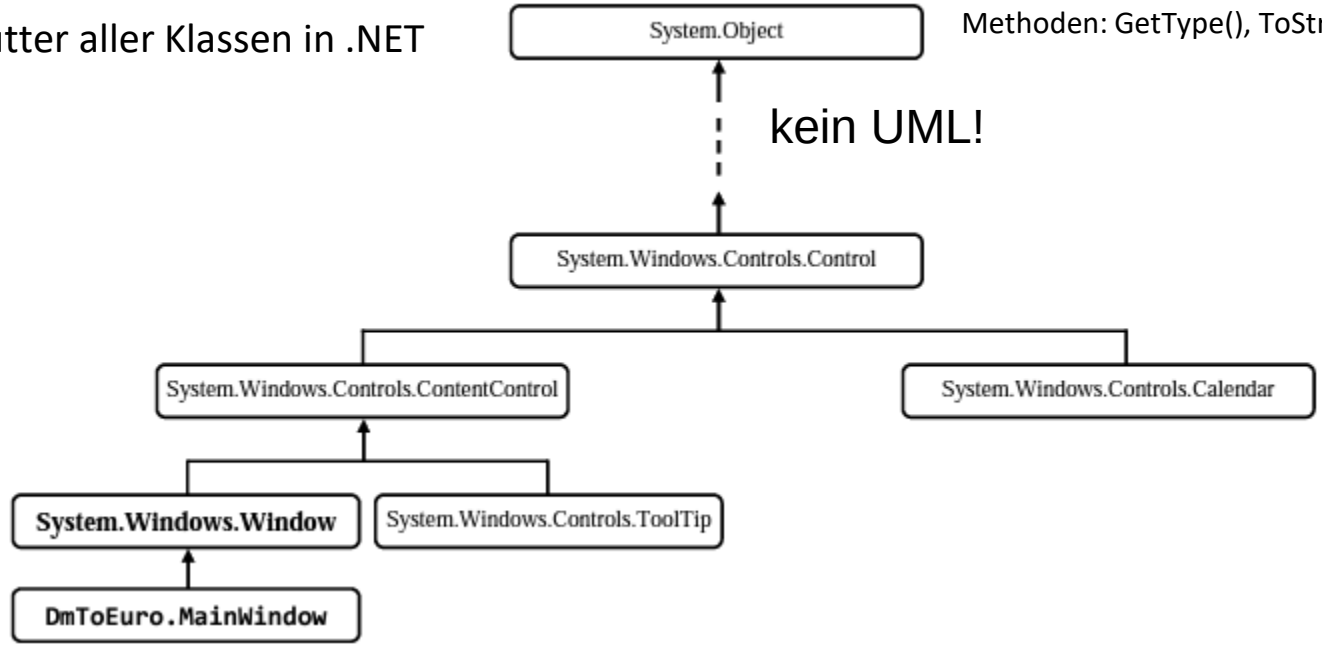
DHBW
Duale Hochschule
Baden-Württemberg
Heidenheim

Vererbung: Details und Varianten

Vererbung in Windows

Die Basisklasse Window ist Bestandteil eines komplexen Stammbaums. Ein kleiner Ausschnitt:

Mutter aller Klassen in .NET



Methoden: GetType(), ToString(), ...

Vererbung von System.object

Quellcode	Ausgabe

namespace CRentACar
{
public class RentACar
{
static void Main()
{
var b=new CTruck(...);
Console.WriteLine(b.GetType());
	CRentACar:CTruck GetType()
	CTruck GetType().Name

Methodenaufruf, gerichtet an die Klasse **Console**

Referenz auf ein CZiehung-Objekt

Eigenschaftszugriff, gerichtet an das von **GetType()** gelieferte **Type**-Objekt, liefert ein **String**-Objekt

```
Console.WriteLine(b.GetType().Name);
```

Methodenaufruf, gerichtet an das CDress-Objekt b, mit einer Referenz auf ein Type-Objekt als Rückgabewert

Klasse CTruck erbt indirekt von Klasse object, und damit auch die Methode GetType()

Vererbung von System.object

Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  var o = new Object();`
`  var s = "abc";`
`  int i = 13;`
`  Console.WriteLine(o.GetType() + "\n" +`
`  s.GetType() + "\n" +`
`  i.GetType());`
`  }`
`}`
	`System.Object`
	`System.String`
	`System.Int32`

GetType() liefert dynamischen Laufzeittyp

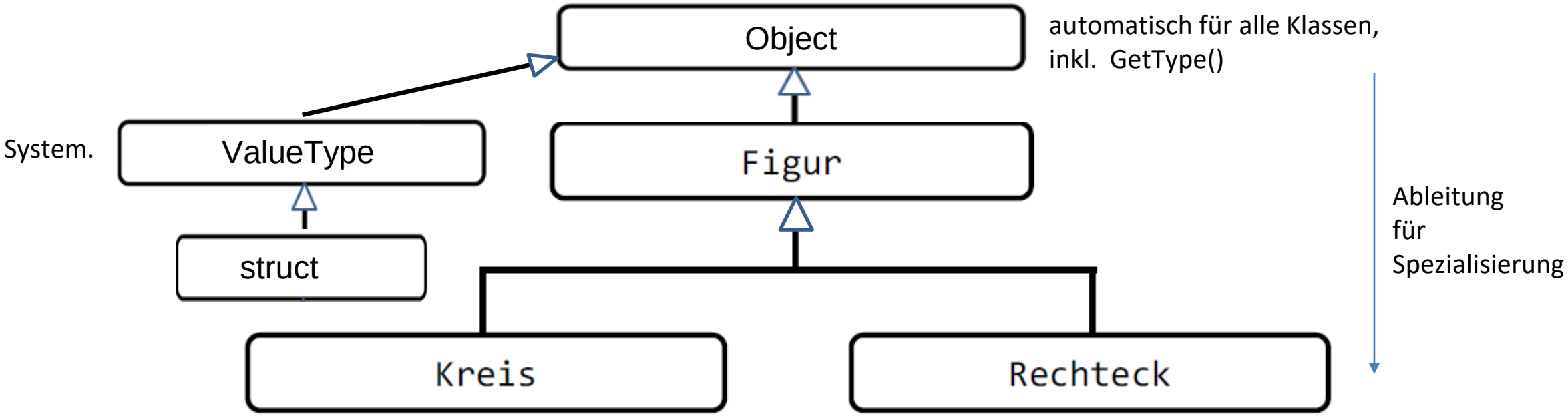
```
using System;
class Program
{
        static void Main()
        {       stat. Typ                                          dynamische Typen
                object[] oar = new object[3]    { new object(),   "123",   13 };   // 3 verschiedene Typen, die oar[i] referenziert!
                 foreach(var o in oar)
                     Console.WriteLine($"{o, 15} hat den Laufzeittyp {o.GetType()}");
        }
}
```

produziert die Ausgabe:

```
System.Object hat den Laufzeittyp System.Object
          123 hat den Laufzeittyp System.String
           13 hat den Laufzeittyp System.Int32
```

Beispiel

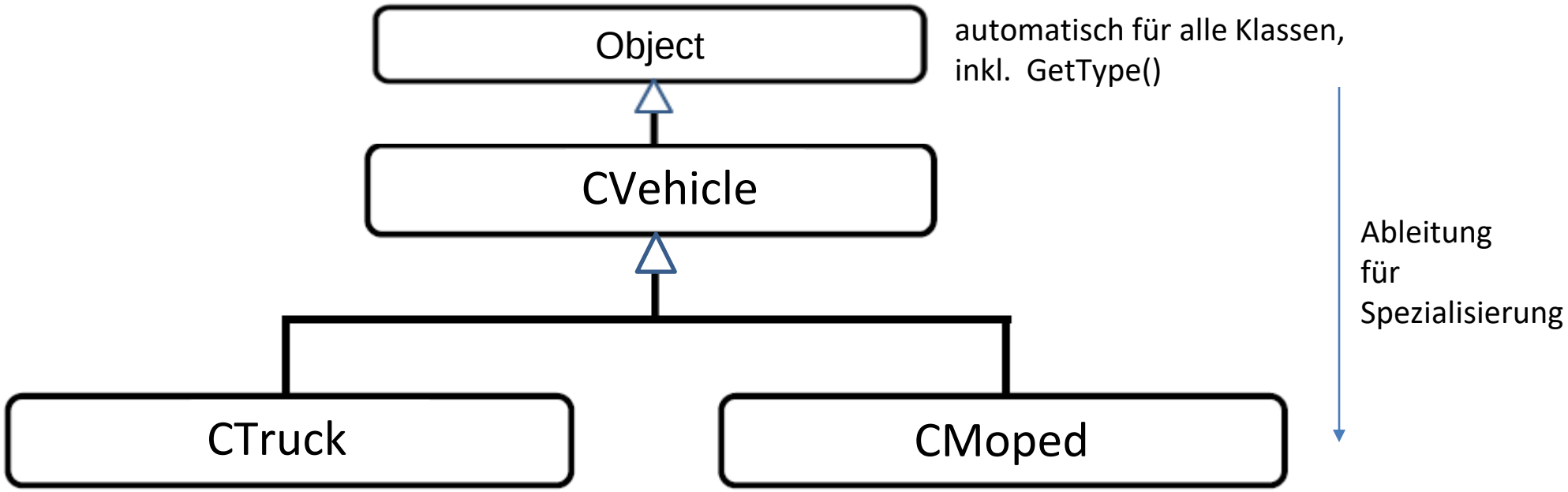
Vererbung: Man geht von der **allgemeinsten** Klasse aus und leitet durch **Spezialisierung** neue Klassen ab, nach Bedarf in beliebig vielen Stufen.



Abgeleitete Klasse kann:

- zusätzliche Felder deklarieren
- zusätzliche Methoden oder Eigenschaften definieren
- geerbte Methoden ersetzen/ überschreiben, d.h. unter Beibehaltung der Signatur umgestalten

Beispiel

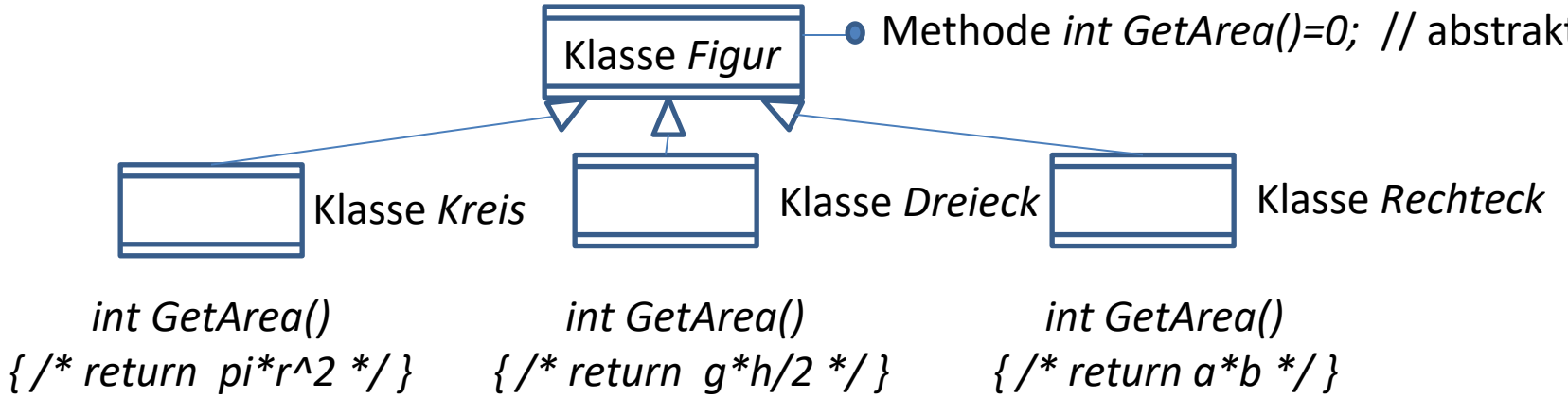

Object
automatisch für alle Klassen,
inkl. GetType()
CVehicle
Ableitung
für
Spezialisierung
CTruck
CMoped

Polymorphie

Open-Closed - Prinzip (Robert C. Martin): http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Klassen sollen **offen** sein für Verwendung, aber
geschlossen für Veränderung

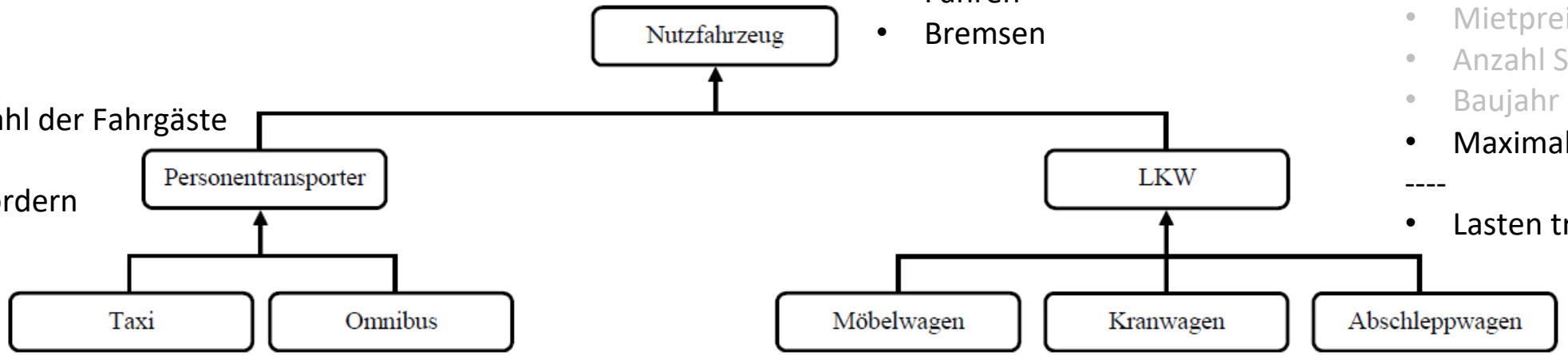
Erweiterungen bei Verwendung über Ableitung neuer Klassen + Polymorphie (Überschreiben) + Erweiterung oder Nutzung aus neuen Klassen



Vererbung

Merkmale aller Nutzfahrzeuge:
- Mietpreis
- Anzahl Sitze
- Baujahr

Methoden:
- Fahren
- Bremsen



- Mietpreis
- Anzahl Sitze
- Baujahr
- maximale Anzahl der Fahrgäste

- Fahrgäste befördern
- Fahren
- Bremsen

- Mietpreis
- Anzahl Sitze
- Baujahr
- Maximale Anhängelast

- Lasten transportieren
- Fahren
- Bremsen

C# erlaubt keine Mehrfachvererbung! (... ausser von Interfaces...erben von mehreren Interfaces erlaubt)

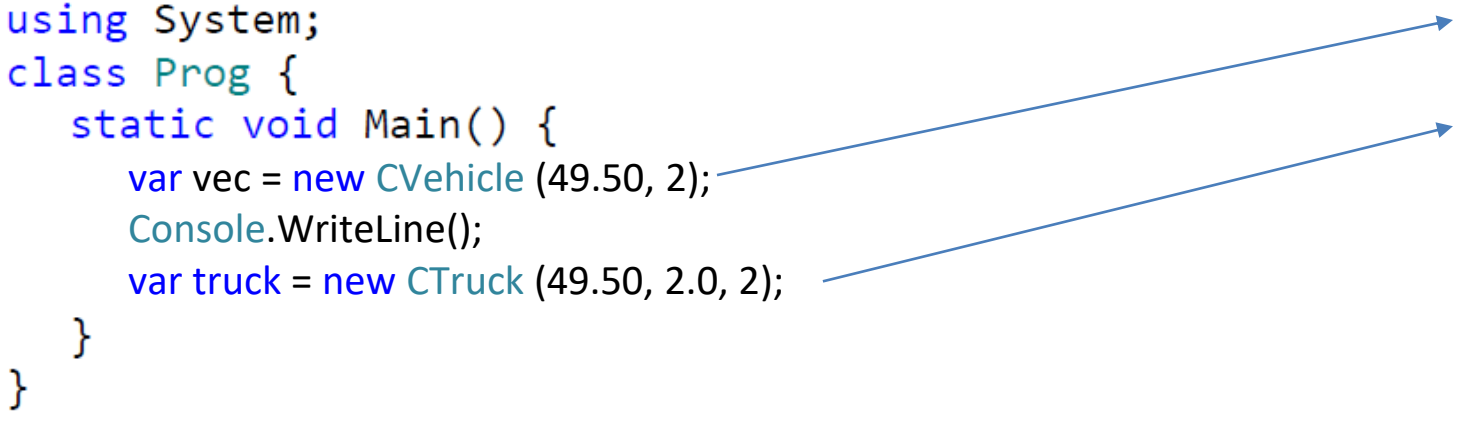
Beispiel RentACar

```
using System;

public class CVehicle // Basisklasse CVehicle
{
    protected int seats=4;                    // Instanzvariable für die Anzahl der Sitze
    protected float pricePerDay=75.5;         // Instanzvariable für den Tagesmietpreis

    public CVehicle (float pricePerDay, int i_seats)              // Konstruktor
    {
      if (i_pricePerDay > 20 && i_seats >= 1) { pricePerDay=i_pricePerDay; seats=i_seats; }
      Console.WriteLine(„CVehicle-Konstruktor");
    }

    public CVehicle() { }                     // parameterloser Konstruktor

    public long? NextTUEV()                   // Methode zur Berechnung des nächsten TÜV-Termins in ticks
    {
      // (Erstzulassung + 3 Jahre) > now ?  (Erstzulassung+3 Jahre) : (LastTUEV + 2 Jahre)
    }
}
```

Beispiel RentACar

```
using System;
public class CTruck : CVehicle // Klasse Truck als Spezialisierung des Fahrzeugs
{
  float payLoad = 2;                    // neues Feld in Klasse Ctruck
  public CTruck (int seats, float payLoad , float pricePerDay): base (pricePerDay , seats) // Konstruktor inkl. explizitem Aufruf des
  {                                                                                        // Basisklassen- Konstruktors
    if (i_payLoad >= 1) payLoad=i_payLoad;                                                 // mit Initialisierung
    Console.WriteLine(„CTruck-Konstruktor");
  }
  public CTruck() { }                                  // parameterloser Konstruktor

}
```

Konstruktoren: Ausführreihenfolge

Quellcode	Ausgabe
	CVehicle-Konstruktor CVehicle-Konstruktor CTruck-Konstruktor

```
using System;
class Prog {
   static void Main() {
      var vec = new CVehicle (49.50, 2);
      Console.WriteLine();
      var truck = new CTruck (49.50, 2.0, 2);
   }
}
```

Object-Konstruktor vor CVehicle-Konstruktor
nur deshalb nicht sichtbar,
da er keine Ausgabe macht

Geerbte Methoden, Properties und Indexer verdecken (overwrite)...

... in einer **abgeleiteten Klasse** durch eine Methode mit **gleicher** Signatur

(Überladung: gleiche Klasse, gleicher Name, andere Signatur!)

Bsp.:
In der C*Moped*-Klasse steht die von *CVehicle* geerbte Methode *GetNextTUEV()* zur Verfügung (Moped bis 50 ccm muß aber nicht zum TUEV)

-> Bessere Variante für *CMoped* :

```
using System;
public class CMoped : CVehicle
{
   . . .
   public new long? GetNextTUEV() // explizites new blendet Warnung bzgl. versehentlichem Verdecken aus
   {
     return null;
   }
}
```

Tips zum Verdecken

- auch **statische Methoden** können verdeckt, aber jederzeit über ihren Klassennamen wieder angesprochen werden

  Basisklasse.Methode() statt *Methode()*

- statische und Instanz**variable** von Basisklassen können auch verdeckt werden (in Ausnahmefällen), aber Vorsicht:

  die Methoden der Basisklassen verwenden nach wie vor die originalen Instanzvariablen!

-

- bei geplantem Verdecken immer Modifikator ***new*** angeben (sonst Warnung)

Typen einer Referenz

Jede Referenz hat 2 Typen:

- statisch: der Typ ihrer Deklaration

```
CVehicle  i = new CTruck(); // statischer Typ: CVehicle
                            // dynamischer Typ: CTruck
```

- dynamisch: der Typ des Objekts,
  auf das die Referenz gerade zeigt

```
i = new CMoped();  // dynamischer Typ: CMoped
```

- GetType() liefert: dynamischen Typ!

Nutzung des **statischen** Typs von Referenzen

```
public class CVehicle // Basisklasse
{
    public float PricePerDay {get; set; }
    public int Seats {get; }
    public DateTime FirstRegistration {get; }

    public CVehicle(float i_pricePerDay, int i_seats)

    {
        PricePerDay = i_pricePerDay;
        Seats=i_seats;
    }

    public CVehicle() { }

    public long? GetNextTUEV()  // overwrite möglich

    {
        Console.WriteLine(„CVehicle“);
        DateTime next= FirstRegistration.AddYear(3 );
       return ( (next- DateTime.Now).Days>0 ? next :
             LastTUEV.AddYear (2)) ;     }
}
```

```
using System;
public class CMoped : CVehicle
{
   int helmet   = 1;
   public CMoped (int h, float x, int y) : base(x, y)
   {
      helmet=h;
   }

   public CMoped() { }


   public Helmet              // Property
   {
     get {return helmet;}
     set { helmet = value;}
   }
   public new long? GetNextTUEV() // overwrite
   {
         Console.WriteLine(„CMoped“);
         return null;
    }
}
```

```
using System;
public static void Main()
{
    CVehicle ref1 = new CVehicle();

    CMoped  ref2 = new CMoped();

    ref1.GetNextTUEV();  // „CVehicle“

    ref2. GetNextTUEV();  // „CMoped“

    ref1=ref2;  // ref1 zeigt auf Moped

    ref1.GetNextTUEV(); // immer noch „CVehicle“
        // Methoden sind nicht virtual
        //  statischer Typ der Referenz genutzt
}
```

Nutzung des **dynamischen** Typs von Referenzen

```
public class CVehicle // Basisklasse
{
    public float PricePerDay {get; set; }
    public int Seats {get; }
    public DateTime FirstRegistration {get; }

    public CVehicle(float i_pricePerDay, int i_seats)

    {
        PricePerDay = i_pricePerDay;
        Seats=i_seats;
    }

    public CVehicle() { }

    public virtual long? GetNextTUEV()   // override möglich

    {
        Console.WriteLine(„CVehicle“);
        DateTime next= FirstRegistration.AddYear(3 );
       return ( (next- DateTime.Now).Days>0 ? next :
            LastTUEV.AddYear (2)) ;
    }
}
```

```
using System;
public class CMoped : CVehicle
{
   int helmet   = 1;
   public CMoped (int h, float x, int y) : base(x, y)
   {
      helmet=h;
   }

   public CMoped() { }


   public Helmet              // Property
   {
     get {return helmet;}
     set { helmet = value;}
   }
   public override long? GetNextTUEV() // override
   {
         Console.WriteLine(„CMoped“);
         return null;
    }
}
```

```
using System;
public static void Main()
{
    CVehicle ref1 = new CVehicle();

    CMoped  ref2 = new CMoped();

    ref1.GetNextTUEV();  // „CVehicle“

    ref2. GetNextTUEV();  // „CMoped“

    ref1=ref2;  // ref1 zeigt auf Moped

    ref1.GetNextTUEV(); // jetzt „CMoped“
        //  dynamischer Typ der Referenz genutzt
}
```

Überschreiben von Property-Methoden

```
public class CVehicle
{
   // ...

   public float PricePerDay
   {
      get;
      set;
   }
   protected int seats;
   public virtual int Seats
   {
      get;
      set;
   }
}
```

```
public class CTruck: CVehicle
{
   public override int Seats
   {
      get { return seats; }
      set { if ((value==1) && (value ==2))
                  seats=value;
          }
   }
   public float Payload
   {
      get;
      set;
   }
}
```

```
public class CMoped: CVehicle
{
   public override int Seats
   {
      get { return seats; }
      set { if (value==1)
                  seats=value;
          }
   }
   public int Helmet
   {
      get;
      set;
   }
}
```

Overload – Override – Overwrite

Aktion	Schlüsselwort Base class	Schlüsselwort Derived class	Signatur	Referenztyp	OO Polymorphismus
Override/ Überschreiben	virtual oder abstract	override	gleich	dynamisch	ja
Overwrite/ Verdecken	(virtual)	(new)	gleich	statisch	ja
Overload/ Überladen	- (gleiche Klasse)	- (gleiche Klasse)	unterschiedlich	statisch	nein

Overwrite geerbter Felder

Quellcode

```
using System;
class BK {
   protected string x = "Bast";
   public void BM() {
      Console.WriteLine("x in BK-Methode:\t"+x);
   }
}

class AK : BK {
   new int x = 333;
   public void AM() {
      Console.WriteLine("x in AK-Methode:\t"+x);
      Console.WriteLine("base-x in AK-Methode:\t"+
                        base.x);
   }
}

class Prog {
   static void Main() {
      AK ako = new AK();
      ako.BM();
      ako.AM();
   }
}
```

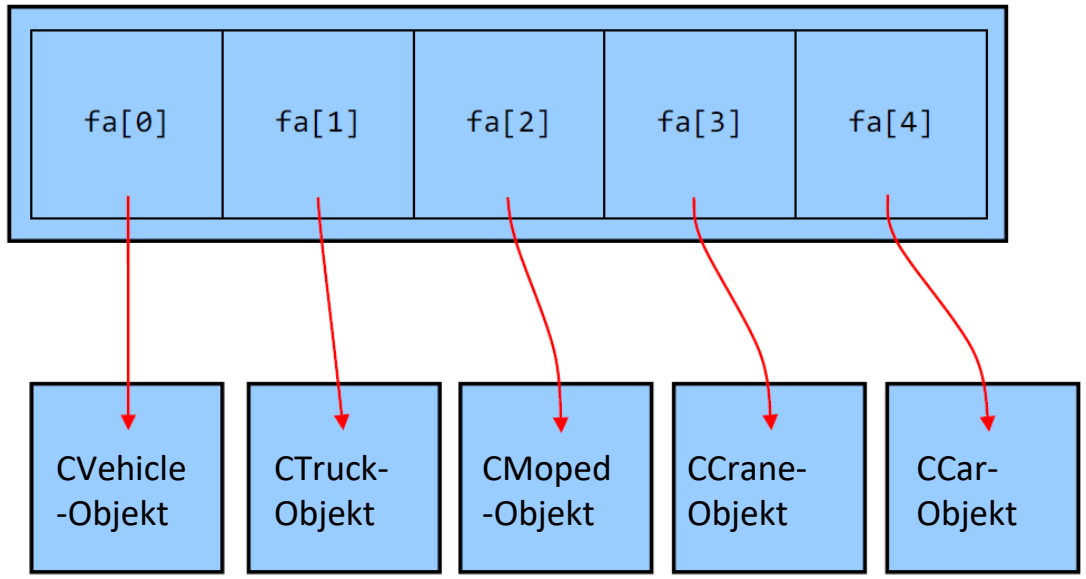
Ausgabe

```
x in BK-Methode:        Bast
x in AK-Methode:        333
base-x in AK-Methode:   Bast
```

Verwaltung von Objekten über Basisklassen-referenzen

Möglich:

Artikel-Array fa mit Elementtyp CItem




```
using System;
class Prog
{
    static void Main()
    {
            CVehicle[] fa = new Cvehicle [5];   // legt 5 Referenzen an

            fa[0] = new CVehicle (55, 5);  // wenn nicht abstrakt
            fa[1] = new CTruck (payload: 2, 180, 2);
            fa[2] = new CMoped (helmet: 1, 45, 1);
            fa[3] = new CCrane (400, 1);
            fa[4] = new CCar (75, 5);
            foreach (CVehicle e in fa)  e.GetNextTUEV();
    }
}
```

is – Operator für dynamischen Typ

is- (Typtest-)Operator:

prüft nicht den deklarierten (statischen), sondern den Laufzeittyp (den aktuellen, **dynamischen** Typ)!

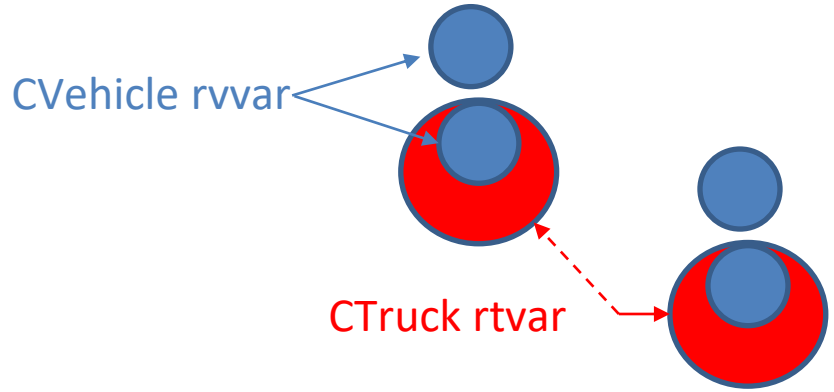
Quellcode	Ausgabe

`using System;`
`class Prog {`
`  static void Main() {`
`  CTruck truck=null;`
`  Console.WriteLine (truck is CTruck);`
`  }`
`}`
	False
	
	(weil *truck* noch keinen dynamischen Typ hat)

Verwaltung von Objekten über Basisklassenreferenzen

Eine Basisklassen-Referenzvariable darf die Referenz eines beliebigen Unterklassenobjekts aufnehmen:
Unterklasse besitzt die komplette Ausstattung der Basisklasse (und mehr) und kann auf alle Methodenaufrufe und Zugriffe geeignet reagieren!

```
CVehicle rvvar = new CTruck (2, 180, 2);   // Truck ist ein Fahrzeug, erlaubt


CTruck rtvar = new CVehicle (180, 2);     // Compilererror, weil ein Fahrzeug noch lange kein Truck ist! PayLoad?
CTruck rtvar = (CTruck) new CVehicle (180, 2);  // System.InvalidCastException zur Laufzeit

ruvar = (ruvar is CTruck) ? null :  new CVehicle(...);   // wenn statischer Typ unbekannt ist
```

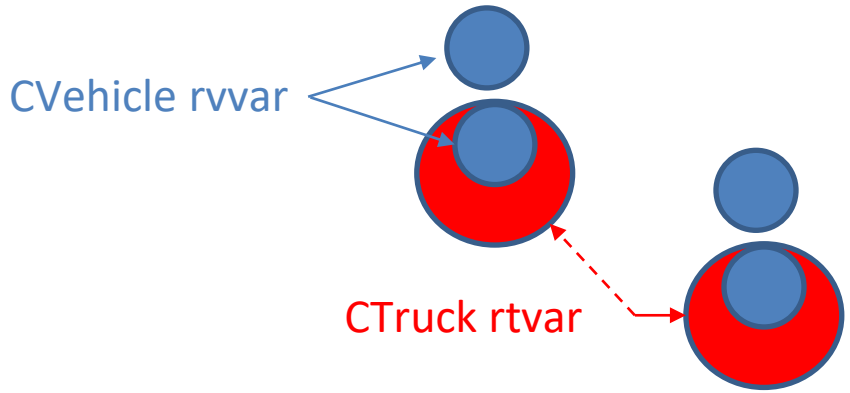


Verwaltung von Objekten über Basisklassenreferenzen

Über eine CVehicle-Referenzvariable rvvar, die auf ein CTruck-Objekt zeigt,
sind *Erweiterungen* der CTruck-Klasse (Zuladung) *nicht* unmittelbar zugänglich!

➔ wenn nötig, dann mit expliziter Typumwandlung:

```
((CTruck)rvvar).PayLoad
// Cast-Operator, u.U. Exception (Ausnahme)

oder besser:   if (rvvar is CTruck) ((CTruck)rvvar).PayLoad

oder noch besser:   (rvvar as CTruck).PayLoad  // as-Operator
// liefert im Unterschied zum Cast-Operator keinen
// Ausnahmefehler, sondern u.U. den Ergebniswert null.
```



as - Operator

Führt Typumwandlungen zwischen kompatiblen Referenztypen durch.

Gibt bei Kompatibilität ein Object zurück oder ***null*** (statt einer InvalidCast-Exception), wenn die Umwandlung nicht möglich ist

```
e as CTruck
```

implementiert als:

```
CTruck result = (e is CDress) ? (Dress)e : null
```

Versiegeln von Methoden

Schutz gegen Überschreiben, z.B. einer Methode Passwd() zum Anfragen des Passworts

```
class A
{
  public virtual void Passwd() { }
}

class B : A
{
  public sealed override void Passwd() { } // kann nicht nochmals überschrieben werden
}
```

Abstrakte: Methoden/ Properties/ Indexer

Wir haben die Methode *GetNextTUEV()* in CVehicle implementiert, und in der abgeleiteten *CMoped*-Klasse mit new verdeckt.

Nicht gut.

Besser:

Abstrakte CVehicle-Klasse teilimplementieren (ohne Methode *GetNextTUEV()*), alle Fahrzeuge ableiten, Methode zwangsweise unterschiedlich implementieren.

Abstrakte: Methoden/ Properties/ Indexer

Um Methoden verschiedener Unterklassen über **Referenzen auf die Basisklassen** zu verwenden, müssen die beteiligten Methoden in der Basisklasse vorhanden sein

Keine sinnvolle Implementierung in der Basisklasse möglich: **abstrakte** Methode:

- Man beschränkt sich auf die Methodendeklaration (kein body) und setzt dort den Modifikator **abstract** (statt virtual)
- direkt abgeleitete Klassen **müssen** die Methode implementieren, sonst sind auch sie abstrakt

- bei mind. einer abstrakten Methode ist die ganze Klasse abstrakt, und kann nicht mehr selbst instantiiert werden, sondern nur noch vererben (*abstract* muss auch im Klassenkopf angegeben werden)

Beispiel: Methode *GetNextTUEV()*

```
public abstract class CVehicle
{
    . . .
    public ~~virtual~~ abstract long? GetNextTUEV();
    . . .
}
```

```
public class CMoped : CVehicle
{
    int Helmet {get; set; } = 1;
    . . .
    public override long? GetNextTUEV()
    {
        return null;
    }
    .
```

Übung

Warum kann der folgende Quellcode nicht übersetzt werden?

```
using System;
public class Basisklasse
{
   int ibas = 3;
   public Basisklasse(int i) { ibas = i; }
   public virtual void Hallo()
   {   Console.WriteLine("Hallo-Methode der Basisklasse");  }
}

public class Abgeleitet : Basisklasse
{
   public override void Hallo()
   {   Console.WriteLine("Hallo-Methode der abgeleiteten Klasse"); }
}
```

```
class Prog
{
    static void Main()
    {
        Abgeleitet s = new Abgeleitet();
        s.Hallo();
    }
}
```

Übung

Im folgenden Beispiel wird die Klasse `Kreis` aus der Klasse `Figur` abgeleitet.
Trotzdem erlaubt der Compiler im initialisierenden `Kreis`-Konstruktor den `Kreis`-Objekten keinen direkten Zugriff auf ihre geerbten Instanzvariablen `xpos` und `ypos`.

Wie ist das Problem zu erklären und zu lösen?

```
public class Figur {
        double xpos = 100.0, ypos = 100.0;
        public Figur(double x, double y) {
            xpos = (x >= 0) ? x : 0;
            ypos = (y >= 0) ? y : 0;
        }
        public Figur() { }
    }
public class Kreis : Figur {
        double radius = 75.0;
        public Kreis(double x, double y, double rad) {
            xpos = (x>=0)?x:0;
            ypos = (y>=0)?y:0;
            radius = (rad>=0)?rad:0;
        }
        public Kreis() { }
    }
```

Übung

a. Überladen von Methoden (overload)
b. Verdecken von Methoden (overwrite)
c. Überschreiben von Methoden (override)

Welche von den drei genannten Programmiertechniken ist bei statischen Methoden *nicht* anwendbar?