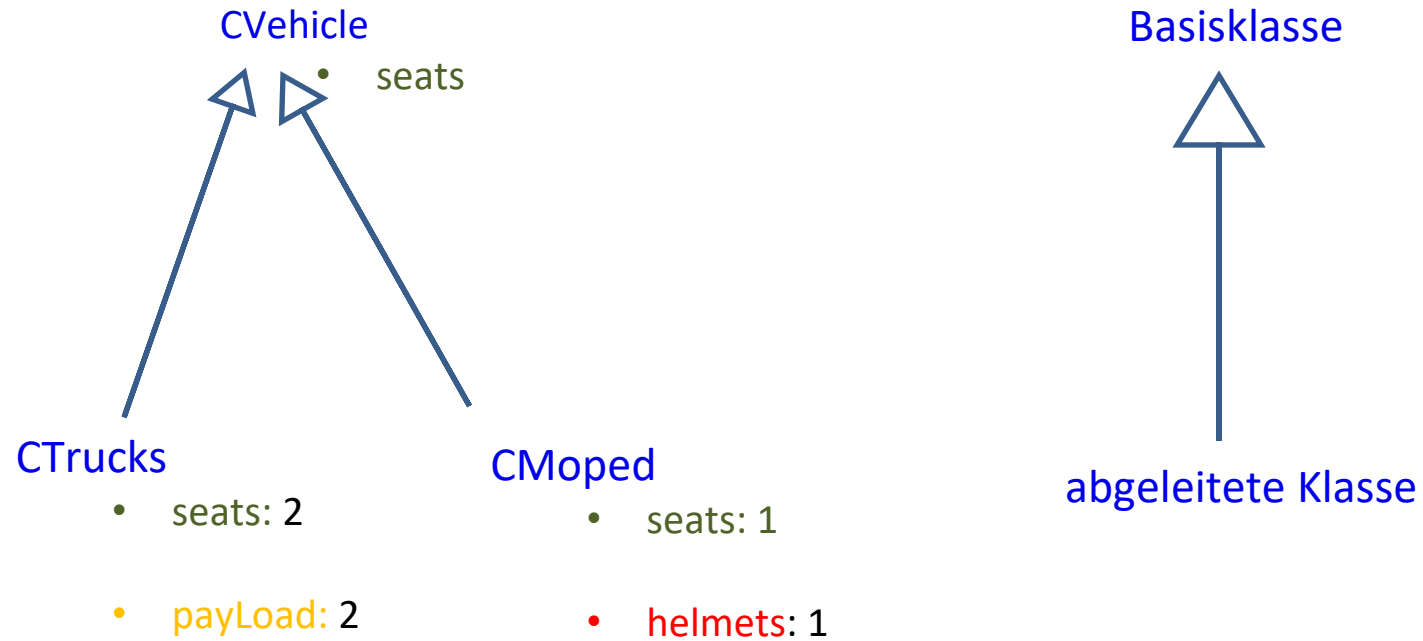


5. Klassen und Objekte



Vorteile von OOP




Spezialisierung über zusätzliche Felder

Generalisierung über Gemeinsamkeiten

Membervariablen - Felder

Automatische Initialisierung

Lokale Variable dürfen vor einer Initialisierung/ Wertzuweisung nicht verwendet (gelesen) werden (Compilerfehler)!  CS0165 Verwendung der nicht zugewiesenen lokalen Variablen...

Instanzvariablen (eines Objekts) erhalten automatisch Voreinstellungswerte:

Datentyp	Voreinstellungswert
sbyte, byte, short, ushort, int, uint, long, ulong	0
float, double, decimal	0.0
char	'\0'(Unicode-Zeichennummer)
bool	false
Referenztyp (inkl. string)	null



Innerhalb der Instanzmethoden einer Klasse:

Instanzvariablen des *aktuellen* (die Methode ausführenden) Objekts können direkt über ihren Namen angesprochen werden (siehe *price*, *size*).

Gelegentlich kann es sinnvoll oder erforderlich sein, einem Instanzvariablennamen über das Schlüsselwort **this** eine **Referenz auf das handelnde Objekt** voranzustellen (z.B. bei gleichnamigen lokalen Variablen), wobei das Schlüsselwort und der Variablenname durch den **Punktoperator** zu trennen sind.

```
private int seats; // =0
CTruck (int seats, /* ... */) // new CTruck (4, ...
{
    this.seats = seats; // this.seats=4;
}
```

Zugriff auf Methoden eines anderen Objekts: über . – Operator:

```
if (truck.Seats < 2) ...
```

Memberfunktionen - Methoden



In den Methoden eines Programms werden

- vordefinierte (z. B. der FCL-Standardbibliothek entstammende) oder
- selbst erstellte Klassen und Objekte zur Erledigung von Aufgaben verwendet.

Ein Programm besteht aus Klassen, die als Baupläne für Objekte und/oder als Akteure dienen.

Die Akteure (Objekt und Klassen) haben jeweils einen Zustand (abgelegt in Membervariablen).

Sie können Botschaften empfangen und senden

(heißt: eigene Methoden ausführen und die anderer Objekte/ Klassen aufrufen).

BTW: Headerdateien und forward declarations sind bei C# absolut unüblich!

Methoden: Parameter-Modifikatoren

Methodensyntax:

int Add (int a, int b)

Zusätzlich:

Parameter-Modifikatoren für Valuetypes,

int Add (<mod> int a, <mod> int b)

Übergabe der Referenz durch Laufzeitumgebung

Alle Parameter-Modifikatoren müssen bei Definition und beim Aufruf angegeben werden!

Methoden: Parameter-Modifikatoren

Arten:

- **ref-Parameter:** *int Add (ref int a, ref int b) // a,b lesbar und schreibbar*
ermöglicht den Informationstransfer in beide Richtungen, Parameter muss vor Aufruf initialisiert sein..
- **out-Parameter** *int Add (out int a, out int b) // a,b in Methode nur schreibbar*
aufgerufene Methode schreibt auf (verändert) Variable der rufenden Methode
(Parameter muss vor Aufruf nicht initialisiert sein, reiner Returnwert)
- **in-Parameter** *int Add (in int a, in int b) // a,b trotz Referenz nur lesbar*
Übergabe einer Referenz, aber Verhinderung eines Schreibzugriffs.
Parameter muss vor Aufruf initialisiert sein.
- **kein** Modifikator (Kopie des ValueTypes wird übergeben, kein Schreibzugriff in Methode)



Quellcode	Ausgabe
<pre>using System; class Prog { void Tausche(ref int a, ref int b) { int temp = a; a = b; b = temp; } static void Main() { Prog p = new Prog(); int x = 1, y = 2; Console.WriteLine("Vorher: x = {0}, y = {1}", x, y); p.Tausche(ref x, ref y); Console.WriteLine("Nachher: x = {0}, y = {1}", x, y); } }</pre>	<p>Vorher: x = 1, y = 2 Nachher: x = 2, y = 1</p>



out-Parameter

Programm liest für Aufrufer Werte von Konsole ein und übergibt diese in Out-Variable

Quellcode	Ausgabe (Eingaben fett)
<pre>using System; class Prog { void Lies(out int x, out int y) { Console.Write("x = "); x = Convert.ToInt32(Console.ReadLine()); Console.Write("\ny = "); y = Convert.ToInt32(Console.ReadLine()); } static void Main() { Prog p = new Prog(); int x, y; p.Lies(out x, out y); Console.WriteLine("\nx % y = " + (x % y)); } }</pre>	<pre>x = 29 y = 5 x % y = 4</pre>



out-Parameter

Aufrufname kann durch `_` ersetzt werden, wenn man den Wert nicht abholen möchte
(wird aber dennoch verändert!)

Quellcode	Ausgabe (Eingaben fett)
<pre>using System; class Prog { void Lies(out int x, out int y) { . . . } static void Main() { Prog p = new Prog(); int x; p.Lies(out x, out _); Console.WriteLine("\nx = " + x); } }</pre>	<pre>x = 4 y = 7 x = 4</pre>



in-Parameter

Für Übergabe komplexer, großer Parameter ohne Wertkopie (größer als eine Speicheradresse).

Variable muss initialisiert sein (wird gelesen), es findet keine automatische Typanpassung statt.



Serien-Modifikator params []

Array-Parameter, der an *letzter* Stelle der Parameterliste stehen und in der Definition durch das Schlüsselwort **params** gekennzeichnet werden muss

- erlaubt die Übergabe einer variablen Anzahl von Parametern gleichen Typs.
- keine Angabe von **params** bei Aufruf!

Quellcode	Ausgabe
<pre>using System; class Prog { void PrintSum(params double[] args) { double summe = 0.0; foreach (double arg in args) summe += arg; Console.WriteLine("Die Summe ist = " + summe); } static void Main() { Prog p = new Prog(); p.PrintSum(1.2, 1.0); p.PrintSum(1.2, 1.0, 3.6); } }</pre>	<pre>Die Summe ist = 2,2 Die Summe ist = 5,8</pre>



- **async**

Durch den Modifikator **async** wird eine Methode als *asynchron* deklariert(-> Multithreading)

- **unsafe**

Markierung unsicheren Codes (z. B. unter Verwendung von Zeigeroperationen oder mit eingeschränkter Portierbarkeit) -> kein Bestandteil der Vorlesung



Definition innerhalb einer ausführbaren Programmeinheit (z. B. Methode).
Keine Ausnahme, sondern Normalfall!

Sinn:

lokal benötigte Hilfsmethoden, stärkere Modularisierung, mehrfacher Aufruf aus äußerer Methode
(aber nur aus dieser) möglich



Lokale Methoden

LiesZwei() verwendet Lies():

Quellcode	Ausgabe (Eingaben fett)
<pre>using System; class CProgram { void LiesZwei(out int z, out int n) { int Lies(string name) { Console.Write(name + " = "); return Convert.ToInt32(Console.ReadLine()); } z = Lies("Erstes Argument"); n = Lies("Zweites Argument"); } static void Main(string[] args) { p.LiesZwei(out x, out y); Console.WriteLine("\nx % y = " + (x%y)); } }</pre>	<pre>Erstes Argument = 28 Zweites Argument = 5 x % y = 3</pre>

Benannte Parameter (named arguments)

Angabe des Parameternamens beim Aufruf vor dem Wert

Vorteile:

- Stellung der Parameter muss nicht bekannt sein und übereinstimmen
- Lesbarkeit des Codes wird deutlich erhöht

Beispiel: *int Divide (int divisor, int dividend, out int remainder, bool setRem)*

```
b1.Divide(9, 3, rem, false); // Stellungsparameter
```

lässt sich äquivalent auch so formulieren:

```
b1.Divide(dividend: 3, divisor: 9, remainder: rem, setRem: false); //
```

oder

```
b1.Divide(9, 3, rem, setRem:false); // oder  
b1.Divide(divisor: 9, 3, rem, false);
```



Optionale Parameter

Formalparameter mit Voreinstellungswert, Angabe kann bei Aufruf weggelassen werden (von hinten)

Regeln:

- Als Voreinstellungswert ist ein konstanter Ausdruck erlaubt (Wert steht schon zur Übersetzungszeit fest:

```
void Opt (int i = Int32.MaxValue - 1)
{
    ...
}
```

- Auf einen optionalen Formalparameter darf kein obligatorischer mehr folgen, nur weitere optionale.

Beispiel:

```
public int Divide (int divisor, int dividend, out int remainder, bool setRem=false)
{
    ...
}
```

Aufruf z.B.: b1.Divide(9, 3, rem);

Default bei fehlender Angabe: **private**

Explizit angebbar (möglichst erst nach allen private Membern):

- **public (Schnittstelle):** Methode oder Attribut steht allen Klassen und Objekten zur Verfügung
- **private:** Zugriff haben nur:
Member dieser Klasse.
- **protected:** Zugriff haben nur:
Member dieser Klasse und alle von dieser Klasse abgeleiteten Klassen (auch in anderen Assemblies).
- **private AND protected (protected --):**
Zugriff haben nur:
Member dieser Klasse und alle von dieser Klasse abgeleiteten Klassen in derselben Assembly.

Fortsetzung:

- **internal:** Zugriff haben nur:
Member aller Klassen der aktuellen Assembly (.exe oder .dll). Unit Tests nicht!
- **protected OR internal** (internal ++):
Zugriff haben nur:
Member dieser Klasse, alle Klassen innerhalb der gleichen Assembly
und alle von dieser Klasse abgeleiteten Klassen anderer Assemblies
(wie internal + abgeleitete Klassen in anderen Assemblies).




Modifikator(en)	Der Zugriff ist erlaubt für ...				
	eigene Klasse (Abk. K) u. innere Klassen	nicht von K abgel. Klassen im eigenen Assembly	von K abgeleitete Klassen		sonstige Klassen
			im eigenen Assembly	in anderen Assemblies	
<i>ohne</i> oder private	ja	nein	nein	nein	nein
internal	ja	ja	ja	nein	nein
protected	ja	nein	geerbte Member	geerbte Member	nein
protected internal	ja	ja	ja	geerbte Member	nein
private protected	ja	nein	geerbte Member	nein	nein
public	ja	ja	ja	ja	ja



Signatur einer Methode

<Return-Typ> Methodenname (<ParamTyp 1> , <ParamTyp2>, ... <ParamTyp n>)



Signatur

zur Signatur zählen nicht:

- Typ des Returnwerts (Grund: wird vom Aufrufer häufig ignoriert – dann kann die Methode nicht ausgewählt werden)!
- params-Modifier

Methoden überladen (Overloading)

Gleichnamige Methoden sind erlaubt, solange die Signatur unterschiedlich ist.

Besitzt eine Klasse mehrere Methoden mit demselben Namen, liegt eine sogenannte *Überladung* von Methoden vor, wenn sich die **Signaturen** der beteiligten Methoden **unterscheiden**.

Signaturen unterscheiden sich NICHT, wenn:

- Die Namen der Methoden sind identisch und
- Die Formalparameterlisten sind gleich lang und
- Positionsgleiche Parameter stimmen hinsichtlich Datentyp und Parameterart (Wert-, **ref**-, **in**- bzw. **out**-Parameter) überein.

Nicht verwechseln mit:

- Überschreiben (override, später): Methoden *abgeleiteter Klassen* haben bei gleicher Signatur unterschiedliche Implementierungen



eine Klasse, **überladen** mit gleichnamigen Methoden

Methoden überladen

Vorteile überladener Methoden mit unterschiedlichen Parametertypen:

- Berechnung des Betrags einer Zahl in math.h, unterschiedlich codiert:

```
public static float  Abs(decimal value)  
public static double Abs(double value)  
public static long   Abs(long value)
```

Überladen von Methoden mit optionalen Parametern:

- wenn eine Überladung mit einer kürzeren Parameterliste existiert, wobei mindestens ein optionaler Parameter fehlt, dann wird bei einem Aufruf mit der kürzeren Aktualparameterliste die Methode *ohne* optionale Parameter aufgerufen (kürzester Match gegen eine Deklaration).

In dieser Konstellation ist die Definition von Voreinstellungswerten wirkungslos (s. nächste Folie).



Quellcode	Ausgabe
<pre>using System; class Prog { void Test(int i = 13) { Console.WriteLine("Opt. Par. = " + i); } void Test() { Console.WriteLine("Ohne Parameter"); } static void Main() { Prog p = new Prog(); p.Test(); } }</pre>	Ohne Parameter

Objekte

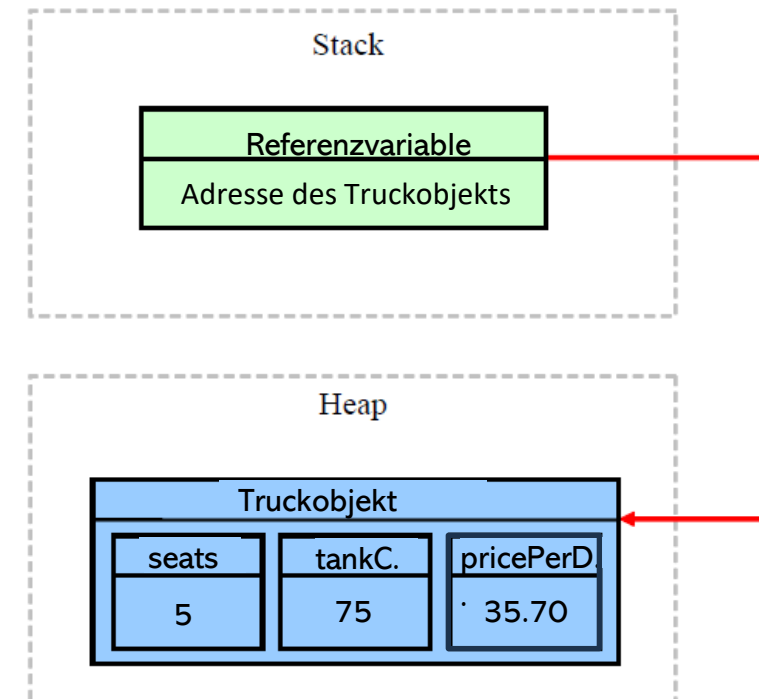


Objekte

```
CTruck truck;    // truck: Referenzvariable, kann auf Objekte der Klasse CTruck und Objekte aller abgeleiteten  
                // Klassen zeigen (noch ohne Wert), automatisch initialisiert auf null
```

```
truck = new CTruck (seats=5, pricePerDay=35.70f, tankCapacity=75);
```

```
// Anlegen eines Objekts durch Aufruf der  
// Konstruktormethode CTruck::CTruck(int, float, float)  
// Zuweisung der Referenz des Objekts an Referenz truck
```



Objekte erzeugen - Konstruktoren

Konstruktoren: spezielle Methoden für das Erzeugen/ Initialisieren neuer Objekte

Aufgaben:

- Instanzvariablen des Objekts initialisieren und/oder
- andere Arbeiten verrichten (z. B. Öffnen einer Datei oder Netzwerkverbindung)

Ziel:

- ein neues Objekt in einen validen Zustand bringen und für seinen Einsatz vorbereiten

Aufruf:

- **new**-Operator, Konstruktormethode der gewünschten Klasse als Operand



Objekte initialisieren – abgeleitete Klassen

Schlüsselwort *base*:

- Ruft einen Konstruktor der nächsthöheren Basisklasse (hier: CVehicle) auf:

```
public class CTruck: CVehicle  
{  
    public CTruck (int seats, float payLoad , float pricePerDay): base (pricePerDay , seats)  
    {  
    }
```

- Fehlt der explizite Aufruf: parameterloser Standardkonstruktor der Basisklasse wird aufgerufen

-> hat die Basisklasse aber einen parameterbehafteten Konstruktor, muss sie den parameterlosen selbst definieren!



Objekte initialisieren - Konstruktoren

```
public class CTruck: CVehicle
{
    public CTruck (int seats, float payLoad , float pricePerDay): base (pricePerDay , seats)
    {
        this.payLoad = payLoad;
        // ...
    }
}
```

oder:

```
public class CTruck: CVehicle
{
    public CTruck (int seats, float payLoad , float pricePerDay) // CVehicle braucht impliziten oder expliziten parameterlosen c`tor
    {
        this.seats = seats; // Member von CVehicle
        this.payLoad = payLoad; // Member von CTruck
        this.pricePerDay = pricePerDay; // Member von CVehicle
        // ...
    }
}
```

Objekte initialisieren - Konstruktoren

Abarbeitungsreihenfolge (jeder Konstruktor ruft auf):

- die null-Initialisierung der eigenen Instanzvariablen
- explizite Feldinitialisierer der eigenen Klassen und der Basisklassen werden ausgeführt
- dann
 - (implizit) den parameterlosen Standard-/Default-Konstruktor seiner unmittelbaren Basisklasse (inkl. null-Initialisierung der geerbten Member), der ruft wiederum zunächst den Konstruktor seiner Basisklasse u.s.w.
 - oder explizit über *base* einen parameterbehafteten Basisklassenkonstruktor (und s.o.)
- Erst dann wird der Anweisungsteil des Konstruktors selbst ausgeführt.

(Compilerfehler, wenn parameterloser Konstruktor in base fehlt, es aber einen mit Parametern gibt)



Objekte initialisieren - Konstruktoren

Syntax:

- Methodenname entspricht Klassenname
- kein Rückgabewert
- Parameterliste möglich
- Es sind beliebig viele überladene Konstruktoren mit demselben Namen und unterschiedlicher Parameterliste möglich
- müssen (außer Standardkonstruktor) explizit `public` gesetzt werden, wenn gewünscht
- Konstruktoren können i.d.R. nicht direkt aufgerufen, sondern nur als Argument des **new**-Operators verwendet werden

Objekte initialisieren – Konstruktor-Arten

Vorhandensein:

- a) es ist kein Konstruktor definiert (und nur dann):

C# legt public Standard-Konstruktor (.ctor) ohne Parameter an,

Aufruf: `CRevenue rev = new CRevenue();`

oder: `var rev = new CRevenue();`

- b) mindestens ein Konstruktor mit Parameterliste ist definiert:

-> parameterloser Konstruktor muss gegebenenfalls ebenfalls explizit definiert werden
(Standard-Konstruktor wird nicht mehr automatisch angelegt)

- c) nur eigener parameterloser Konstruktor, z.B. `public CRevenue() { amount=0.0;}`



Konstruktor: Form und Aufrufe

Expression body:

Der Konstruktor (oder beliebige andere Methode/ Property) enthält nur 1 Statement:

-> ‚Expression body‘ ist möglich:

```
public CRevenue () => amount = 0.0; // Lambda operator =>
```

Anderen eigenen Konstruktor aufrufen:

- nur möglich über das Schlüsselwort `this` (statt `base`) zwischen Parameterliste und Implementierung:

```
public CTruck (int seats, float tankCapacity , float pricePerDay)  
: this (int seats)  
{...}
```

a) Statischer Konstruktor:

- Initialisiert statische Klassenvariable oder statische Klassen.
- nicht-statischer Konstruktor ist bei statischen Klassen nicht erlaubt (auch kein Standardkonstruktor)
- wird automatisch von Laufzeitumgebung aufgerufen

b) privater Konstruktor:

- verhindert das automatische Anlegen des public Standard-Konstruktors durch die Laufzeitumgebung, vor allem bei Klassen mit rein statischen Members. Dort möchte man das Instanzieren ja vermeiden!
- sind alle Member einer Klasse statisch: besser die ganze Klasse statisch machen, und dann siehe a)

c) Zieltypisierter Aufruf (ab C# 9.0):

- bei Konstruktoraufruf mit new kann die doppelte Nennung des Klassennamens vermieden werden:
`CRevenue rev = new CRevenue();` lässt sich vereinfachen zu: `CRevenue rev = new();`
oder: `var rev = new CRevenue();`

Es muss kein parametrisierter Konstruktor benutzt werden, um Member zu initialisieren.

→ Verwendung der Properties in einer Initialisierungsliste

Dafür wird automatisch der parameterlose default-Konstruktor aufgerufen

```
using System;
```

```
class CRentACar
```

```
{  
    private CFleet fleet;  
    private CRevenue revenue;  
  
    public CRentACar ()  
    {  
        fleet= new CFleet ();  
        revenue= new CRevenue (500); // Anfangskapital, Konstruktor mit Parameter notwendig  
        // alternativ:    revenue= new (500); // Konstruktor mit Parameter notwendig  
        // alternativ:    revenue = new CRevenue {Amount=500}; // keine Konstruktordefinition notwendig, aber Setter  
        // nicht möglich: revenue = new {Amount=500};  
    }  
}
```

Abräumen von Objekten durch Garbage collection

„Garbage collector“ = „Müllsammler“

Vorteile des vollautomatischen Garbage Collectors für Daten/ Objekte auf dem Heap:

- vermeidet lästigen Aufwand
- weil der Programmierer keine Verpflichtung (und Berechtigung) zum Entsorgen von Objekten hat, kann es nicht zu Programmabstürzen durch Zugriff auf voreilig vernichtete Objekte kommen.
- es entstehen keine **Speicherlöcher** (engl.: *memory leaks*) durch versäumte Speicherfreigaben bei überflüssig gewordenen, nicht mehr erreichbaren Objekten