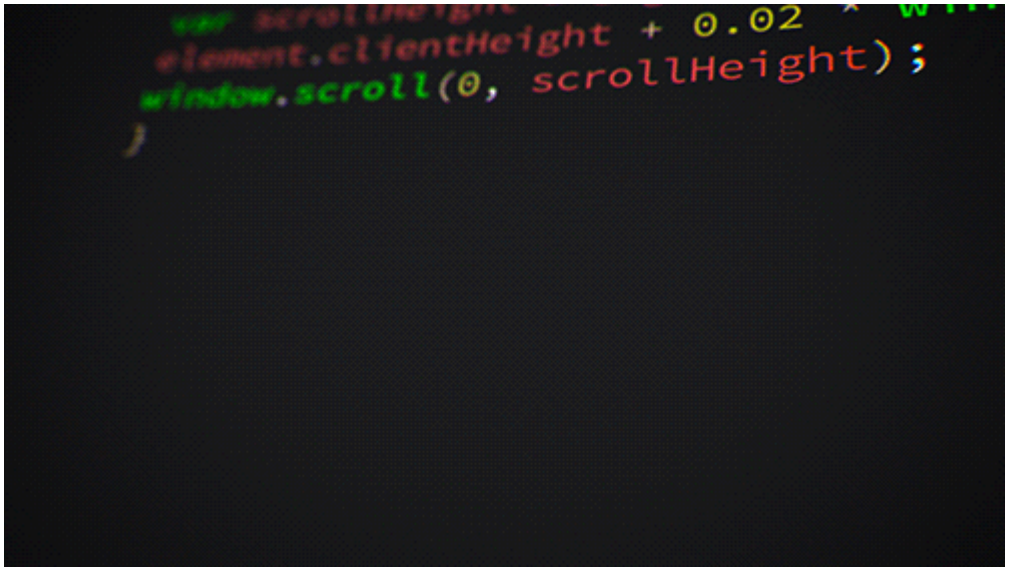
Vererbung

Parameter	Kursinformationen
Veranstaltung:	`Vorlesung Softwareentwicklung`
Teil:	`9/27`
Semester	`Sommersemester 2025`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	Abstrakte Klassen und Methodens, Interface-Definition und -bedeutung, `cast` Operationen
Link auf den GitHub:	https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/09_Vererbung.md
Autoren	Sebastian Zug, Galina Rudolf, André Dietrich



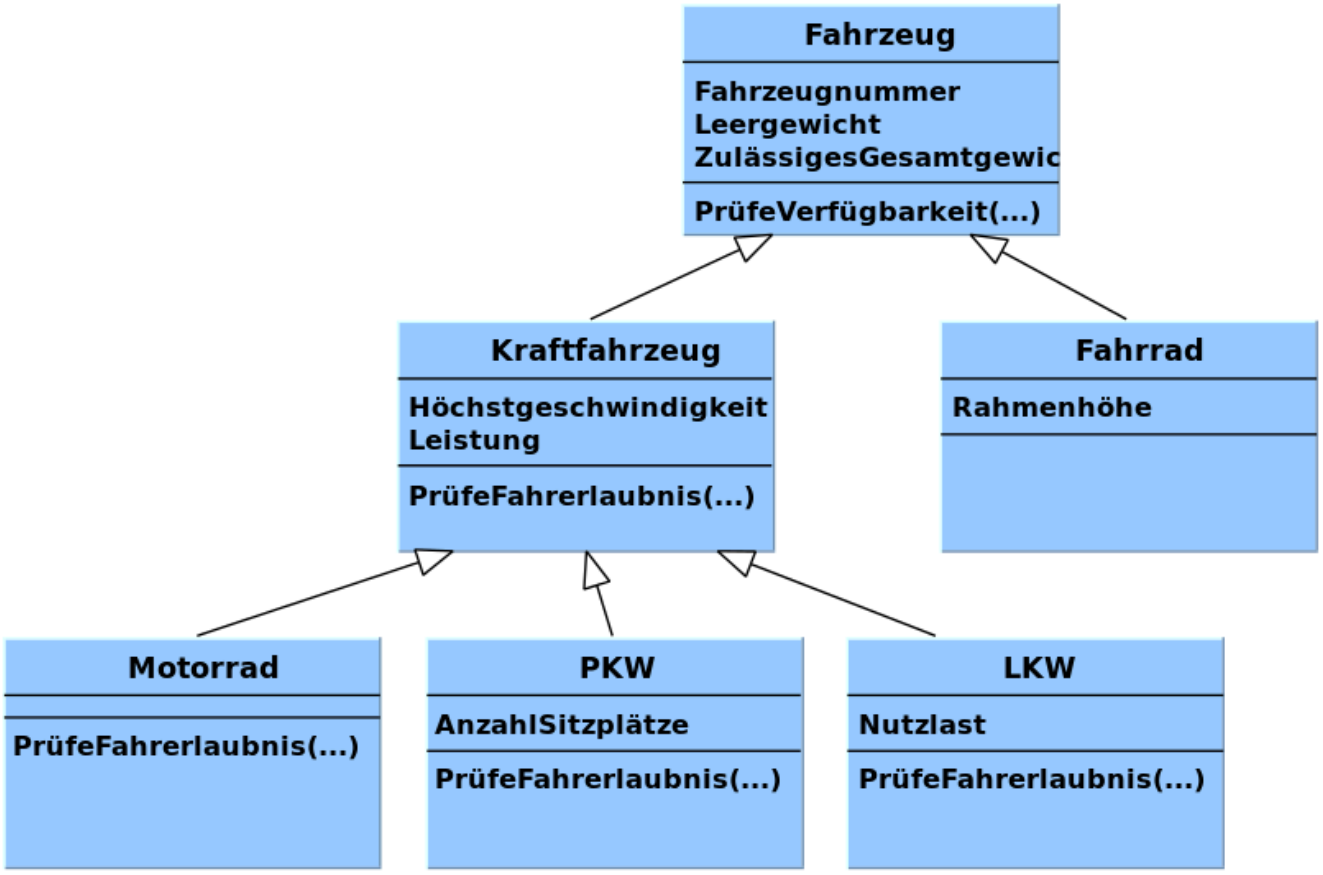
Vererbung in C#

> Vererbung bildet neben Kapselung und Polymorphie die zentrale Säule des objektorientierten Programmierens. Die Vererbung ermöglicht die Erstellung neuer Klassen, die ein in exisitierenden Klassen definiertes Verhalten wieder verwenden, erweitern und ändern. [MS.NET Programmierhandbuch]

Beispiele

Die Klasse, deren Member vererbt werden, wird **Basisklasse** genannt, die erbende Klasse als **abgeleitete Klasse** bezeichnet.

Basisklasse	**abgeleitete Klassen**	**Gemeinsamkeiten**
Fahrzeug	Flugzeug, Boot, Automobil	Position, Geschwindigkeit, Zulassungsnummer, Führerscheinpflicht
Datei	Foto, Textdokument, Datenbankauszug	Dateiname, Dateigröße, Speicherort
Nachricht	Email, SMS, Chatmessage	Adressat, Inhalt, Datum der Versendung



Beispiel einer Vererbungshierarchie in UML Notation [WikiInheri]

Umsetzung in C#

Vererbung

```
using System;
using System.Reflection;
using System.ComponentModel.Design;

public class Person {
  public int geburtsjahr;
  public string name;
}

public class Fußballspieler : Person {
  public byte rückennumemr;
}

public class Schiedsrichter : Person {
  public bool assistent = true;
}

public class Program
{
  public static void Main(string[] args){
    Person Mensch = new Person {geburtsjahr = 1956, name = "Löw"};
    Console.WriteLine("{0,4} - {1}", Mensch.geburtsjahr, Mensch.name
    Console.WriteLine("Felder in der Instanz '{0}' von '{1}'", Mensch
      Mensch);
    var fields = Mensch.GetType().GetFields();
    foreach (FieldInfo field in fields){
      Console.WriteLine(" x " + field.Name);
    }
  }
}
```

```
1956 - Löw
Felder in der Instanz 'Löw' von 'Person'
 x geburtsjahr
 x name
```

Merke: Im Unterschied zu Klassen ist für Structs unter C# keine Vererbung möglich!

In C# kann jede Klassendefinition nur eine Basisklasse referenzieren. Im Sinne einer realitätsnahen Modellierung wären Mehrfachvererbungen aber durchaus zielführend. Ein Amphibienfahrzeug leitet sich aus den Basisklassen Wasserfahrzeug und Landfahrzeug ab, ein Touchpad integriert die Member von Eingabegerät und Ausgabegerät. C# verzichtet drauf um Mehrdeutigkeiten und Fehler ausschließen zu können, die aus gleichnamige Membern hervorgehen.

... und wie erfolgt die Initialisierung?

Konstruktoren werden nicht vererbt, jedoch

- kann mit dem Schlüsselwort `base` auf die Konstruktoren der Basisklasse zurückgegriffen werden.
- wird sofern aus der abgeleiteten Klasse kein expliziter Aufruf erfolgt, der Standardkonstruktor der Basisklasse aufgerufen.

Ein Beispiel für den impliziten Aufruf des Standardkonstruktors:

ImplicitConstructorCall

```
1  using System;
2  using System.Reflection;
3  using System.ComponentModel.Design;
4
5  public class Person {
6    public int geburtsjahr;
7    public string name;
8
9    public Person(){
10     geburtsjahr = 1984;
11     name = "Orwell";
12     Console.WriteLine("ctor of Person");
13   }
14
15   public Person(int auswahl){
16     if (auswahl == 1) {name = "Micky Maus";}
17     else {name = "Donald Duck";}
18   }
19 }
20
21 public class Fußballspieler : Person {
22   public byte rückennummer;
23 }
24
25 public class Program
26 {
27   public static void Main(string[] args){
28     Fußballspieler champ = new Fußballspieler();
29     Console.WriteLine("{0,4} - {1}", champ.geburtsjahr, champ.name );
30   }
31 }
```

```
ctor of Person
1984 - Orwell
ctor of Person
1984 - Orwell
```

Zugriffsmechanismen

Wer darf auf welche Methoden, Properties, Variablen usw. zurückgreifen? Mit der Einführung der Vererbung steigt die Komplexität der Sichtbarkeitsregeln nochmals an.

Zugriffsmodifizierer	Innerhalb eines Assemblys		Außerhalb eines Assemblys	
	Vererbung	Instanzierung	Vererbung	Instanzierung
`public`	ja	ja	ja	ja
`private`	nein	nein	nein	nein
`protected`	ja	nein	ja	nein
`internal`	ja	ja	nein	nein
`ìnternal protected`	ja	ja	ja	nein

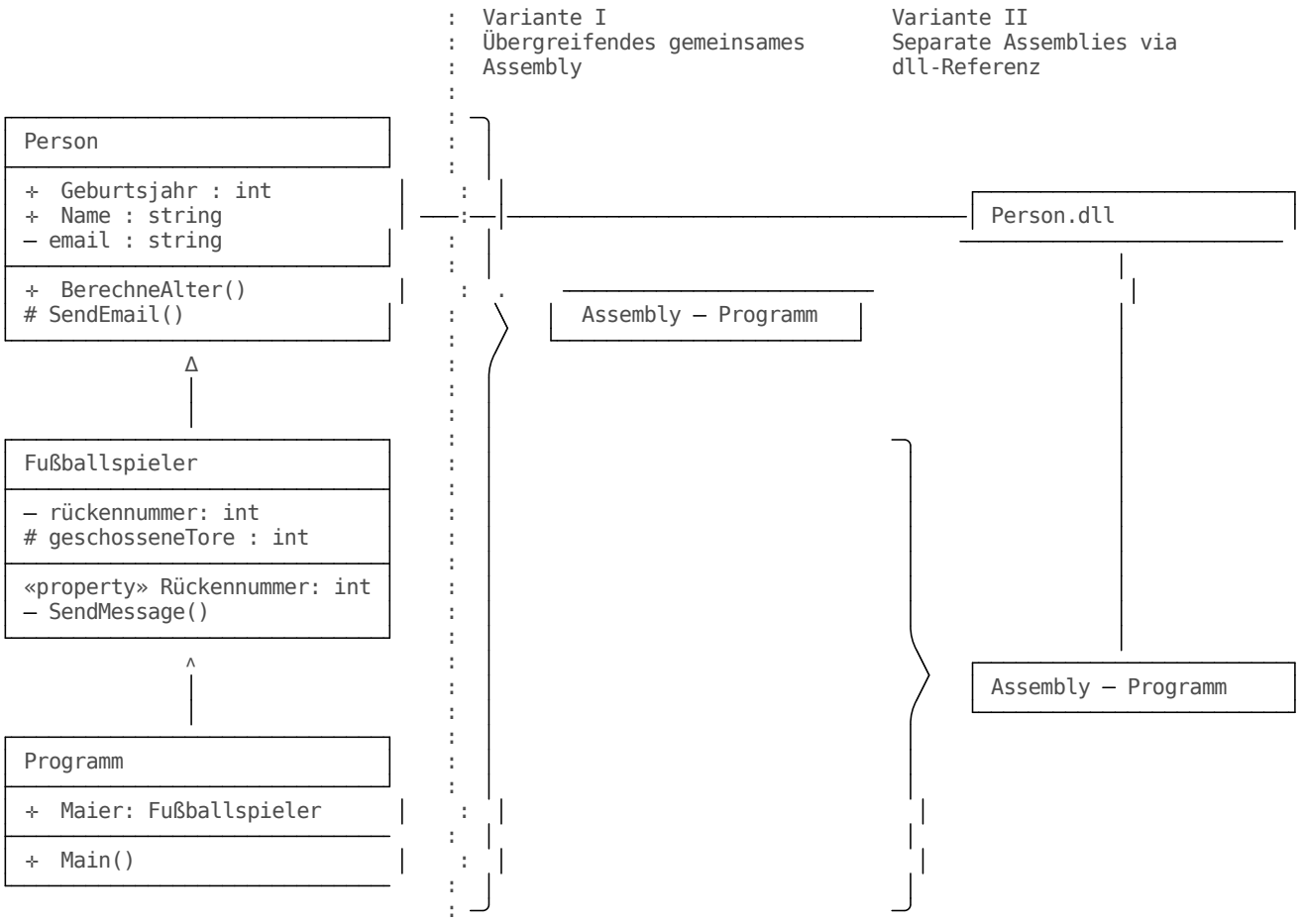
`protected` definiert eine differenzierten Zugriff für geerbte und Instanz-Methoden. Während bei geerbten Elementen uneingeschränkt zugegriffen werden kann, bleiben diese bei der bloßenn Anwendung geschützt.

Die Konzepte von `internal` setzen diese Überlegung fort und kontrollieren den Zugriff über Assembly-Grenzen.

Member der Klasse

Kriterien der Zugriffsattribute:

- innerhalb/außerhalb einer Klasse
- innerhalb der Vererbungshierachie einer Klasse / außerhalb ("nutzt")
- innerhalb des Assemblys / außerhalb



Für Methoden, Membervariablen etc. ist das klar, aber macht es Sinn geschützte private Konstruktoren zu definieren?

Private Konstruktoren werden verwendet, um die Instanziierung einer Klasse zu verhindern, die ausschließlich statische Elemente hat. Ein Beispiel dafür ist die `Math` Klasse, die Methoden definiert, die ohne eine Instanz der Klasse aufgerufen werden. Wenn alle Methoden in der Klasse statisch sind, wäre es ggf. sinnvoll die gesamte Klasse statisch anzulegen.

privateConstructors

```
1  using System;
2
3  public class Counter
4  {
5      private Counter() { }
6      public static int currentCount;
7
8      public static int IncrementCount()
9      {
10         return ++currentCount;
11     }
12 }
13
14 public class Program
15 {
16   public static void Main(string[] args){
17     Counter myCounter = new Counter();
18     //Console.WriteLine()
19   }
20 }
```

```
Compilation failed: 1 error(s), 0 warnings
main.cs(17,25): error CS0122: `Counter.Counter()' is inaccessible due
to its protection level
main.cs(5,13): (Location of the symbol related to previous error)
Compilation failed: 1 error(s), 0 warnings
main.cs(17,25): error CS0122: `Counter.Counter()' is inaccessible due
to its protection level
main.cs(5,13): (Location of the symbol related to previous error)
```

Klasse

Auch für Klassen selbst können Zugriffsattribute das Verhalten bestimmen:

- 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)

Polymorphie in C#

Strukturieren Sie die Klassen "Zug", "GüterZug", "PersonenZug" und "ICE" in einer sinnvolle Vererbungshierarchie. Wie setzen Sie diese in C# Code um?

Constructors

```
using System;
using System.Reflection;
using System.ComponentModel.Design;

class Zug
{
  string nummer;
  public Zug()
  {
    Console.WriteLine("Zug-ctor");
  }
  public Zug(string nummer)
  {
    this.nummer = nummer;
    Console.WriteLine("Generischer Zug-ctor");
  }
}

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

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

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


public class Program
{
  public static void Main(string[] args)
  {
    Console.WriteLine("Generiere neuen ICE ");
```

```
49      Ice ice = new Ice();
50      Console.WriteLine("Generieren neuen Güterzug");
51      GueterZug gueter = new GueterZug();
52    }
53  }
```

```
Generiere neuen ICE
Generischer Zug-ctor
Personen Zug-ctor
ICE-ctor
Generieren neuen Güterzug
Zug-ctor
GueterZug-ctor
Generiere neuen ICE
Generischer Zug-ctor
Personen Zug-ctor
ICE-ctor
Generieren neuen Güterzug
Zug-ctor
GueterZug-ctor
```

> **Merke:** Konstruktoren werden nicht geerbt! Jede Unterklasse deklariert (implizit) eigene Konstruktoren.

Die Konstruktoren der Basisklasse können jeweils mit `base()` aufgerufen werden. Erfolgt dies nicht, wird der parameterlose Konstruktor der Basisklasse automatisch aufgerufen.

Die Ausgabe des oben aufgeführten Beispiels illustriert diese Aufrufhierachie. Entfernen Sie dem `base` Aufruf in Zeile 23 und erklären Sie den Unterschied.
In diesem Fall ist `Zug` die Basisklasse und `PersonenZug`, `GueterZug` und `ICE` sind abgeleitete Klassen.

Eine Variablen vom Basisdatentyp kann immer eine Instanz einer abgeleiteten Klasse zugewiesen werden. Entsprechend unterscheidet man dann zwischen dem statischen und dem dynamischen Typ der Variablen. Der statische Typ ist immer der, der auch deklariert wurde. Der dynamische Typ wird durch die aktuelle Referenz einer Instanz einer abgeleiteten Klasse von Zug bestimmt und ist veränderlich.

Zuweisung	statischer Typ von Zug	dynamischer Typ von Zug
`Zug RB51 = new Zug()`	Zug	Zug
`RB51 = new PersonenZug()`	Zug	PersonenZug
`RB51 = new Ice`	Zug	ICE

Laufzeitprüfung

Entsprechend brauchen wir eine Typprüfung, die untersucht, ob die Variable von einem bestimmten dynamischen Typ oder einem daraus abgeleiteten Typ ist.

- der dynamische Typ einer Klasse kann zur Laufzeit geprüft werden
- Typtest liefert bei null-Werten immer `false`

Typprüfung

```
using System;
using System.Reflection;
using System.ComponentModel.Design;

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");
  }
}

public class Program
{
  public static void Main(string[] args)
  {
    Zug IC239 = new Ice();
    Console.WriteLine($"Statischer Typ von IC239 {IC239.GetType()}");
    Console.WriteLine("IC239 ist ein Zug? " + (IC239 is Zug)); // tru
    Console.WriteLine("IC239 ist ein PersonenZug? " + (IC239 is Perso
      )); // true
    Console.WriteLine("IC239 ist ein Ice? " + (IC239 is Ice)); // tru
    IC239 = null;
    Console.WriteLine("IC239 ist ein Ice? " + (IC239 is Ice)); // fal
  }
}
```

```
Zug-ctor
PersonenZug-ctor
ICE-ctor
Statischer Typ von IC239 Ice
IC239 ist ein Zug? True
IC239 ist ein PersonenZug? True
IC239 ist ein Ice? True
IC239 ist ein Ice? False
Zug-ctor
PersonenZug-ctor
ICE-ctor
Statischer Typ von IC239 Ice
IC239 ist ein Zug? True
IC239 ist ein PersonenZug? True
IC239 ist ein Ice? True
IC239 ist ein Ice? False
```

Grundidee der Polymorphie

Objekte einer Basisklasse können somit Instanzen einer abgeleiteten Klassen umfassen. Damit lassen sich ähnlich einem Container sehr unterschiedliche Objekte einer Vererbungslinie bündeln.

> Warum ist das wichtig? Was bringt mir das?

Polymorphie

```
1  using System;
2  using System.Reflection;
3  using System.ComponentModel.Design;
4
5  class Zug
6  {
7    public Zug()
8    {
9      Console.WriteLine("Zug-ctor");
10   }
11 }
12
13 class PersonenZug : Zug
14 {
15   public PersonenZug() : base()
16   {
17     Console.WriteLine("PersonenZug-ctor");
18   }
19 }
20
21 class Ice : PersonenZug
22 {
23   public Ice()
24   {
25     Console.WriteLine("ICE-ctor");
26   }
27 }
28
29 class Abfahrtszeit
30 {
31   public int stunde;
32   public int minute;
33   public Abfahrtszeit(int stunde, int minute, Zug zug)
34   {
35     this.zug = zug
36     this.stunde = stunde;
37     this.minute = minute;
38   }
39 }
40
41 public class Program
42 {
43   public static void Main(string[] args)
44   {
45     Zug IC239 = new Ice();
46     Abfahrtszeit abfahrt = new Abfahrtszeit(12, 30, IC239);
47   }
48 }
```

```
Compilation failed: 1 error(s), 0 warnings
main.cs(36,4): error CS1525: Unexpected symbol `this'
Compilation failed: 1 error(s), 0 warnings
main.cs(36,4): error CS1525: Unexpected symbol `this'
```

Wir haben schon gesehen, dass die Vererbung unter anderem Methoden umfasst. Auf welche Klassenmember greife ich überhaupt zurück?

Merke Polymorphie bezeichnet die Tatsache, dass Klassenmember ausgehend von Ihrer Nutzung ein unterschiedliches Verhalten erzeugen.

Das heißt, die Methoden der Klassen einer Vererbungshierarchie können auf verschiedenen Ebenen gleiche Signatur, aber unterschiedliche Implementierungen haben. Welche der Methoden für ein gegebenes Objekt aufgerufen wird, wird erst zur Laufzeit bestimmt (dynamische Bindung).

Merke Dynamische Bindung bezeichnet die Tatsache, dass bei Aufruf einer überschriebenen Methode über eine Basisklassenreferenz oder ein Interface trotzdem die Implementierung der abgeleiteten Klasse zum Einsatz kommt.

Dynamische Bindung erlaubt den Aufruf von überschriebenen Methoden aus der Basisklasse heraus, wobei das Überschreiben muss in der Basisklasse explizit erlaubt werden muss.

Überschreiben von Methoden

In C# können abgeleitete Klassen Methoden mit dem gleichen Namen wie Basisklassen-Methoden enthalten. Diese Methoden müssen dann in der Basisklasse mittels `virtual` als explizit überschreibbar deklariert werden:

```
public virtual void makeSound() => Console.WriteLine("I'm an Animal");
```

Zum Überschreiben wird das Schlüsselwort `override` genutzt, welches ein erneutes Deklarieren ermöglicht:

```
public override void makeSound() => Console.WriteLine("Quack!");
```

Dabei müssen beide Methoden die gleiche Signatur haben, d.h. sie sollen die den gleichen Namen und eine identische Parameterliste besitzen. Ansonsten ist es nur Überladung!

Polymorphy.cs

```
using System;

class Animal
{
  public string Name;
  public Animal(string name){
    Name = name;
  }
  public virtual void makeSound(){
    Console.WriteLine("I'm an Animal");
  }
}

class Duck : Animal
{
  public Duck(string name) : base(name) { }
  public override void makeSound(){
    Console.WriteLine("{0} - Quack ({1})", Name, this.GetType().Name)
  }
}

class Cow : Animal
{
  public Cow(string name) : base(name) { }
  public override void makeSound(){
    Console.WriteLine("{0} - Muh ({1})", Name, this.GetType().Name);
  }
}

public class Program
{
  public static void Main(string[] args){
    Animal[] animals = new Animal[3]; // <- Statischer Typ  Animal
    animals[0] = new Duck("Alfred");  // <- Dynamischer Typ Duck
    animals[1] = new Cow("Hilde");
    animals[2] = new Animal("Bernd");
    foreach (Animal anim in animals)
      anim.makeSound();
  }
}
```

```
Alfred - Quack (Duck)
Hilde - Muh (Cow)
I'm an Animal
Alfred - Quack (Duck)
Hilde - Muh (Cow)
I'm an Animal
```

Die verschiedenen Tierklassen werden auf ihre Basisklasse gecastet, trotzdem aber die individuelle Implementierung von makeSound ausgeführt. Damit erlaubt die Polymorphie ein gleichartiges Handling unterschiedlicher Klassen, die über die Vererbung miteinander verknüpft sind.

Interessant ist die Möglichkeit die ursprüngliche Implementierung der Methode aus der Basisklasse weiterhin zu nutzen und zu erweitern:

```
class Horse : Animal
{
  public Horse(string name) : base(name) { }
  public override void makeSound()
  {
    base.makeSound();
    Console.WriteLine("Ich ziehe Kutschen");
  }
}
```

Dazu kann die Methode aus der Basisklasse über `base.<Methodenname>` aufgerufen werden

Verdecken von Methoden

Sollen die spezifischen Methoden aber nur im Kontext der Klasse realisierbar sein, so werden sie vor der Basisklasse "verdeckt". Dazu ist das Schlüsselwort `new` erforderlich. In diesem Fall wird keine dynamische Bindung realisiert, sondern die Methode der Basisklasse aufgerufen.

newOperator

```
using System;

class Animal
{
  public string Name;
  public Animal(string name){
    Name = name;
  }
  public virtual void makeSound(){
    Console.WriteLine("I'm an Animal");
  }
}

class Cat : Animal
{
  public Cat(string name) : base(name) { }
  public new void makeSound(){
    Console.WriteLine("{0} - Miau ({1})", Name, this.GetType().Name);
  }
}

public class Program
{
  public static void Main(string[] args){
    Cat myCat = new Cat("Kity");
    myCat.makeSound();
    Animal myCatAsAnimal = new Cat("KatziTatzi");
    myCatAsAnimal.makeSound();
  }
}
```

```
Kity - Miau (Cat)
I'm an Animal
Kity - Miau (Cat)
I'm an Animal
```

Verdeckt werden können alle Klassenmember einer Basisklasse:

- Felder
- Properties und Indexer
- Methoden usw.

Wenn kein Schlüsselwort angegeben ist, wird implizit `new` angenommen. Im oben genannten Beispiel folgt daraus, dass die in `Cat` implementierte Ausgabe ausschließlich von Objekten des statischen Typs `Cat` aufgerufen werden kann. Testen Sie die Wirkung und ersetzen Sie `new` durch `override`.

Das folgende Beispiel entstammt dem C# Programmierhandbuch und kann unter Link nachgelesen werden.

Nehmen wir an, dass Ihre Software eine Grafikbibliothek nutzt, die folgende Funktionen bietet:

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
}
```

Sie haben darauf aufbauend eine umfangreiches Framework geschieben und in einer Klasse, die von GraphicsClass erbt eine Methode `DrawRectangle` implementiert.

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public void DrawRectangle() { }
}
```

Nun entwickelt der Hersteller eine neue Version von GraphicsClass und integriert eine eigene Realisierung von `DrawRectangle`. Sobald Sie Ihre Anwendung neu gegen die Bibliothek kompilieren, erhalten Sie vom Compiler eine Warnung. Diese Warnung informiert Sie darüber, dass Sie das gewünschte Verhalten der DrawRectangle-Methode in Ihrer Anwendung bestimmen müssen. Welche Möglichkeiten haben Sie - override oder new oder umbenennen? Welche Konsequenzen ergeben sich daraus?

Zusammenfassung

Kriterium	Virtuelle Methode	Nicht-virtuelle Methode
Schlüsselwort	`virtual` / `override`	(kein Schlüsselwort) / `new`
Überschreibbar?	Ja	Nein (nur versteckbar)
Laufzeitbindung	Spät (dynamic dispatch)	Früh (static dispatch)
Polymorphie möglich?	Ja	Nein

Versiegeln von Klassen oder Membern

Die Mechanismen der Vererbung und Polymorphie können aber auch aufgehoben werden, wenn ein Schutz notwendig ist. Das Schlüsselwort `sealed` ermöglicht es sowohl Klassen von der Rolle als Basisklasse auszuschließen als auch das Überschreiben von Methoden zu verhindern.

```
class A {}
```

```
sealed class B : A {}
```

Im Beispiel erbt die Klasse B von der Klasse A, allerdings kann keine Klasse von der Klasse B erben.

Merke: Da Strukturen implizit versiegelt sind, können sie nicht geerbt werden.

sealedMethods

```
using System;

sealed public class Animal
{
  public string Name;
  public Animal(string name){
    Name = name;
  }
  public virtual void makeSound(){
    Console.WriteLine("I'm a Crocodile");
  }
}

class Cat : Animal
{
  public Cat(string name) : base(name) { }
  public sealed override void makeSound(){   // sealed schützt die Ca
    .makeSound methode
    Console.WriteLine("{0} - Miau ({1})", Name, this.GetType().Name);
  }
}

class Tiger : Cat
{
  public Tiger(string name) : base(name) { }
  public override void makeSound(){
    Console.WriteLine("{0} - Grrrr ({1})", Name, this.GetType().Name)
  }
}

public class Program
{
  public static void Main(string[] args){
    Tiger evilTiger = new Tiger("Shir Khan");
    evilTiger.makeSound();
  }
}
```

```
Compilation failed: 3 error(s), 0 warnings
main.cs(14,7): error CS0509: `Cat': cannot derive from sealed type
`Animal'
main.cs(3,21): (Location of the symbol related to previous error)
main.cs(9,23): error CS0549: New virtual member `Animal.makeSound()' is
declared in a sealed class `Animal'
main.cs(25,24): error CS0239: `Tiger.makeSound()': cannot override
inherited member `Cat.makeSound()' because it is sealed
main.cs(17,31): (Location of the symbol related to previous error)
Compilation failed: 3 error(s), 0 warnings
main.cs(14,7): error CS0509: `Cat': cannot derive from sealed type
`Animal'
main.cs(3,21): (Location of the symbol related to previous error)
main.cs(9,23): error CS0549: New virtual member `Animal.makeSound()' is
declared in a sealed class `Animal'
main.cs(25,24): error CS0239: `Tiger.makeSound()': cannot override
inherited member `Cat.makeSound()' because it is sealed
main.cs(17,31): (Location of the symbol related to previous error)
```

Casts über Klassen

Konvertierungen zwischen unterschiedlichen Datentypen lassen sich auch auf Klassen anwenden, allerdings sind hier einige Besonderheiten zu beachten.

- implizit auf die Basisklasse (upcast)
- explizit auf die abgeleitete Klasse (downcast)

gecastet werden. Zunächst ein Beispiel für einen *upcast* anhand unseres Fußballbeispiels. Zugriffe auf Member, die in der Basisklasse nicht enthalten sind führen logischerweise zum Fehler.

Upcast

```
using System;
using System.Reflection;
using System.ComponentModel.Design;

public class Person {
  public int geburtsjahr;
  public string name;
}

public class Fußballspieler : Person {
  public byte rückennummer;
}

public class Program
{
  public static void Main(string[] args)
  {
    Fußballspieler champ = new Fußballspieler {geburtsjahr = 1956,
                                               name = "Maier",
                                               rückennummer = 13};
    Console.WriteLine("Felder in der Instanz '{0}' von '{1}'", champ.
      champ);
    var fields = champ.GetType().GetFields();
    foreach (FieldInfo field in fields){
      Console.WriteLine(" x " + field.Name);
    }
    Person human = champ;     // Castoperation Fußballspieler -> Pers

    Console.WriteLine("Felder in der Instanz '{0}' von '{1}'", human.
      human);
    var fields2 = human.GetType().GetFields();
    foreach (FieldInfo field in fields2){
      Console.WriteLine(" x " + field.Name);
    }

    Console.WriteLine("human ist ein Fußballspieler? " + (human is
      Fußballspieler));
    // Console.WriteLine(human.rückennummer);
  }
}
```

```
Felder in der Instanz 'Maier' von 'Fußballspieler'
 x rückennummer
 x geburtsjahr
 x name
Felder in der Instanz 'Maier' von 'Fußballspieler'
 x rückennummer
 x geburtsjahr
 x name
human ist ein Fußballspieler? True
Felder in der Instanz 'Maier' von 'Fußballspieler'
 x rückennummer
 x geburtsjahr
 x name
Felder in der Instanz 'Maier' von 'Fußballspieler'
 x rückennummer
 x geburtsjahr
 x name
human ist ein Fußballspieler? True
```

In umgekehrter Richtung vollzieht sich der *Downcast*, eine Instanz der Basisklasse wird auf einen abgeleiteten Typ gemappt.

Downcast

```
using System;

public class Person {
  public int geburtsjahr;
  public string name;
}

public class Fußballspieler : Person {
  public byte rückennummer;
}

public class Program
{
  public static void Main(string[] args)
  {
    // Erzeuge eine Instanz der abgeleiteten Klasse
    Fußballspieler original = new Fußballspieler {
      geburtsjahr = 1956,
      name = "Maier",
      rückennummer = 1
    };

    // Upcast: speichere sie in einer Person-Variablen
    Person human = original;

    // Downcast: jetzt ist es sicher!
    Fußballspieler champ = (Fußballspieler) human;

    Console.WriteLine($"{champ.name} trägt die Nummer {champ.rückennu
      );
  }
}
```

```
Maier trägt die Nummer 1
Maier trägt die Nummer 1
```

Beispiel

Upcast und *Downcast* ... wozu brauche ich das den? Nehmen wir an, dass wir eine Ausgabemethode für beide Typen - Person und Fußballspieler - benötigen. Ja, es wäre möglich diese als Memberfunktion zu implementieren, problematisch wäre aber dann, dass wir an unterschiedlichen Stellen im Code spezifische Befehle für die Ausgabe in der Konsole zu stehen haben. Sollen die Log-Daten nun plötzlich in eine Datei ausgegeben werden, müsste diese Anpassung überall vollzogen werden. Entsprechend ist eine externe

(statische) Logger-Klasse wesentlich geeigneter diese Funktionalität zu kapseln. Allerdings wäre dann ein überladen der entsprechenden Ausgabefunktion mit allen vorkommenden Typen notwendig. Dies kann durch entsprechende Casts umgangen werden.

UpCastExample

```
using System;
using System.Reflection;
using System.ComponentModel.Design;

public class Person
{
  public int geburtsjahr;
  public string name;
}

public class Fußballspieler : Person
{
  public byte rückennummer;
}

public static class Logger
{
  public static void printPerson(Person person){
     Console.WriteLine("{0} - {1}", person.name, person.geburtsjahr)
     if (person is Fußballspieler)
       Console.WriteLine("{0} - {1}", person.name, (person as
         Fußballspieler).rückennummer);
  }
}

public class Program
{
  public static void Main(string[] args)
  {
     Person Mensch = new Person {geburtsjahr = 1956,
                                 name = "Maier"};
     Logger.printPerson(Mensch);
     Fußballspieler Champ = new Fußballspieler{geburtsjahr = 1967,
                                               name = "Müller",
                                               rückennummer = 13};
     Logger.printPerson(Champ);
  }
}
```

```
Maier - 1956
Müller - 1967
Müller - 13
Maier - 1956
Müller - 1967
Müller - 13
```