Verhalten	Methoden

Objekte



Alles, was Räder und Sitze hat,
und fahren und bremsen kann,
ist ein Auto!

Und:
Was ein Auto sein will, muss Räder und Sitze haben,
fahren und bremsen können!

http://openbook.rheinwerk-verlag.de

Objektorientierung

Eine Klasse ist gekennzeichnet durch:

Felder (Instanz- bzw. Klassenvariablen)

- gehören zu den Objekten bzw. Instanzen der Klasse (z. B. Farbe eines Kleides)
- oder statisch zur Klasse selbst (z. B. Zeitpunkt des zuletzt erzeugten Objektes der Klasse)

Handlungskompetenzen (Methoden):

- entweder individuellen Objekten bzw. Instanzen zugeordnet (z. B. Verkauf eines bestimmten Kleides)
- oder der Klasse selbst (z. B. Informieren über die Anzahl bisheriger Verkäufe)

- realisieren Algorithmen

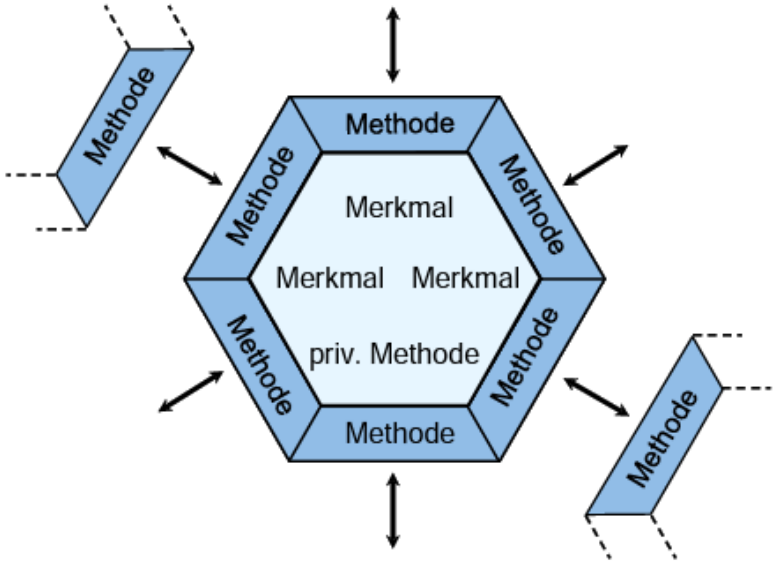
- dienen der Kommunikation zwischen Klassen bzw. deren Objekten: ein anderes Objekt oder eine andere Klasse aufzufordern, eine bestimmte Methode auszuführen

Objektorientierung

Jede Klasse ist für die Manipulation ihrer **Felder** (Merkmale, Memberwerte) selbst verantwortlich:
Felder sind eingekapselt, vor direktem Zugriff durch fremde Klassen geschützt

Methoden einer Klasse sind in der Regel von anderen Klassen ansprechbar,
sie bilden die Schnittstelle zur Kommunikation mit anderen Klassen.

Private Methoden für den ausschließlich internen Gebrauch



Klassen und Objekte in C#

- Klassendeklaration mit Schlüsselwort **class** statt struct, abgeschlossen mit „;“:

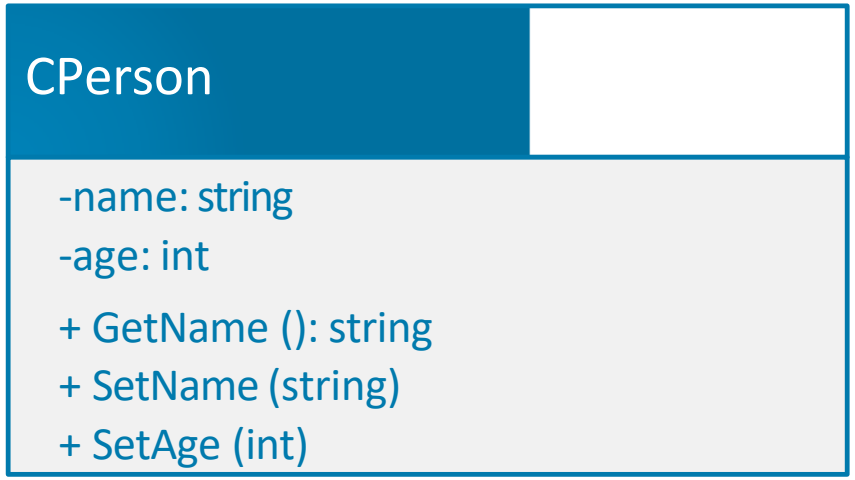
```
class Klassenname
{
  // Deklaration von Feldern und Methoden
};
```

- Enthält u.a. Deklaration (Schnittstelle) und Definition von Feldern und Methoden (Überbegriff in C#: *member*)

```
class CPerson
{   // default: alles private
  string name;
  int   age;

   public void SetName (string i_name) { name=i_name; }
   public void SetAge (int i_age) {age=i_age; }

};
```

Klassen und Objekte: UML-Klassendiagramm

```
class CPerson
{
          int age;  // private
          string name; // private

   public void SetName (string i_name) { name=i_name; }
   public string GetName() {return name;}
   public void SetAge (int i_age) {age=i_age; }
};
```



Einstiegsbeispiel

Problem mit einem C# - Einstiegsbeispiel...

- Einfache Beispiele sind für das Programmieren mit C# nicht besonders repräsentativ, z. B. ist von Objektorientierung außer einem gewissen Formalismus nichts vorhanden.

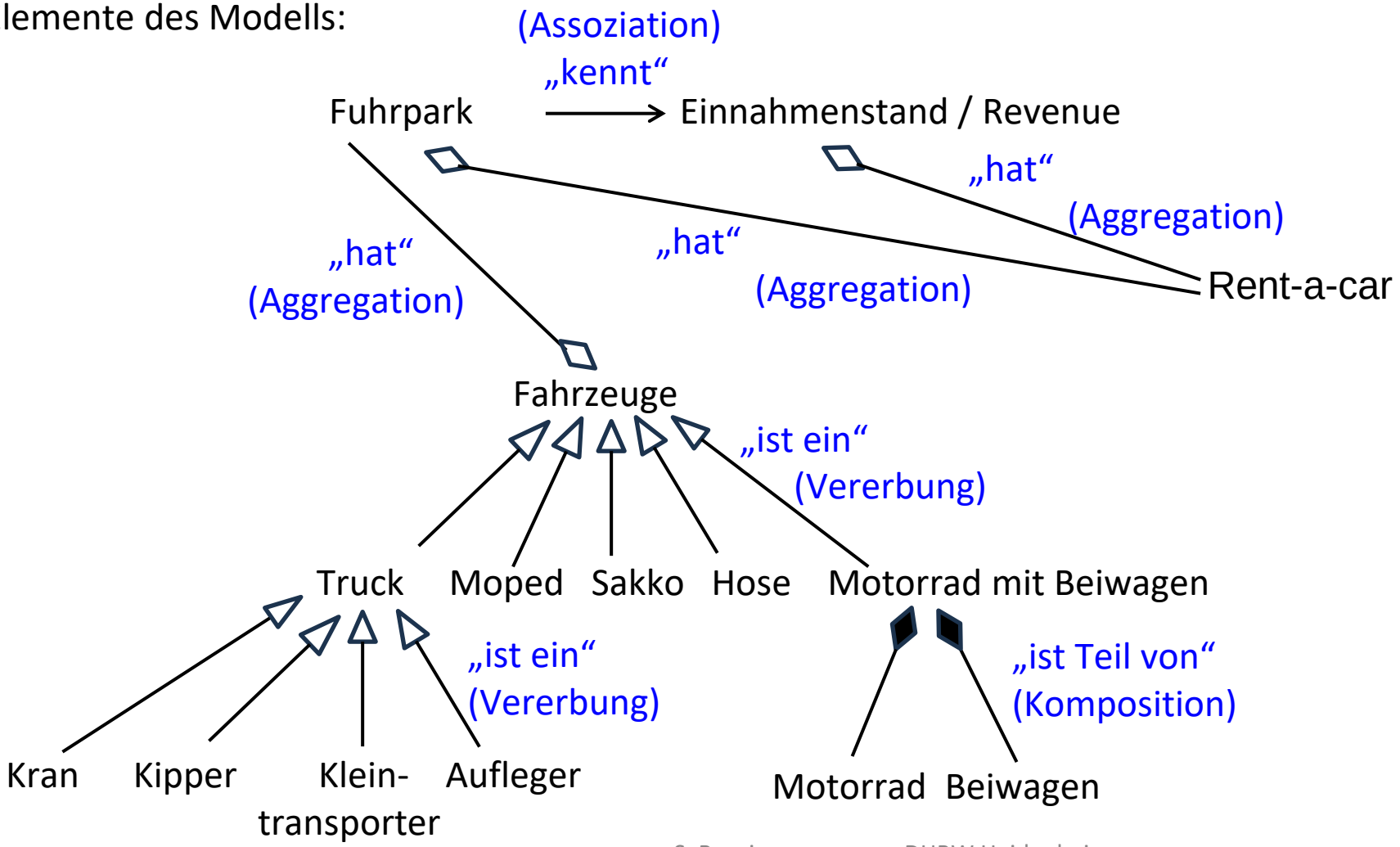
- Repräsentative C# - Programme eignen sich in der Regel wegen ihrer Länge und Komplexität (aus der Sicht des Anfängers) nicht für eine Detailanalyse.

-> Beispielprogramm, trotz angestrebter Einfachheit objektorientiert

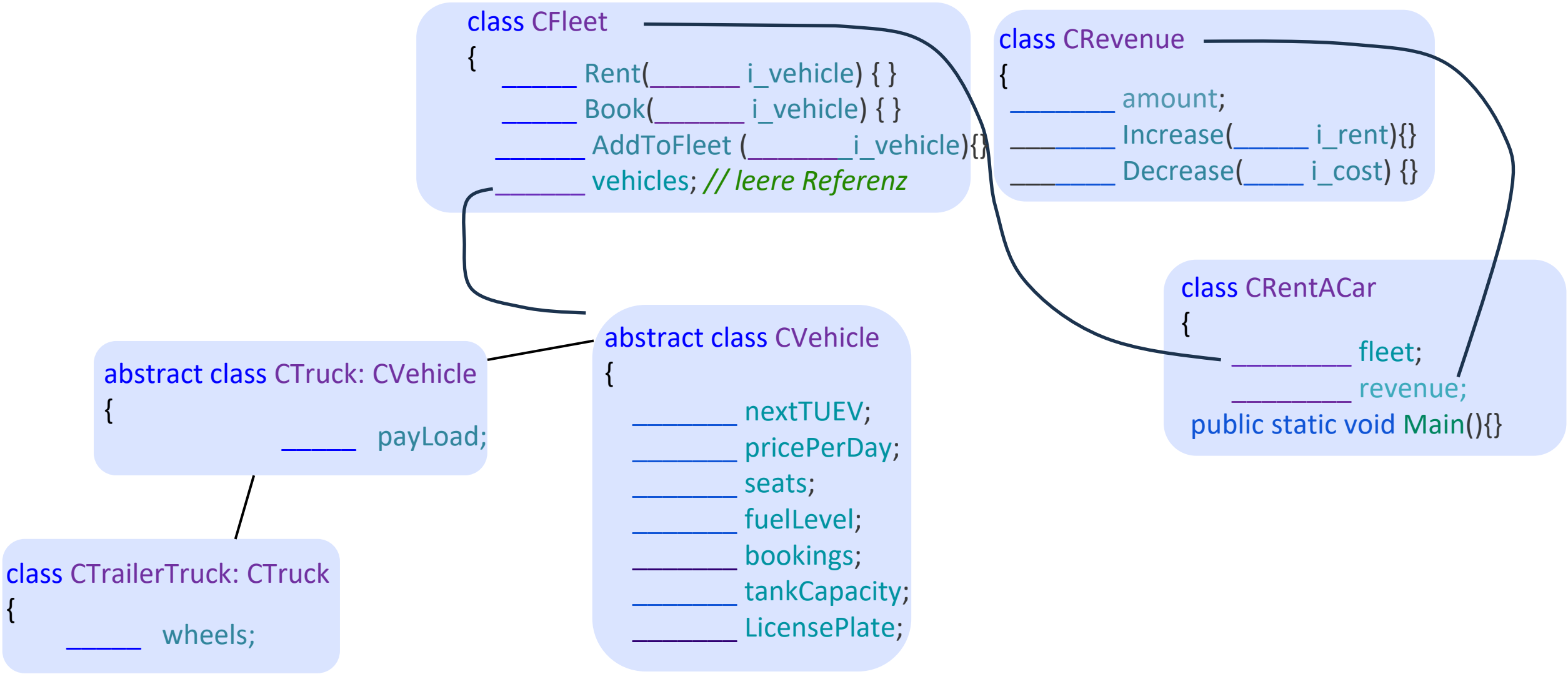
Ziel ist, einen Fahrzeugverleih abzubilden,
Fahrzeuge auch online anzubieten,
die Einnahmen und die Fahrzeuge zu verwalten.

Objektorientierung: Beispiel

Elemente des Modells:

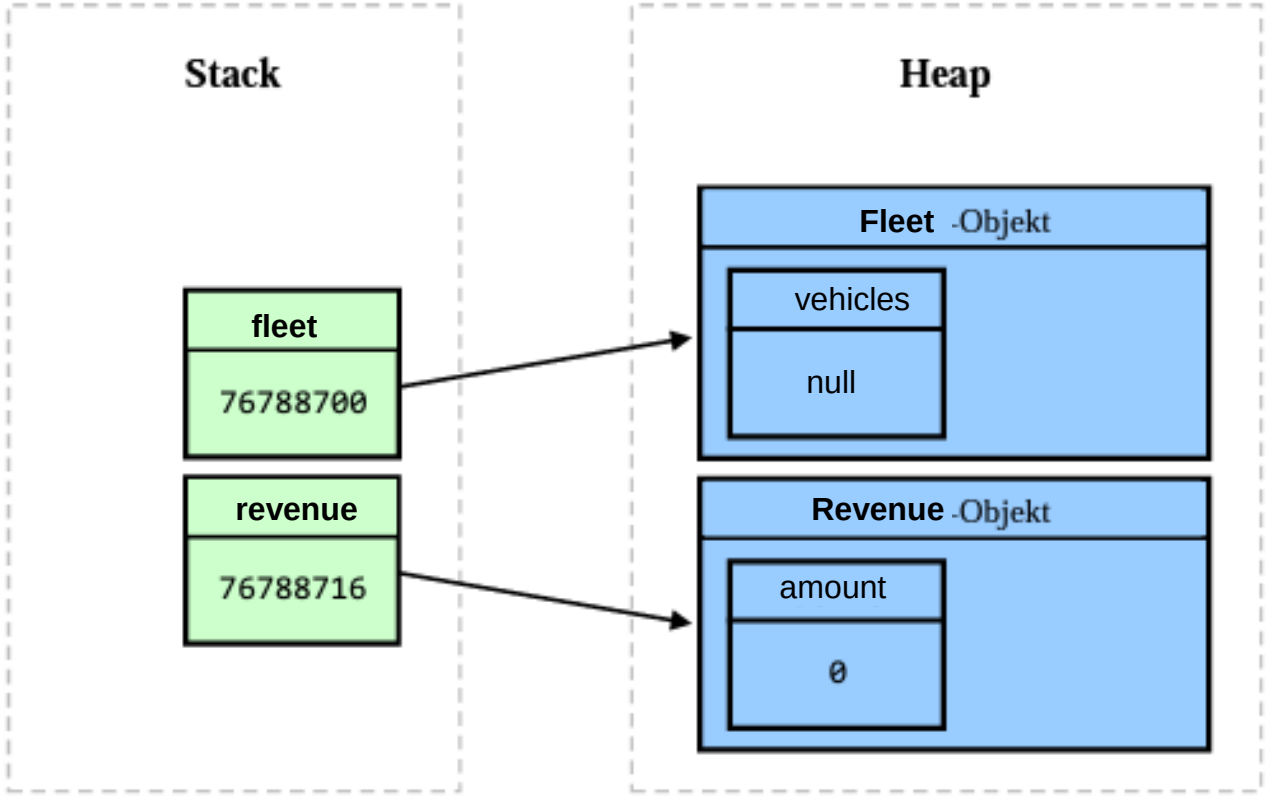


Objektorientierung: Beispiel



Klassifizierung von Variablen nach Zuordnung zu Objekt oder Klasse

```
class CRentACar
{
    CFleet fleet;
    CRevenue revenue;
    public static void Main()
    {
        fleet = new CFleet();
        revenue = new CRevenue();

        // ....
    }
```



Vorteile von OOP: Datenkapselung

Datenkapselung

- Methoden **kapseln direkte Zugriffe auf Klassendaten**
  (Bsp: der Preis darf für Anzeigezwecke abgefragt, aber nicht verändert werden)
- public Methoden bilden die **Schnittstelle** der Klasse nach außen
- Günstige Voraussetzungen für Test und Fehlerbeseitigung:
  Klassen können **an** ihren **public Schnittstellen getestet** werden,
  codierte Tests sind reproduzierbar nach Änderungen (Unit Testing)
- Innovationsoffenheit bei **gekapselten Implementierungsdetails**, ohne Kooperation mit anderen Klassen zu gefährden
- Produktivität durch wiederholt **bequem verwendbare** Klassen
- Erfolgreiche parallele Teamarbeit durch **abgeschottete Verantwortungsbereiche** und **Vorabdefinition von Interfaces**

Vorteile von OOP: Vererbung

Vererbung

- Ableitung spezialisierter Klassen aus einer Klasse zur Lösung neuer Aufgaben
- abgeleitete Klassen erben alle Member der Basisklasse
- Wiederverwendung von Software ohne lästiges und fehleranfälliges Kopieren von Quellcode
- Design der abgeleiteten Klasse kann sich auf neue Member beschränken oder bei manchen Erbstücken Modifikationen vornehmen
- Kontrakte der Basisklasse dürfen nicht gebrochen werden!

Open-Closed-Principle für Klassendesign:

Offen sein für Verwendung (auch als Basisklasse), geschlossen für Veränderung!

Meyer, Bertrand (1988). Object-Oriented Software Construction. Prentice Hall
Robert C. Martin "The Open-Closed Principle", *C++ Report*, January 1996

Vorteile von OOP: Polymorphie

Polymorphie (unterschiedliche Gestalt des Verhaltens)

- Objekte unterschiedlicher, von der gleichen Basisklasse abgeleiteten Klassen können bei **gleichem Methodenaufruf** ein unterschiedliches Verhalten zeigen (Methoden überschreiben)
  Klasse: CLebewesen, Methode Stoffwechsel(), abgeleitete Klassen CTier (organisch) und CPflanze (anorganisch) überschreiben Methode unterschiedlich
- dabei dürfen nur solche Methoden aufgerufen werden, die schon in der Basisklasse definiert sind
- Methode ist in abgeleiteten Klassen anders implementiert:
  jedes (per Basisklassenreferenz angesprochene Objekt) führt sein angepasstes Verhalten aus
- derselbe Methodenaufruf -> unterschiedliche (**polymorphe) Verhaltensweisen** zur Folge
- welche Methode (der Basis- oder abgeleiteten Klasse) ausgeführt wird, entscheidet sich erst zur Laufzeit (späte Bindung) über den Referenztyp
- lose Kopplung von Klassen ist möglich, Wiederverwendbarkeit von vorhandenem Code wird verbessert

Feldzugriff über Properties: „Syntactic sugar“

```
public class CVehicle
{
    float pricePerDay=10.0;  // private
    int seats=2;          // private
    public float GetPricePerDay()
    {
        return pricePerDay;
    }
    public void SetPricePerDay(int value)
    {
        if (value != 0)  pricePerDay = value;
    }
    // ....  dito für seats
```

```
public class CVehicle
{
    float pricePerDay;  // private
    int seats;     // private
    public float PricePerDay
    {
        get { return pricePerDay; }
        set { if (value != 0) pricePerDay = value; }
    } = 10.0;
    // .... dito für seats
```

```
public class CVehicle
{
    float pricePerDay;  // private
    public float PricePerDay
            {  get { return pricePerDay; }
               set { if (…) pricePerDay = value;}
            } = 10.0;

    public int Seats {get; set;}


    // „Automatische Implementierung“ für Seats
```

```
float myPricePerDay = vehicle.GetPricePerDay();
                      vehicle.SetPricePerDay(77.50);
```

```
float myPricePerDay = vehicle.PricePerDay;   // wird wie ein Feld benutzt
item.PricePerDay=77.50;    // „smart field“
```

Properties

Entweder get oder set darf fehlen:

```
class CVehicle
{
    int seats;
    public int Seats { get; ~~set;~~}
}

....
x = vehicle.Seats;     // o.k.
vehicle.Seats= ...;    // verboten
```

Init-only Property:

```
class CVehicle
{
    int seats;
    public int Seats { get; init; }
}

// Init-Methode: init { if (value != 0) seats = value; }

var seats = new Vehicle() { Seats = 4 };
```

Vorteile von Properties:

- read-only oder write-only Zugriff möglich
- Datenvalidierung beim Zugriff möglich
- wenig Schreibaufwand bei der Vereinbarung

- Property ist nach der Initialisierung *unveränderlich*

Übung daheim

Übungsblatt 1!

Übung 1

1. Warum steigt die Produktivität der Softwareentwicklung durch objektorientiertes Programmieren?

Übung

2. Welche von den folgenden Aussagen sind richtig bzw. falsch?

a) In C# kann man nur Software für Windows entwickeln.

b) Das .NET - Framework für Windows wurde in C# programmiert.

c) Unter den .NET - Programmiersprachen zeichnet sich C# durch eine besonders leistungsfähige Standardbibliothek aus.

d) Die Klassen in einem mit C# erstellten DLL-Assembly können auch in anderen .NET - Programmiersprachen (z. B. VB.NET) genutzt werden.

Übung

3. Welche Aufgaben erfüllt die Common Language Runtime (CLR)?

Übung

4. In welcher Beziehung stehen Assemblies und Namensräume?

Übung

5. Was bedeuten die Abkürzungen CIL?

Übung

5. Was bedeuten die Abkürzungen

FCL?

Übung

5. Was bedeuten die Abkürzungen

CLS?

Übung

5. Was bedeuten die Abkürzungen

COM?