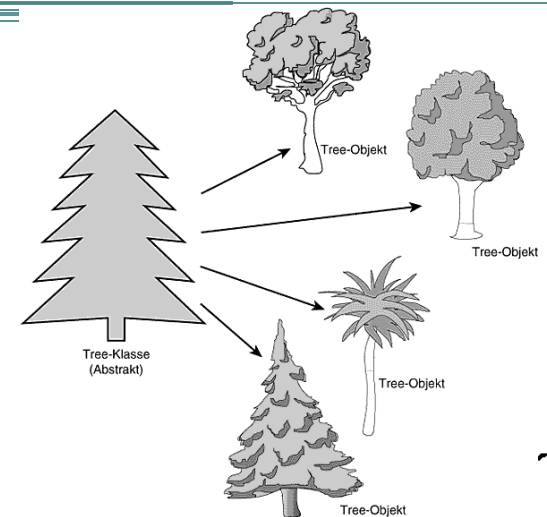


Objektorientierte Programmierung

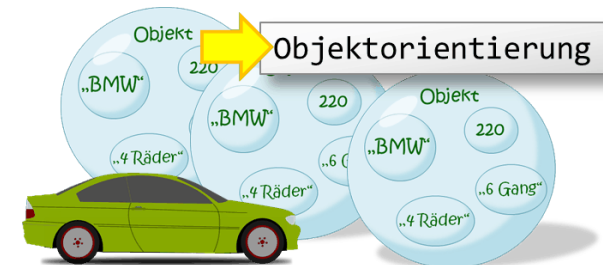
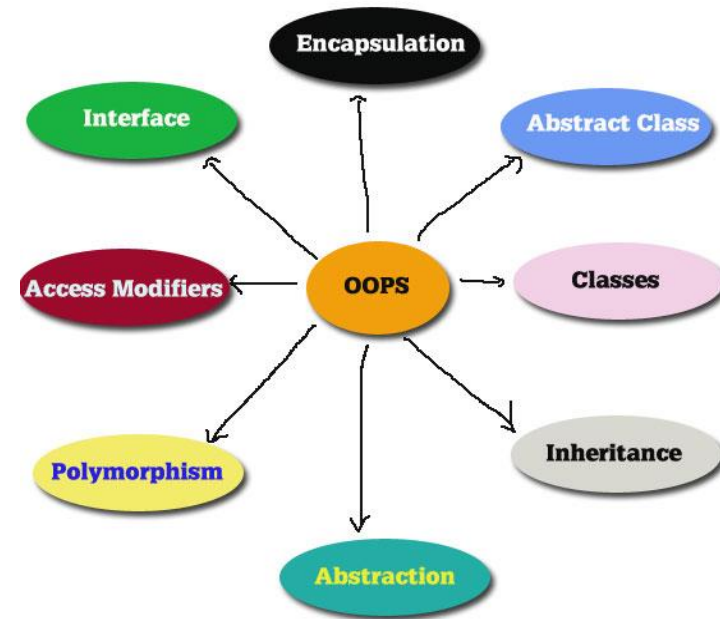
Einführung in
Objektorientiertes Denken

mit C# Beispielen



Übersicht

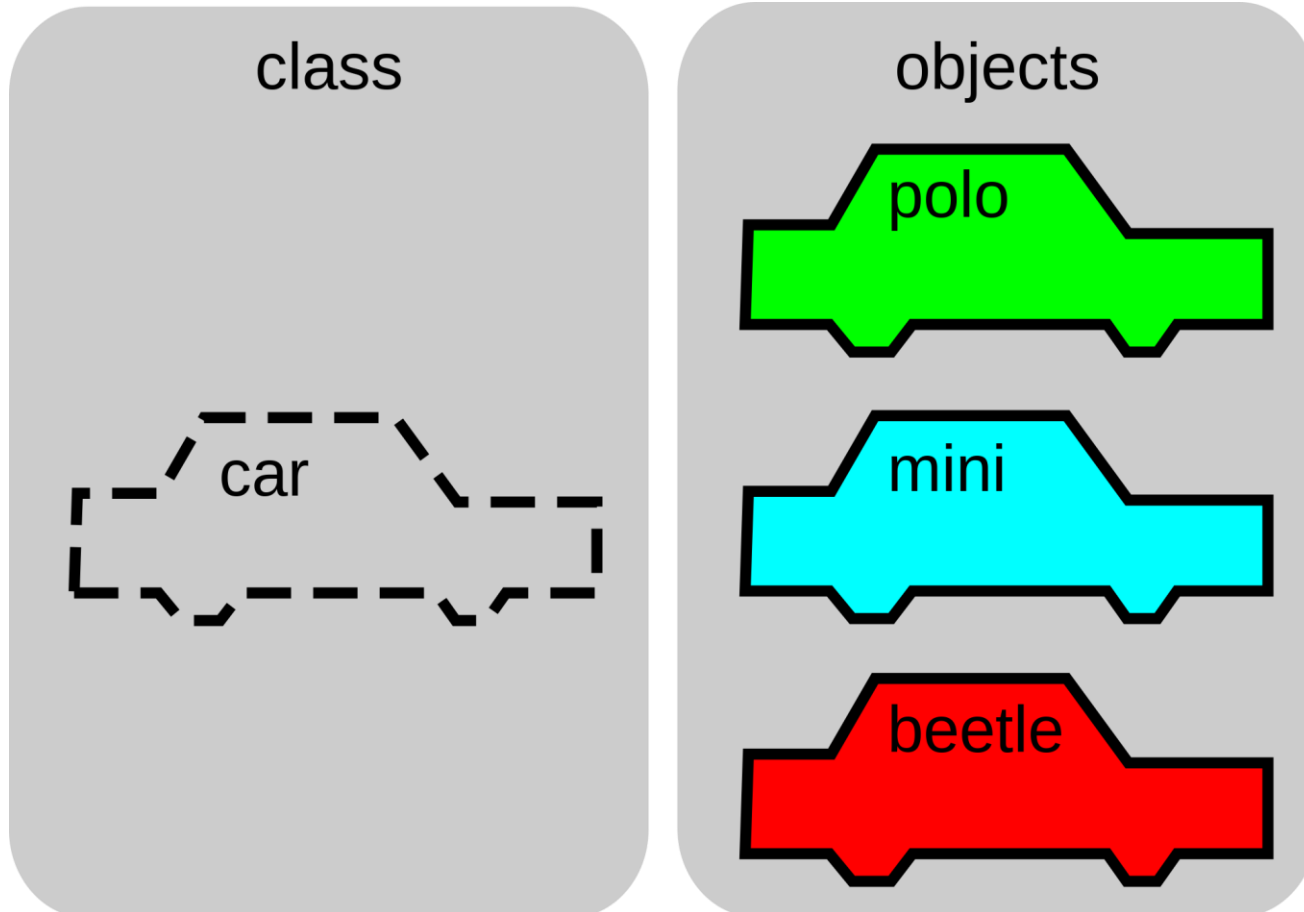
- Denken in Objekten
- Objekt: Klasse vs. Instanz
- Datenkapselung
- Konstruktor
 - Konstruktoren- & Methodenüberladen
 - this vs. base
- Vererbung
 - Konstruktorenkette
- Polymorphie
 - Methodenüberschreiben
 - Statische vs dynamische Bindung
- SOLID Prinzipien



Einfärbig Schule



Klasse vs Objekt (Instanz)



Klasse vs Objekt (Instanz)

Class

Definition of objects that share structure, properties and behaviours.



Building
class



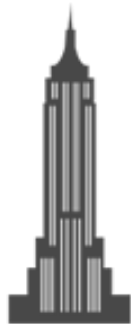
Dog
class



Computer
class

Instance

Concrete object, created from a certain class.



Empire State
instance of Building



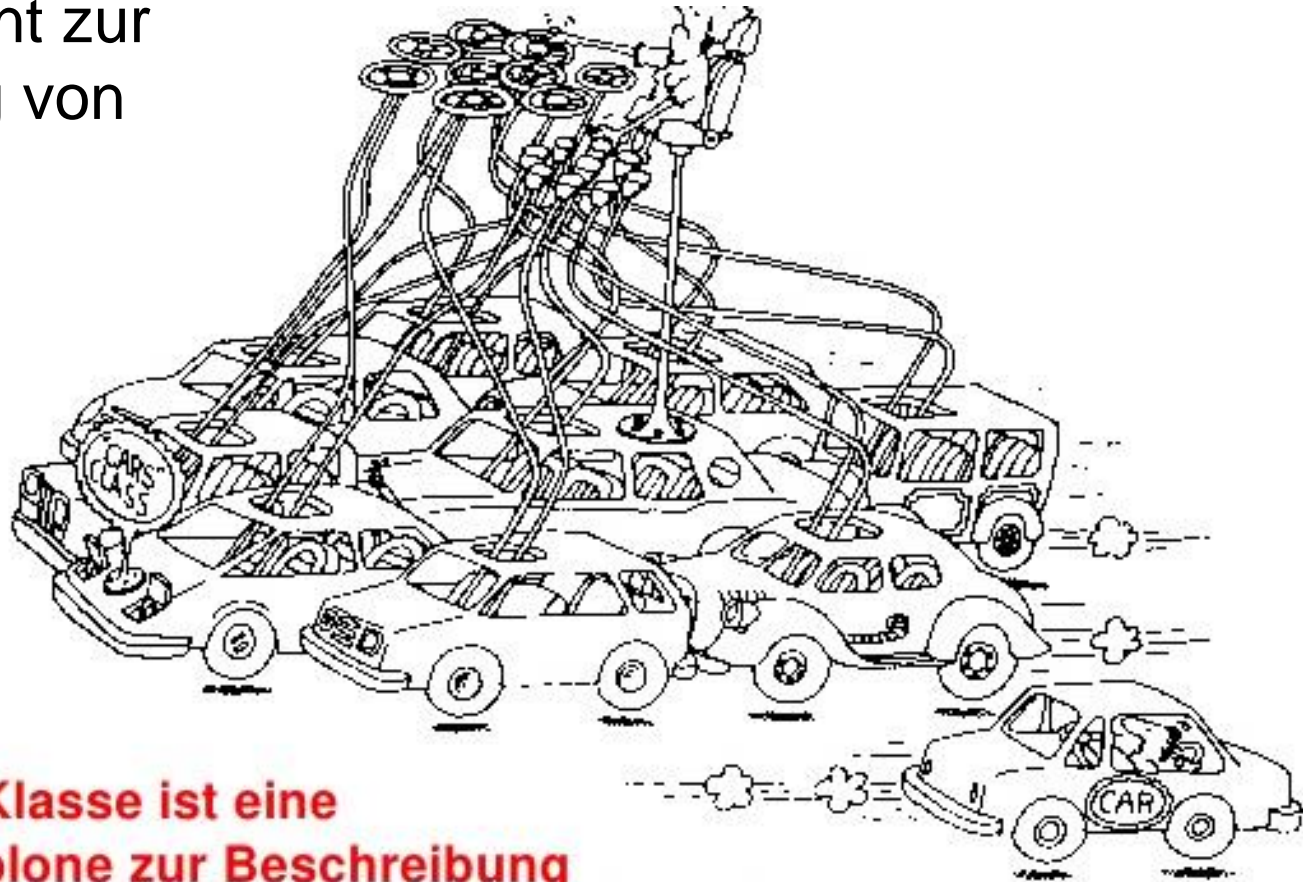
Lassie
instance of Dog



Your computer
instance of Computer

Was ist eine Klasse...

Klasse dient zur
Erzeugung von
Objekten

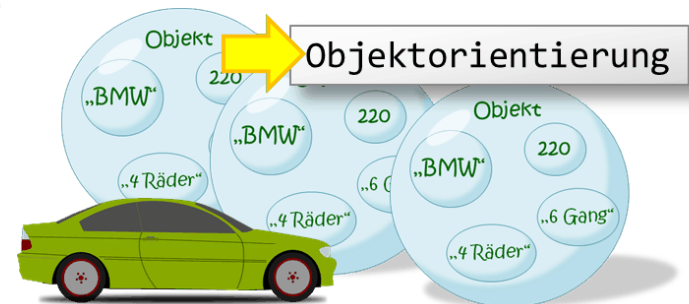


**Eine Klasse ist eine
Schablone zur Beschreibung**

einer Menge von Objekten mit
gemeinsamer Struktur und gemeinsamen Verhalten

Klassen

- zunächst nur 'einfache' Klassen
 - haben **nur Attribute** (Eigenschaften)
 - keine Methoden (erst später)
→ zunächst keine Kapselung
- Klasse besteht aus
 - einem Klassennamen
 - beliebig vielen Attributen
 - Attribute haben einen (einfachen) Typ, z.B. int, double, char, bool, ...
 - Beispiel: Klasse „Car“ mit Name, Anzahl Räder, Ps, Anzahl Gänge



Klassenname

Definition einer Klasse

- Syntax:
[modifier] **class** *Klassenname* {
 [modifier] **Typ1** *Variablenname1*;
 [modifier] **Typ2** *Variablenname2*;
 ...
}

Attribute werden deklariert wie Variablen, stehen jedoch innerhalb einer Klasse.

- Beispiel:
class *Car* {
 string name;
 int ps;
 ...
}

Deklaration der Attribute: jede Instanz (Objekt) der Klasse hat gleichnamige Variablen (Attribute); die Werte können aber für jedes Objekt individuell zugewiesen werden.

Klassen und Objekte

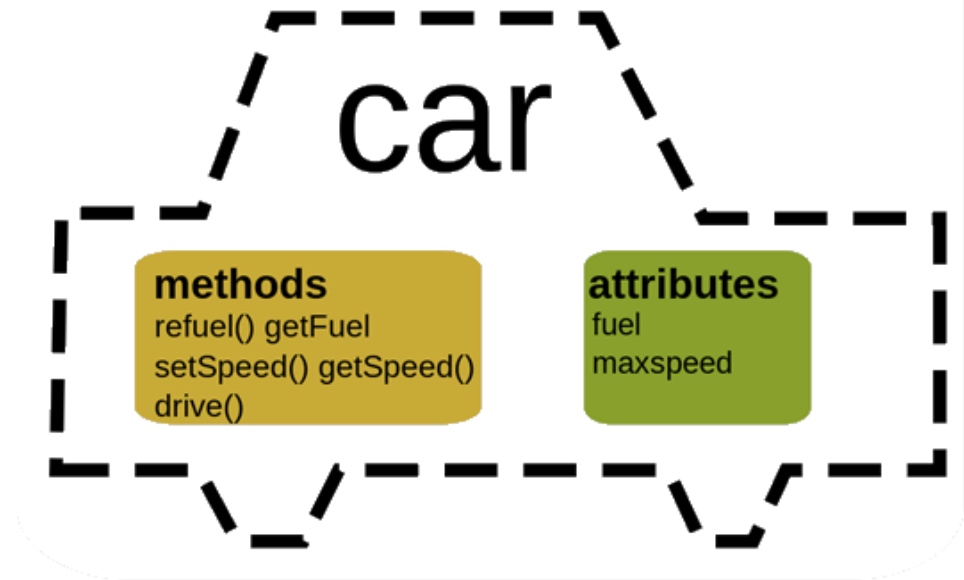
- **Klassen** definieren allgemeine Eigenschaften
 - Definition geometrischer Figuren
 - Bauplan für ein Haus
- konkrete Ausprägung einer Klasse sind **Objekte** oder auch **Instanzen** genannt:
 - Rechteck mit Seitenlängen 10 und 7.5
 - Wohnhaus mit 100m² in Krems

Erzeugen von Objekten einer Klasse

- Syntax:

Klassenname *Variable*; // Deklaration
Variable = new *Klassenname* (); // Instantiierung

- **Car** c1; // Deklaration der Variablen c1
 c1 = new **Car**(); // Erzeugung einer Instanz der
 // Klasse Car; c1 ist eine
 // Referenz auf die Instanz
- **Car** c2; // Es können mehrere Referenzen
 c2 = new **Car**(); // auf eine Instanz zeigen
 Car c3 = c2;



Status & Verhalten

einer Klasse

Status & Verhalten

Klasse besteht aus Status und Verhalten:

- **Status** sind die Eigenschaften eines Objektes
 - Attribute eines Objekts
- **Verhalten** sind die Funktionalitäten
 - Methoden einer Klasse

Klasse besteht aus

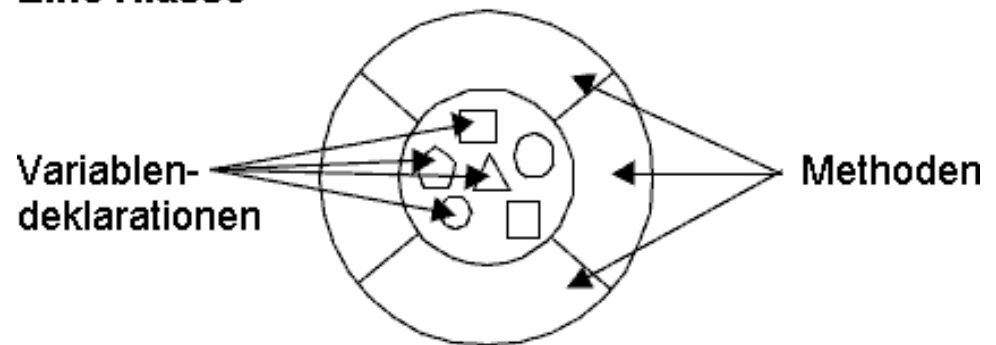
Status

- Attribute
- Eigenschaften
- Variablendeklaration

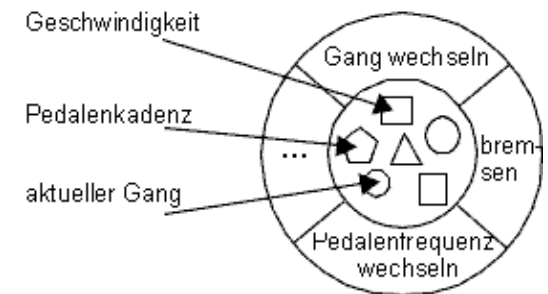
Verhalten

- Methoden
- Funktionalität
- Methodendeklaration

Eine Klasse



Fahrradklasse



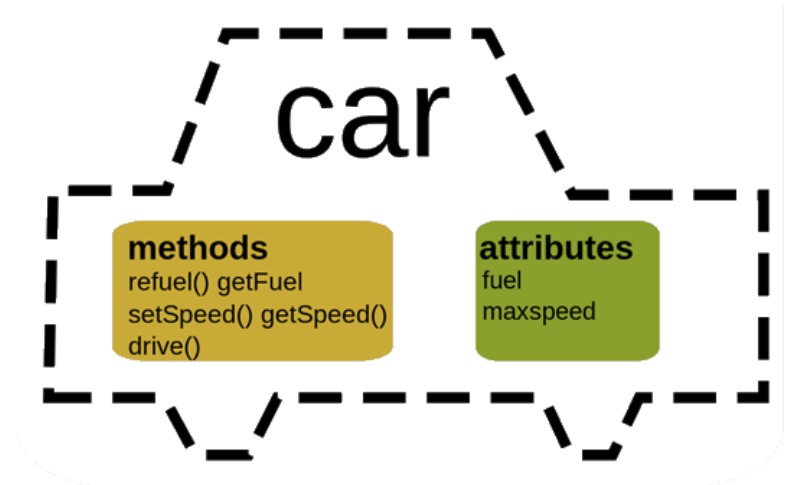
Status & Verhalten

- legt im Bauplan die Eigenschaften fest
- konkrete Objekte erhalten individuelle Werte beim Erzeugen des Objekts
 - Marke eines Autos
 - Farbe eines Autos
 - Anzahl der Türen
- legt bei der Definition der Klasse alle Funktionalitäten fest
 - -> was kann man mit diesem Objekt tun?
 - Fahren
 - Tanken

Status einer Klasse

Eigenschaften einer Klasse:

- Attribute
- Properties
- Get- und Set-Methoden



- Konzepte:
 - Information Hiding private Eigenschaften
 - Data Encapsulation wohldefinierte Schnittstellen
Datenkapselung

Attribute public vs private

Ohne Datenkapselung

```
class Car
{
    String name;
    int horsepower;
}
```

Attribute sind
private, wenn
nichts angegeben ist

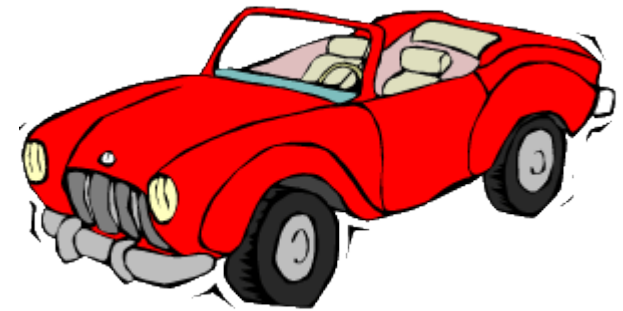
```
class Program
{
    static void Main(string[] args)
    {
        Car c1 = new Car();
        //Zugriff auf Attribute nicht möglich
        //Attribute sind private
        //c1.horsepower = 3;
        //c1.name = "BMW";
    }
}
```

```
class Car
{
    public String name;
    public int horsepower;
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Car c1 = new Car();
        //Zugriff auf Attribute
        c1.horsePower = 3;
        c1.name = "BMW";
    }
}
```


Auto mit öffentlichen Attributen

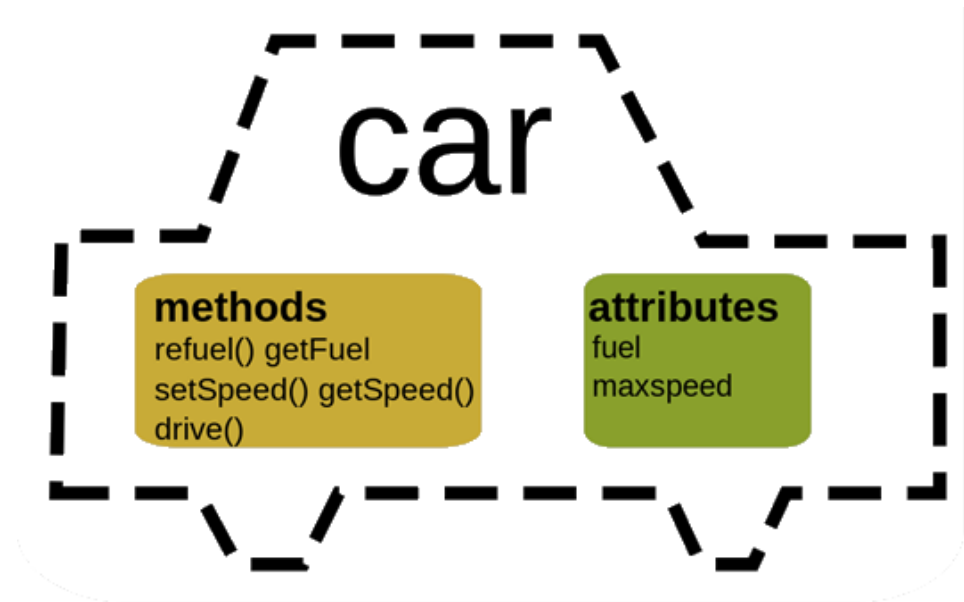
```
public class Car
{
    public String name;
    public int horsepower;
}
```



```
public class Program    {
    static void Main(string[] args)
    {
        Car c1 = new Car();
        //Zugriff auf Attribute
        c1.horsePower = 3;
        c1.name = "BMW";
    }
}
```

Sichtbarkeit von Attributen

- **public:**
 - Attribut kann zB: von der Main aus abgefragt bzw verändert werden
 - Element kann innerhalb der Klasse und in allen anderen Klassen verwendet werden
- **private:**
 - Attribut kann nur innerhalb der Klasse genutzt werden, es ist zB: von der Main aus nicht sichtbar
 - Element kann nur innerhalb der eigenen Klasse verwendet werden (dies ist Standard, wenn nichts angegeben ist = private)



Methoden

Funktionalität eines Objekts in den Methoden einer Klasse festlegen...

Methoden

- Klassen bestehen aus **Attributen** und **Methoden**
- Attribute **speichern** Werte einer Instanz
 - z.B. Seitenlänge = 3.5 bei einem Rechteck
- Methoden werden für eine Instanz aufgerufen
- Methoden führen **Berechnungen** auf der Instanz aus
 - greifen auf Attributwerte der Instanz zu
 - geben diese z.B. aus
 - berechnen z.B. den Flächeninhalt eines Rechtecks
 - ändern z.B. Attributwerte
 -
- Methoden realisieren das Prinzip der **Kapselung**

Methodendefinition

Syntax:

```
class Klassenname {  
    [modifier] Typ1 Variablenname1;  
    ...  
    [modifier] Rückgabetyp Methodenname (Parameterliste) {  
        //Programmcode der Methode  
    }  
}
```

Beispiel:

```
class Rechteck {  
    internal double breite, hoehe;  
    internal double Flaechen () {  
        double flaecheninhalt = breite * hoehe;  
        return flaecheninhalt;  
    }  
}
```

Beispiele für Methodendefinitionen

```
class Rechteck {  
    internal double breite, hoehe;    // Attribute  
  
    // Methode zur Berechnung des Umfangs  
    internal double umfang () {        // keine Parameter; Rückgabetyt double  
        double umfang;                // lokale Variable (ex. nur in der Methode)  
        umfang = 2*breite + 2*hoehe;  // Berechnung anhand der Attributwerte  
        return umfang;                // Rückgabe des berechneten Wertes  
    }  
  
    //Methode zum Ändern der Breite:  
    internal void aendereBreite(double neueBreite){  
        breite = neueBreite;  
    }  
  
    // Methode zur Ausgabe des Objekts (als Text)  
    internal void ausgeben() {        // keine Parameter; kein Rückgabewert  
        Console.WriteLine("Dieses Rechteck hat folgende Eigenschaften:");  
        Console.WriteLine("Breite: " + breite + "Höhe: " + hoehe);  
        Console.WriteLine("Umfang: " + umfang());        //Methodenaufruf!  
    }  
}
```

Aufruf von Methoden

Syntax:

// Methode ohne Rückgabewert (void)

instanzvariable.methode1(Parameterliste);

...

// Methode mit Rückgabewert

variable =

instanzvariable.methode2(Parameterliste);

Beispiel:

```
myRechteck.ausgeben();
```

```
double flaecheninhalt;
```

```
flaecheninhalt = myRechteck.flaeche();
```

Beispiele für Methodendefinitionen

```
class Rechteck {  
    internal double breite, hoehe;  
  
    internal double umfang () {  
        double umfang;  
        umfang = 2*breite + 2*hoehe;  
        return umfang;  
    }  
  
    internal void aendereBreite(double neueBreite){  
        breite = neueBreite;  
    }  
  
    internal void ausgeben() {  
        Console.WriteLine("Dieses Rechteck hat folgende Eigenschaften:");  
        Console.WriteLine("Breite: " + breite + "Höhe: " + hoehe);  
        Console.WriteLine("Umfang: " + umfang());  
    }  
}
```

//Aufruf (dies steht in einer anderen Datei...)

```
Rechteck r = new Rechteck();
```

```
r.aendereBreite(5.2);
```

```
r.ausgeben();
```


Methoden mit und ohne Rückgabewert

Ohne Rückgabewert

- Deklaration:
`void methode(){`
.....
`}`
- Aufruf
`instanz.methode();`
- Methode muss kein „return“ haben

Mit Rückgabewert

- Deklaration:
`double methode(){`
...
`return double_Wert;`
`}`
- Aufruf:
`double d = instanz.methode();`
- Methode muss „return Wert“ haben;
- Typ des Wertes = Rückgabebetyp
- Zweck: Liefert Ergebnis nach außen

Schlüsselwörter: ref - out - params

- **Schlüsselwort** **ref:**
Referenzübergabe (engl.: Call by Reference)
- **Schlüsselwort** **out:**
Ausgangsparmeter
- **Schlüsselwort** **params:**
Übergabe einer beliebigen Anzahl von Argumenten

Methodenüberladung

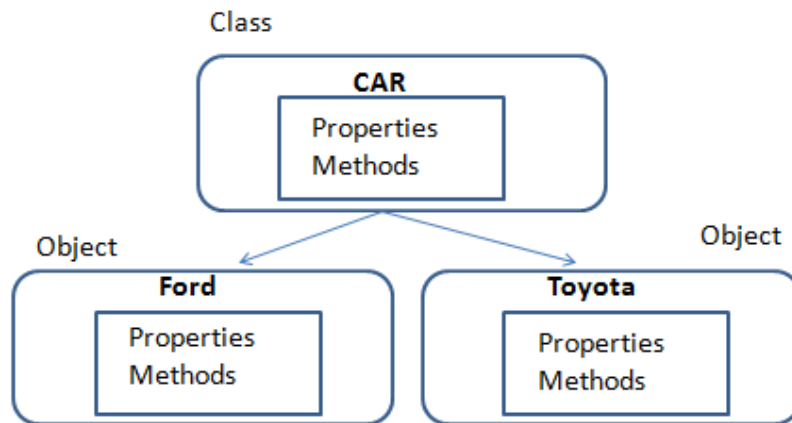
- Mehrere gleichnamige Methoden, die sich in der Parameterliste unterscheiden.
- Wird eingesetzt wenn Basisfunktionalität sich von unterschiedlichem Coding bereitgestellt werden soll.
 - Gleichnamige Methoden
 - Anzahl der Parameter unterschiedlich
 - Gleiche Parameteranzahl aber mindestens ein Parameter vom Typ anders
 - Typgleiche Parameter, aber einmal mit `out` oder `ref` definiert



Konstruktor

Aufgerufen bei der Erstellung einer Instanz

Konstruktor - Objekt Ford & Toyota



```
Car c2;  
c2 = new Car(„Ford“);
```

```
Car c3 = new  
Car(„Toyota“);
```

Konstrukturen: Motivation

- Beispiel: Erzeugen eines Dreiecks mit drei Attributen `seite1`, `seite2`, `seite3`

```
Dreieck d;  
d = new Dreieck();  
d.seite1 = 6.4;  
d.seite2 = 3.1;  
d.seite3 = 5.2;
```

- Mit Konstruktor:

```
Dreieck d;  
d = new Dreieck(6.4 , 3.1, 5.2);
```

Konstrukturen sparen Zeit

- Ohne Konstruktor

```
static void Main(string[] args)
{
    Car Ford = new Car();
    Ford.model = "Mustang";
    Ford.color = "red";
    Ford.year = 1969;

    Car Opel = new Car();
    Opel.model = "Astra";
    Opel.color = "white";
    Opel.year = 2005;

    Console.WriteLine(Ford.model);
    Console.WriteLine(Opel.model);
}
```

- Instanziierung mit Parameterübergabe im Konstruktor:

```
class Program
{
    static void Main(string[] args)
    {
        Car Ford = new Car("Mustang", "Red", 1969);
        Car Opel = new Car("Astra", "White", 2005);

        Console.WriteLine(Ford.model);
        Console.WriteLine(Opel.model);
    }
}
```

Konstruktoren - Details

- Konstruktoren sind spezielle **Methoden**
- Name des Konstruktors = Klassenname
 - Beispiel: Konstruktor für die Klasse *Punkt* mit zwei Attributen *xKoordinate* und *yKoordinate*

```
public Punkt(double x, double y) {  
    xKoordinate = x;  
    yKoordinate = y;  
}
```
 - werden **implizit bei der Erzeugung** mit `new` aufgerufen:

```
Punkt p = new Punkt(4.9, 8.5);
```
 - kein Rückgabewert (auch kein `void`)
 - darf fehlen (dann wird Standardkonstruktor verwendet)
- Zweck:
 - Initialisierung der Instanz,
 - Zuweisung von Anfangswerten

Konstrukturen

- Es kann mehrere Konstruktoren geben
- Diese müssen sich hinsichtlich Anzahl und Typ der Parameter unterscheiden

- **Beispiel:**

```
class Kreis{
    internal double mittelpunktX, mittelpunktY, radius;
    public Kreis() {
        mittelpunktX = 0; mittelpunktY = 0; radius = 1;}
    public Kreis(double r){
        mittelpunktX = 0; mittelpunktY = 0; radius = r;}
    public Kreis(double r, double x, double y){
        mittelpunktX = x; mittelpunktY = y; radius = r;}
}
}
```

Aufruf:
Kreis k = new Kreis();

Aufruf:
Kreis k = new Kreis(56);

Aufruf:
Kreis k = new Kreis(56,8,7);

Namenskonflikte mit „this“ lösen

- this ist ein Schlüsselwort, das **auf die aktuelle Instanz verweist**, und bietet sich an, um auf Methoden oder Eigenschaften des aktuellen Objekts zuzugreifen.
- ```
class ClassA {
 public int myValue;

 public void Method1(int myValue) {
 this.myValue = myValue;
 }
}
```

# This Schlüsselwort

```
class Car
{
 private string name;
 private int horsepower;

 public void SetHorsePower(int horsepower)
 {
 if (horsePower > 0)
 this.horsePower = horsepower;
 else
 Console.WriteLine("Falsche Eingabe");
 }
}
```

# Car - Konstruktor ohne Parameter

```
// Create a Car class
class Car
{
 public string model; // Create a field

 // Create a class constructor for the Car class
 public Car()
 {
 model = "Mustang"; // Set the initial value for model
 }

 static void Main(string[] args)
 {
 Car Ford = new Car(); // Create an object of the Car Class (this will call the constructor)
 Console.WriteLine(Ford.model); // Print the value of model
 }
}

// Outputs "Mustang"
```

# Car - Konstruktor mit Parameter

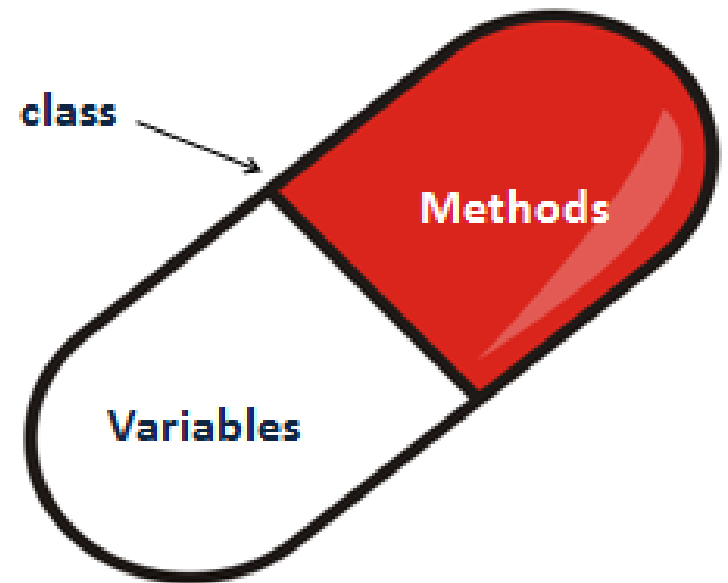
```
class Car
{
 public string model;

 // Create a class constructor with a parameter
 public Car(string modelName)
 {
 model = modelName;
 }

 static void Main(string[] args)
 {
 Car Ford = new Car("Mustang");
 Console.WriteLine(Ford.model);
 }
}

// Outputs "Mustang"
```

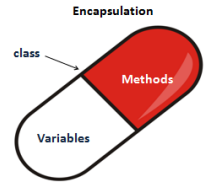
## Encapsulation



# Datenkapselung

- Verstecken von internen Informationen
  - -> Information Hiding
- Mit Hilfe von
  - -> Properties bzw. Get-Set-Methoden

# Information Hiding



- Attribute sollten standardmäßig alle private sein, damit von außen keine beliebige Veränderung stattfinden soll

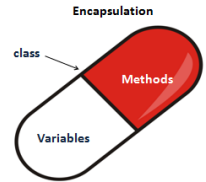
## Problem?

- keine Veränderung des Zustandes möglich?

## Wunsch:

- Zustand eines Objektes sollte änderbar sein
- Zustand eines Objektes sollte immer valide sein

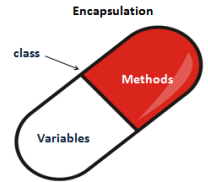
# Lösung: Datenkapselung



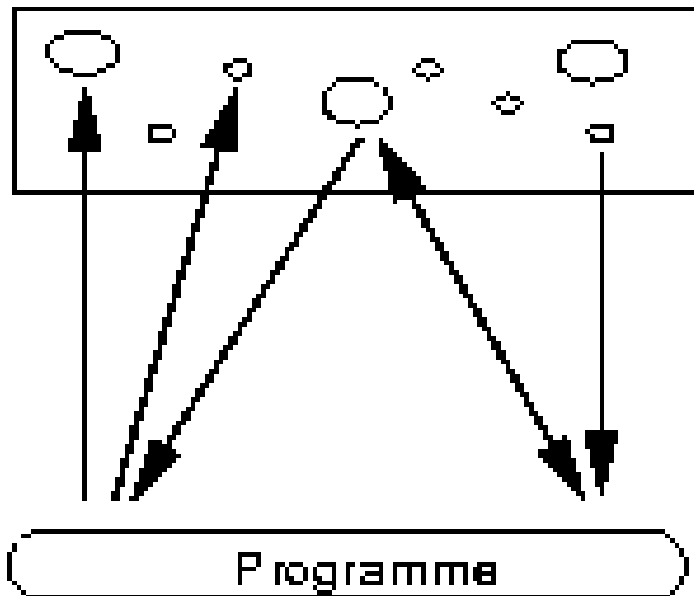
- Verstecken von internen Informationen:
  - Information Hiding
  - private Attribute
- Zugriff nur über wohldefinierte Schnittstellen
  - Properties bzw. Get-Set-Methoden
  - Öffentliche Zugriffsmöglichkeit mit Wertüberprüfung & ev. eingeschränkten Zugriffsrechten
- Genannt wird dieses Konzept Datenkapselung es wird in Folge ausführlich erklärt



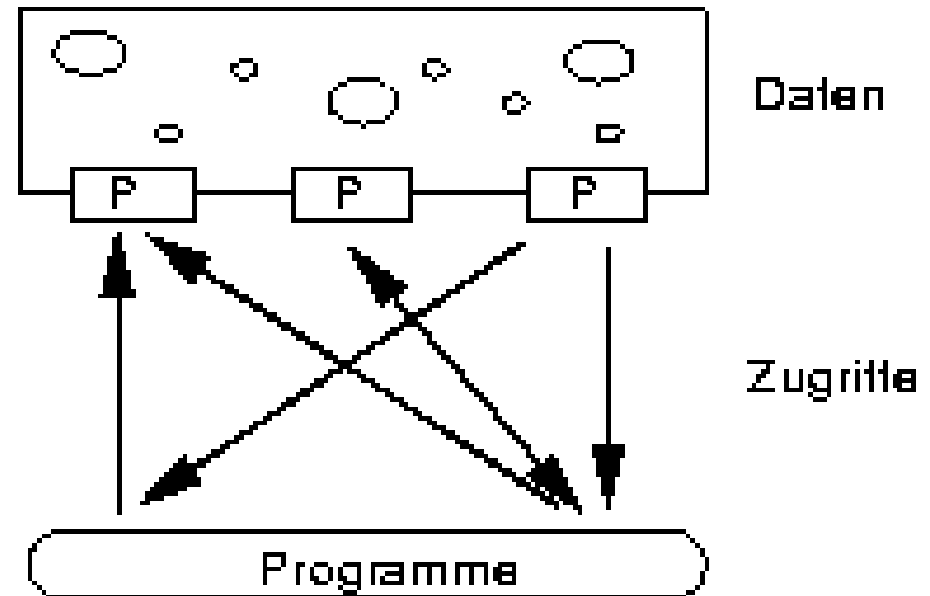
# Datenkapselung



- Private Attribute plus
- Zugriff über Wohldefinierte Schnittstelle

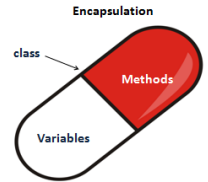


freier Datenzugriff

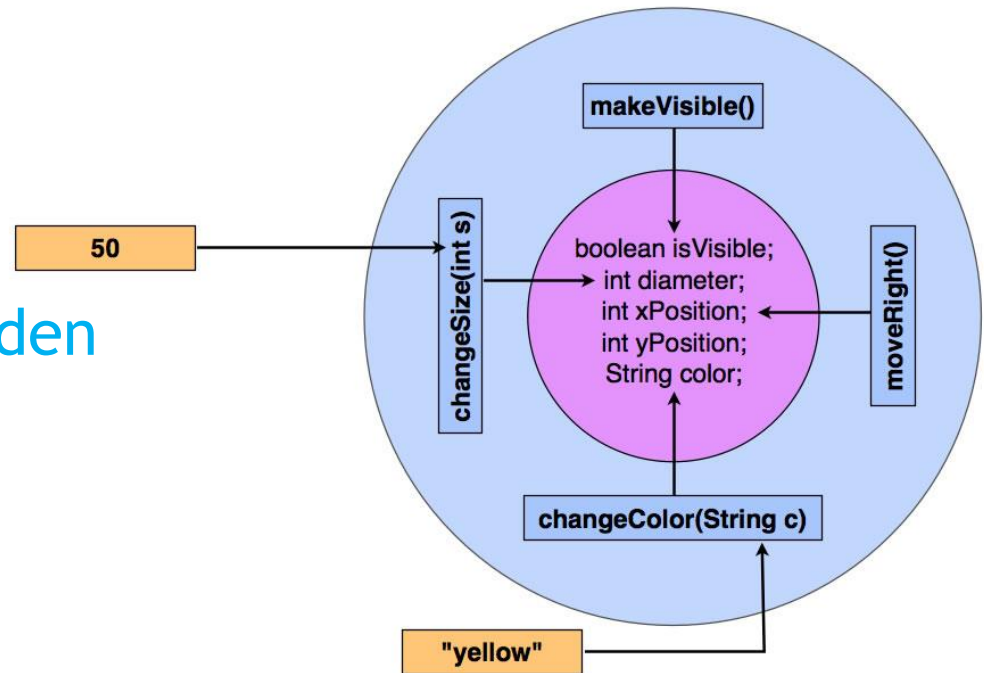


Datenzugriff über Prozeduren (P)

# Information Hiding

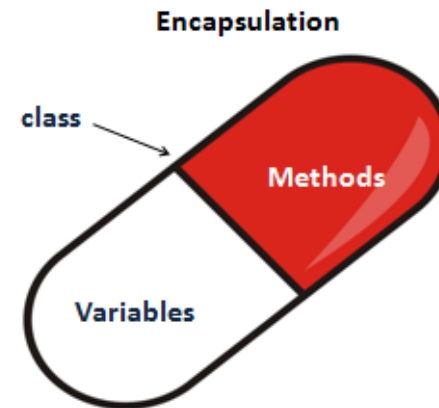


- Attribute sind **private** zu deklarieren und die Deklaration sollte **explizit** stattfinden
- Attribute sollten **private** gesetzt sein
- Änderungen mit **Methoden** möglich:
  - `changeSize(int s)`
  - `changeColor(String c)`



# Kapselung

- Private Attribute
- Zugriff von außen nur über Methoden

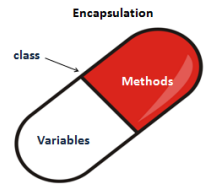


- Mechanismen, um dies sicherzustellen:
  - `private` und `protected` Modifizierer statt `public` und `internal` für Attribute

# Wozu Kapselung?

```
public class Car
{
 public String name;
 public int horsepower;
}
```

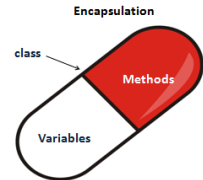
```
class Car {
 private String name;
 private int horsepower;
 public void SetHorsePower(int hp) {
 if (hp > 0)
 horsepower = hp;
 else
 CW("Falsche Eingabe");
 }
}
```



```
//ohne Kapselung:
Car c1 = new Car();
c1.name = "BMW";
c1.horsePower = -3;
// horsepower negativ!
```

```
//mit Kapselung:
Car c1 = new Car();
c1.name = "BMW";
//radius nie negativ!
c1.SetHorsePower(-3)
```

# Eigenschaften -> Status einer Klasse

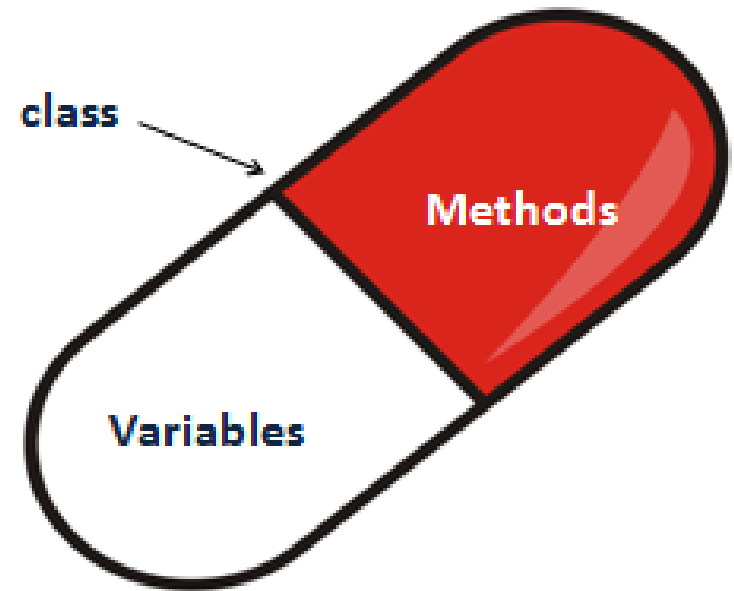


- Standardmäßig als private festgelegt
- Eigene Methoden zum Verändern der Werte die geändert werden dürfen:
  - Datentyp GetValue()
  - void SetValue(Datentyp val)
- **Properties:**

Sind lese und schreibgeschützte Eigenschaften:

  - Nutzung wie bei public Variablen
  - Sicherheit wie bei Get- und Set- Methoden
  - Instanz.PropertyName = Value;

## Encapsulation

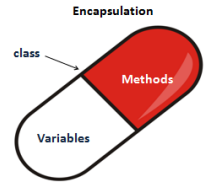


## Properties

C# bietet für Datenkapselung eine Schnellschreibweise an:

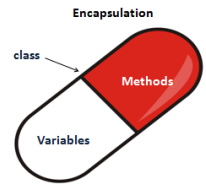
- Statt der Verwendung von Get- und Set-Methoden werden Properties verwendet.

# Eigenschaften - Properties



- Eigenschaften verhalten sich wie öffentliche Variablen einer Klasse bzw. eines Objektes
- Die Syntax ist wesentlich kompakter
- Die Zuweisung eines Werts erfolgt wie bei lokalen Variablen: mit dem „=“-Operator
- Man unterscheidet :
  - Setzen eines Wertes
  - Auslesen eines Wertes

# Get/Set Methoden für Wertüberprüfung ersetzen mit Properties...

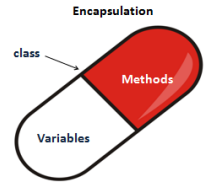


```
class Kreis{
 private double radius; //Instanzvariable

 public double GetRadius()
 {
 return radius;
 }
 public void SetRadius(double radius) //Parameter
 {
 if(neuerRadius > 0)
 this.radius=radius;
 else
 this.radius=0;
 }
}
```



# Syntax für Properties



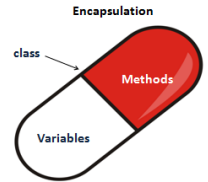
- **Langschreibweise:**

Modifizierer Datentyp Bezeichner

```
{
 [Modifizierer] get { return bezeichner; }
 [Modifizierer] set { bezeichner = value; }
}
```

- Optional  
kann eine Wertüberprüfung getätigt werden

# Eigenschaften - Properties

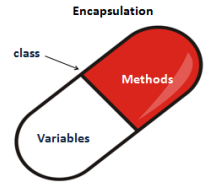


Beispiel:

```
class Kreis{
 private double radius; //Instanzvariable

 public double Radius { //Eigenschaft
 get {
 return this.radius;
 }
 set { if(value > 0)
 this.radius=value; //value enthält den
 else //zuzuweisenden Wert
 this.radius=0;
 }
 }
}
```

# Syntax für Properties

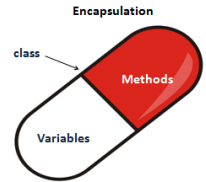


- **Schnellschreibweise**
  - Kein privates Attribut anlegen
  - Wird automatisch im Hintergrund erzeugt
- Public Property erzeugen, ohne Überprüfungsmöglichkeit:

Modifizierer Datentyp Bezeichner

```
{
 [Modifizierer] get;
 [Modifizierer] set;
}
```

# Properties ohne Wertüberprüfung



## Schnellschreibweise

- legt intern eine eigene Variable an und erzeugt den dazugehörigen Verweis darauf
- erzeugt das Property ohne Wertüberprüfung.

```
class Kreis{

 public double Radius { get ; set ; }
}
```

## Vorsicht

- Private Eigenschaft plus Property mit Schnellschreibweise führt zu einem Fehlerfall, die Variable wird doppelt erstellt!



# Zugriffsmodifizierer

Sichtbarkeiten von Klassen und deren Mitgliedern

# Zugriffsmodifizierer der Klassen

- Regelt wer mit der Klasse arbeiten darf
  - **internal:**  
die Klasse kann innerhalb des eigenen Programms verwendet werden
  - **public:**  
die Klasse kann aus jeder beliebigen Anwendung heraus instanziiert werden
- Zugriffsmodifizierer ist optional,  
nicht angegeben entspricht **internal** bei Klassen

# Sichtbarkeit der Mitglieder einer Klasse: Attribute & Methoden

- **public:**
  - Element kann innerhalb der Klasse und in allen anderen Klassen verwendet werden.
- **protected:**
  - Element kann in der eigenen, und in Klassen die von dieser abgeleitet sind (siehe Kapitel Vererbung) verwendet werden.
- **internal:**
  - Element kann in der eigenen und in allen Klassen, die dem gleichen Programm angehören verwendet werden.
- **private:**
  - Element kann nur innerhalb der eigenen Klasse verwendet werden (dies ist Standard, wenn nichts angegeben ist = private)

# Sichtbarkeit für Klassenelemente

AS - Assembly

| <b>Zugriffsmodifikator</b> | <b>Außerhalb</b>      | <b>Kinderklassen</b>  |
|----------------------------|-----------------------|-----------------------|
| public                     | Ja                    | Ja                    |
| internal                   | Ja (innerhalb der AS) | Ja (innerhalb der AS) |
| internal protected         | Ja (innerhalb der AS) | Ja                    |
| protected                  | Nein                  | Ja                    |
| private                    | Nein                  | Nein                  |





# Konstruktorkette

Innerhalb einer Klasse

Innerhalb der Vererbungshierarchie

# Konstruktorkette



- Innerhalb einer Klasse kann es mehrere Konstruktoren geben mit unterschiedlichen Parametern.
  - Genannt Konstruktorenüberladung
  - Es wird je ein Wert in der Parameterliste angefügt und alle Werte werden initialisiert.
  - Es entsteht Code Verdoppelung...

# Doppelter Quellcode...



```
public Student() { }
```

```
public Student(String name)
{
 this.Namen = name;
}
```

```
public Student(String name, int age)
{
 this.Namen = name;
 this.Age = age;
}
```

# Auflösung von Copy/Paste



```
public Student() { }
```

```
public Student(String name): this()
{
 this.Namen = name;
}
```

```
public Student(String name, int age): this(name)
{
 this.Age = age;
}
```

# Konstruktorenkette



```
public School()
{
 studentCount = 0;
 SetAmount(100);
}
public School(String name) : this()
{
 SetName(name);
}
public School(string name, String description) : this(name)
{
 this.description = description;
}
```

# Konstruktorkette - Bsp Schule



```

public class School
{
 private int amount;
 private int studentCount;

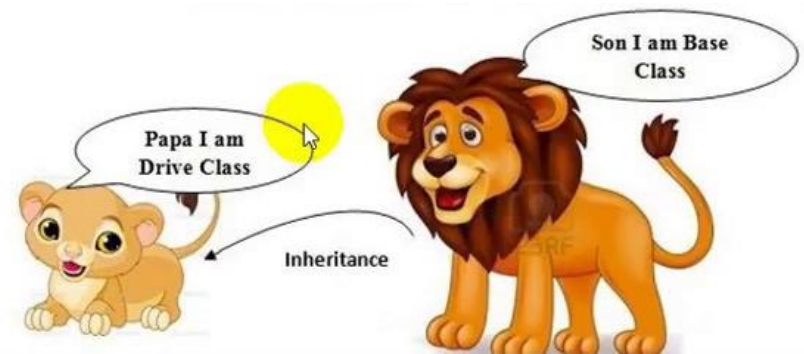
 private String name;
 private String description;

 public Student[] students;

#region Konstruktor
 public School()
 {
 studentCount = 0;
 SetAmount(100);
 }
 public School(String name) : this()
 {
 SetName(name);
 }
 public School(string name, String description) : this(name)
 {
 this.description = description;
 }
#endregion

#region Getter/Setter Methoden
 public String GetName()...
 public void SetName(String name)...
 public int GetAmount(int amount)...
 public void SetAmount(int amount)
 {
 if (amount <= 0)
 Console.WriteLine("Es darf keine neg Zahl gesetzt werden.");
 else
 {
 this.amount = amount;
 students = new Student[amount];
 }
 }
#endregion
}

```

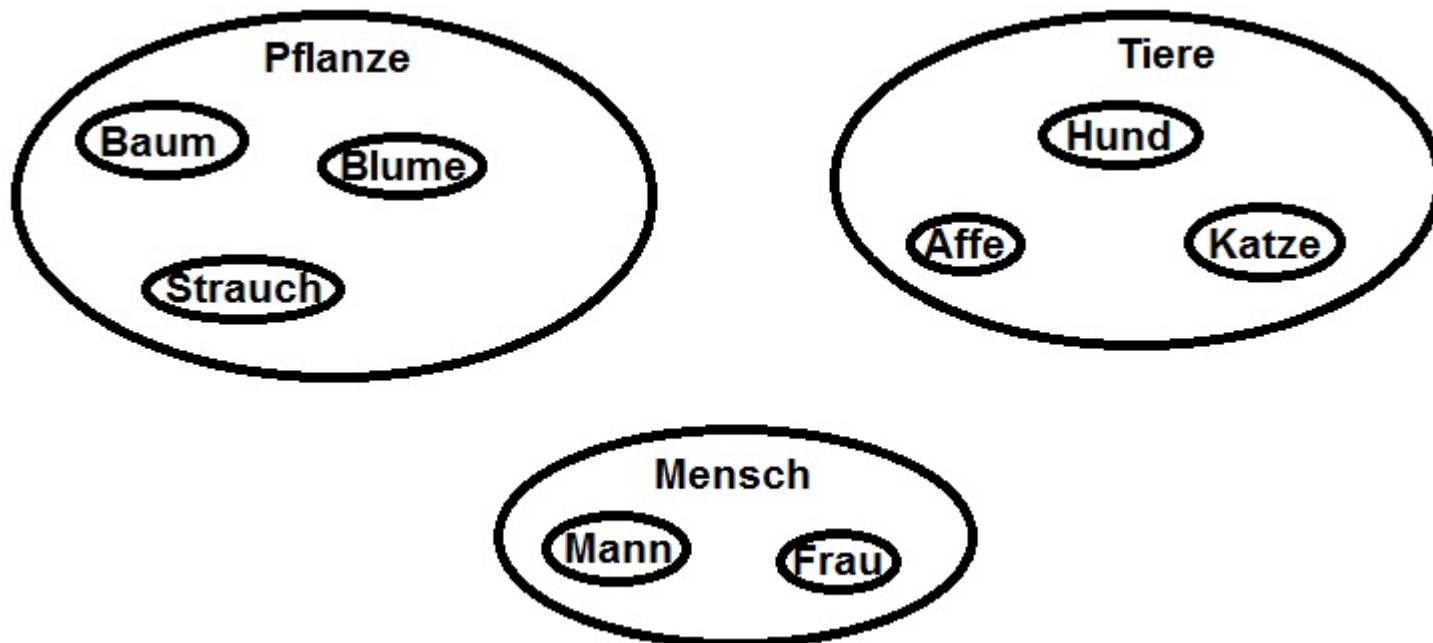


# Vererbung

Objekte können Eigenschaften von anderen Objekte übernehmen und gegebenenfalls modifizieren. Der Vererbungsmechanismus ist insbesondere dann von Bedeutung, wenn Objekte mit Hilfe von Klassen beschrieben werden.

# Denken in Objekten

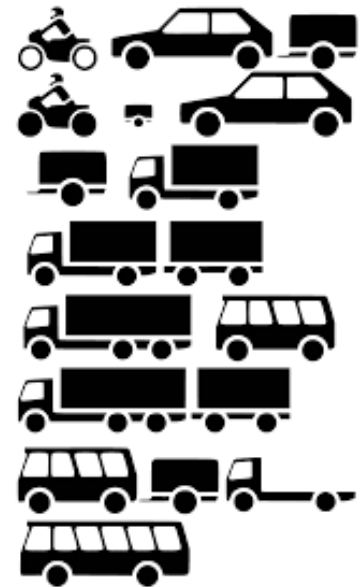
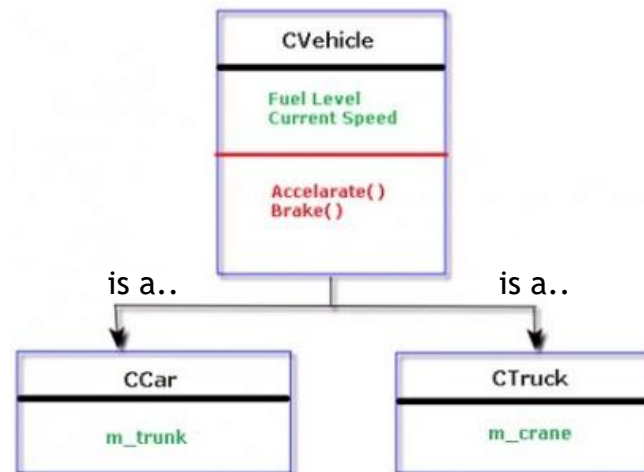
- Alles ist ein Objekt - zusammenfassen von Objekten zu Klassen





# Vererbung - Generalisieren

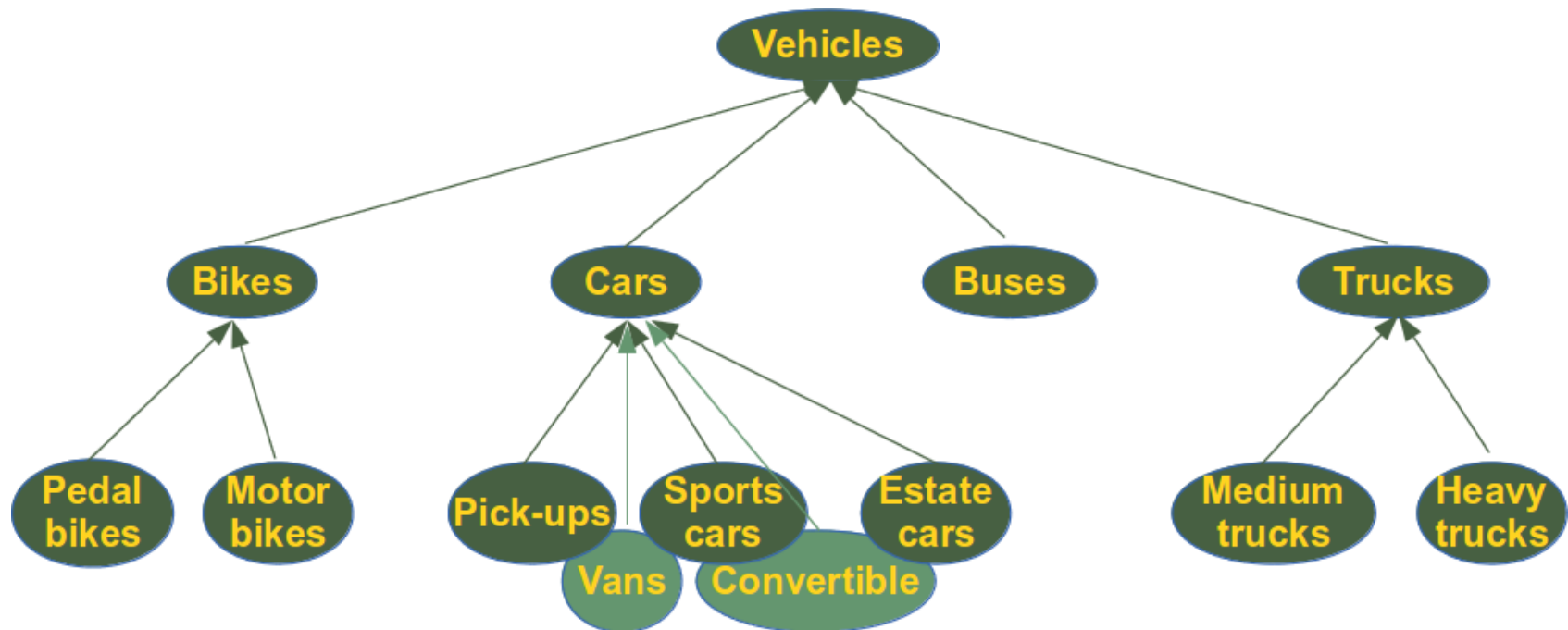
- Gemeinsamkeiten finden  
genannt Generalisieren:
  - Zusammenfassen  
gemeinsamer Eigenschaften
  - mehrerer Unterklassen  
erhalten eine gemeinsamen Basisklasse
- Fahrzeuge
  - Zweirad
  - PKW
  - LKW
  - Bus



# Vererbung - Spezialisierung

**Spezialisieren** - wir suchen zum Überbegriff Fahrzeuge alle möglichen konkreten Unterbegriffe.

Wir bringen diese in eine Hierarchie: Vererbungsbaum



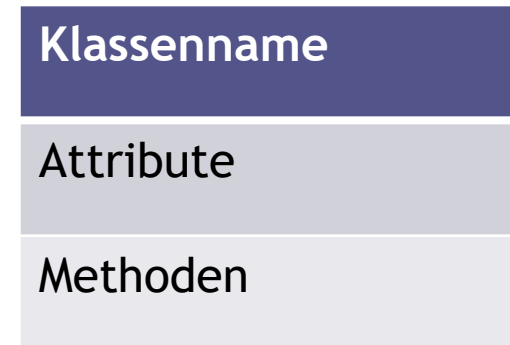
# Vererbung

- Basisklasse:
  - eine Klasse, die ihre Member als Erbgut einer abgeleiteten Klasse zur Verfügung stellt, wird als **Basisklasse** bezeichnet.
  - gibt ihre Elemente weiter (vererbt sie)
- abgeleitete Klasse:
  - die erbende Klasse ist die Subklasse oder einfach nur **die abgeleitete Klasse**.
  - nimmt die Elemente in Empfang (erbt sie)
  - Subklassen (abgeleitete Klassen) verfügen normalerweise über **mehr Funktionalitäten** als ihre Basisklassen.

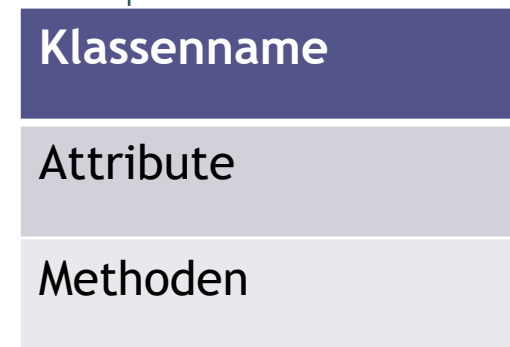
# Vererbung - UML Darstellung

- Unified Modeling Language dient zur Darstellung von Klassen
- Klassenname, Attribute und Methoden werden gegliedert und aufgelistet.
- Vererbungspfeil ist ein Dreieck, der zur Basisklasse zeigt.

Basis  
klasse



Unter  
klasse



# protected

- Um auf ein Attribut der Basisklasse von der abgeleiteten Klasse aus zugreifen zu können

```
class Car
{
 protected String name;
 //Methoden
}
```

# base Schlüsselwort für Attribute



- Mit base.
- kann auf alle Mitglieder der direkten Basisklasse Bezug genommen werden,
  - solange sie nicht als private deklariert sind.

```
class Car
{
 protected String name;
 protected int horsepower;

 public void SetHorsePower(int horsepower) ...
}
class SpecialCar: Car
{
 private String feature;
 SpecialCar(String name, String feature)
 {
 this.feature = feature;
 base.name = name;
 }
}
```

# base für Methodenaufrufe

- ```
class BaseClass {  
    public void TestMethod() {  
        Console.WriteLine("In 'BaseClass.TestMethod() '");  
    }  
}
```

```
class SubClass1 : BaseClass {}
```

```
class SubClass2 : SubClass1 {  
    public void BaseTest() {  
        base.TestMethod();  
    }  
}
```

- Ein umgeleiteter Aufruf an eine indirekte Basisklasse ist nicht gestattet
// unzulässiger Aufruf

```
base.base.TestMethod();
```

Automatische Konstruktorkette

Konstruktor der Basisklasse



Konstruktorkette zur Basisklasse



- Beim erstellen eines Objekts
- wird immer der parameterlose Konstruktor der Basisklasse aufgerufen
 - Bis zur Basisklasse Objekt (Basisklasse aller Klassen)
- anschließend wird der Quellcode des eigenen Konstruktors ausgeführt.

Konstruktoren in abgeleiteten Klassen



- Bei der Erzeugung des Objekts einer Subklasse gelten dieselben Regeln wie beim Erzeugen des Objekts einer Basisklasse:
 - Es wird generell ein Konstruktor aufgerufen.
 - Die Subklassenkonstruktoren dürfen überladen werden.
 - Konstruktor **werden grundsätzlich nicht von der Basisklasse an die Subklasse vererbt**. Daher müssen alle notwendigen Konstruktoren in einer abgeleiteten Klasse definiert werden
- Mit **: base()** kann ein Konstruktor der Basisklasse aufgerufen werden.

Konstruktor der Basisklasse mit Parameter verwenden



```
//Klasse Person
```

```
public Person(String name)
{
    this.Name = name;
}
```

```
//Klasse Student
```

```
//Leitet auf den Konstruktor der Basisklasse Person weiter
```

```
public Student(String name): base(name) { }
public Student(String name, int age): base(name)
{
    this.Age = age;
}
```

this & base bei Konstruktorenkette



```
public class Circle {  
    double radius;  
    public Circle() { }  
    public Circle(double radius) { this.radius = radius; }  
}  
public class GraphicCircle : Circle {  
    int xPos; int yPos;  
    public GraphicCircle() { }  
  
    public GraphicCircle(double radius) : base(radius) { }  
  
    public GraphicCircle(double radius, int xPos, int yPos) : this(radius)  
    {  
        this.xPos = xPos;  
        this.yPos = yPos;  
    }  
}
```

Konstruktorkette Circle

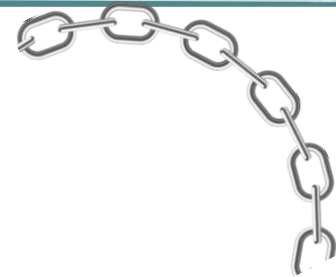


Erstelle die Ausgabe für folgenden Quellcode, wenn jeder Konstruktor ein CW erhält...

```
Console.WriteLine("Circle Test:");  
Circle c = new Circle();  
c = new Circle(3.4);
```

```
Console.WriteLine("GraficalCircle Test:");  
GraphicCircle gc = new GraphicCircle();  
gc = new GraphicCircle(4.4);  
gc = new GraphicCircle(5.5, 2, 2);
```

Circle & GraphicCircle



```
public class GraphicCircle : Circle
{
    int xPos;
    int yPos;
    public GraphicCircle()
    {
        Console.WriteLine("GraphicCircle()");
    }

    // ruft Konstruktor der Basisklasse auf
    public GraphicCircle(double radius) : base(radius)
    {
        Console.WriteLine("GraphicCircle(double radius) : base(radius)");
    }

    // ruft Konstruktor mit einem Parameter in dieser Klasse auf
    public GraphicCircle(double radius, int xPos, int yPos) : this(radius)
    {
        this.xPos = xPos;
        this.yPos = yPos;
        Console.WriteLine("GraphicCircle(double radius, +" +
            "int xPos, int yPos) : this(radius)");
    }
}
```

```
public class Circle
{
    double radius;
    public Circle() { Console.WriteLine("Circle()"); }
    public Circle(double radius)
    {
        this.radius = radius;
        Console.WriteLine("Circle(double radius)");
    }
}

static void Main(string[] args)
{
    Console.WriteLine("Circle Test:");
    Circle c = new Circle();

    c = new Circle(3.4);
    Console.WriteLine();

    Console.WriteLine("GraficalCircle Test:");
    Console.WriteLine("-----");
    GraphicCircle gc = new GraphicCircle();
    Console.WriteLine("-----");
    gc = new GraphicCircle(4.4);
    Console.WriteLine("-----");
    gc = new GraphicCircle(5.5, 2, 2);
    Console.WriteLine("-----");
}
```

Ausgabe der Verkettung



```
static void Main(string[] args)
{
    Console.WriteLine("Circle Test:");
    Circle c = new Circle();

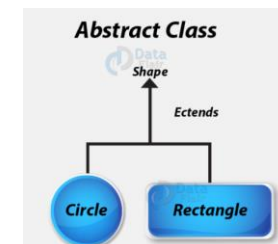
    c = new Circle(3.4);
    Console.WriteLine();

    Console.WriteLine("GraficalCircle Test:");
    Console.WriteLine("-----");
    GraphicCircle gc = new GraphicCircle();
    Console.WriteLine("-----");
    gc = new GraphicCircle(4.4);
    Console.WriteLine("-----");
    gc = new GraphicCircle(5.5, 2, 2);
    Console.WriteLine("-----");
}
```

```
C:\Windows\system32\cmd
GraficalCircle Test:
-----
Circle()
GraphicCircle()
-----
Circle(double radius)
GraphicCircle(double radius) : base(radius)
-----
Circle(double radius)
GraphicCircle(double radius) : base(radius)
GraphicCircle(double radius, int xPos, int yPos)
Drücken Sie eine beliebige Taste . . .
```



Form: Circle, Triangle, Rectangle, ...



Abstrakte Klasse



- können nicht instantiiert werden
- stellen eine Basisklasse für weitere abgeleitete Klassen zur Verfügung
- stellen Attribute und Methoden für alle abgeleiteten Klassen zur Verfügung

```
public abstract class A
{
    // Class members here.
}
```

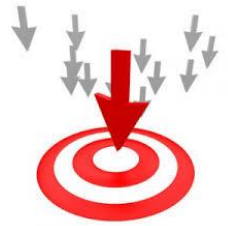
Abstrakte Klasse



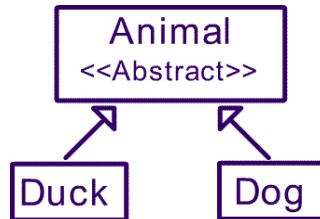
- können auch abstrakte Elemente haben
- mit Hilfe des Schlüsselwort **abstract** vor dem Rückgabewert

```
public abstract class A
{
    public abstract void DoWork(int i);
}
```

Abstrakte Klasse: Namensgebung



- bei Abstrakten Klassen



- wird ein A vor dem Klassennamen hinzugefügt:
 - AAnimal
 - AForm
 - ABuilding
 - ...

```
public abstract class AAnimal
{
    private int age = 3;
    1-Verweis
    public int Age
    {
        get { return age; }
        set {
            if (value > 0)
                age = value;
        }
    }

    0 Verweise
    public void PrintAge() ...

    0 Verweise
    public virtual void Running() ...
}
```

Instanziieren nicht möglich



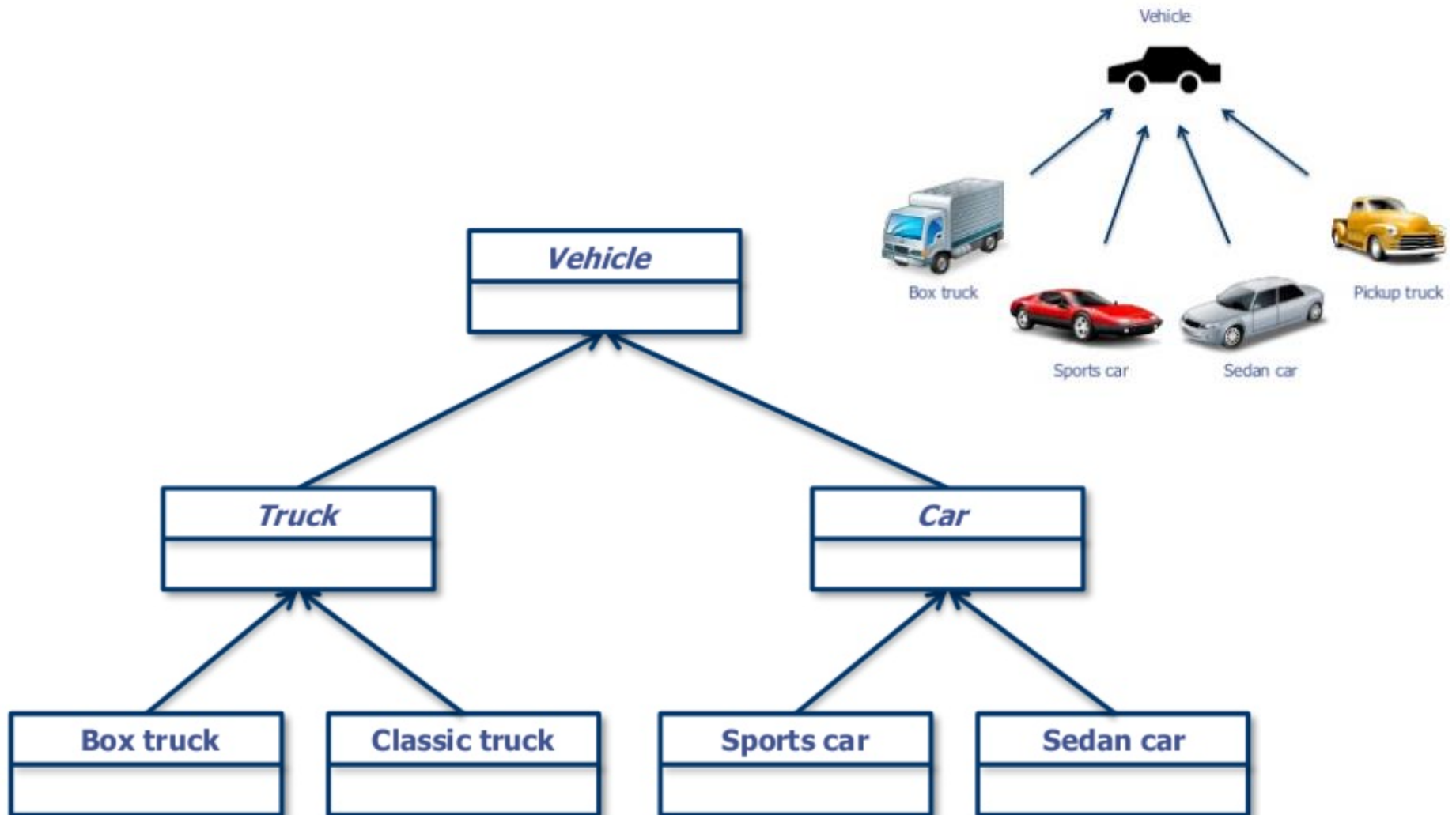
- Es kann keine Instanz einer abstrakten Klasse erstellt werden

```
static void Main(string[] args)
{
    AAnimal a = new AAnimal();
    AAnimal dumbo = new Elefant();
    AAnimal jerry = new Mouse();
    Console.WriteLine("Alter von dumbo: ");
    dumbo.PrintAge();
    dumbo.Running();
    Console.WriteLine("Alter von jerry: ");
    jerry.PrintAge();
    jerry.Running();
    jerry = dumbo;
    jerry.Running();
    Console.ReadKey();
}
```

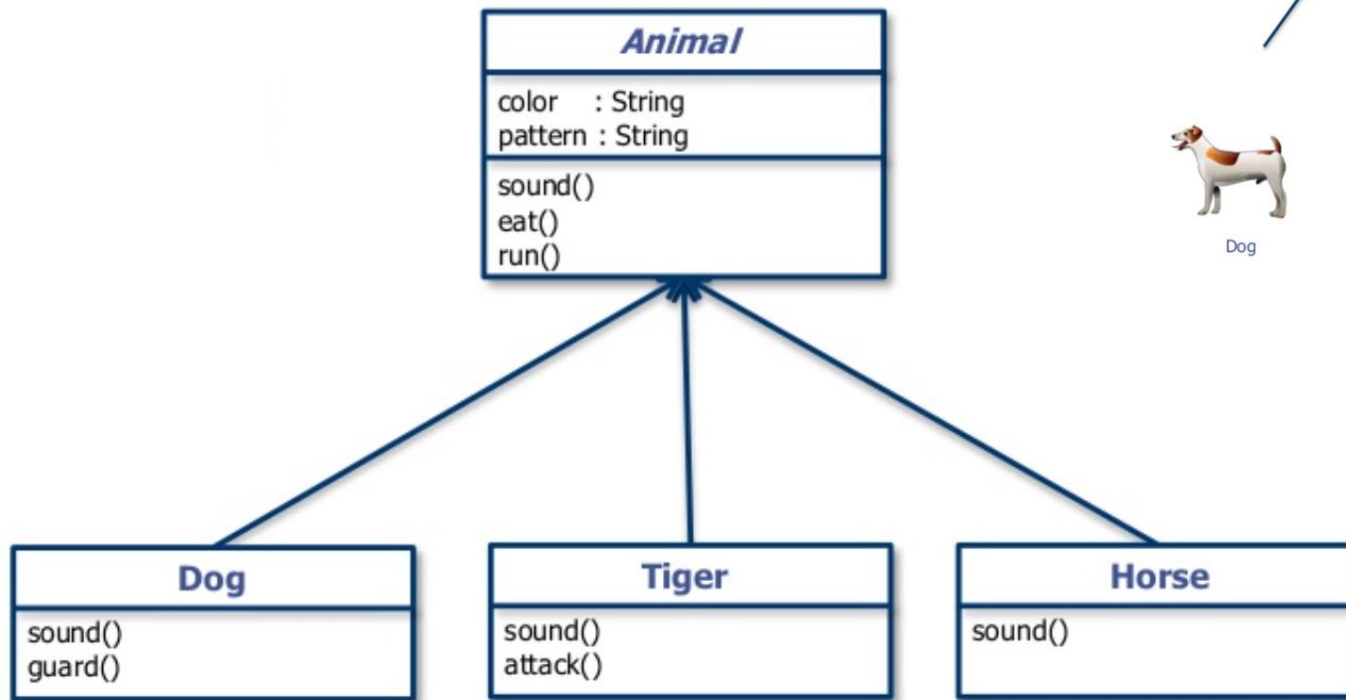
```
public class Mouse : AAnimal
{
    0 Verweise
    public void Running() ...
}
```

```
public class Elefant : AAnimal
{
    0 Verweise
    public void Running() ...
}
```

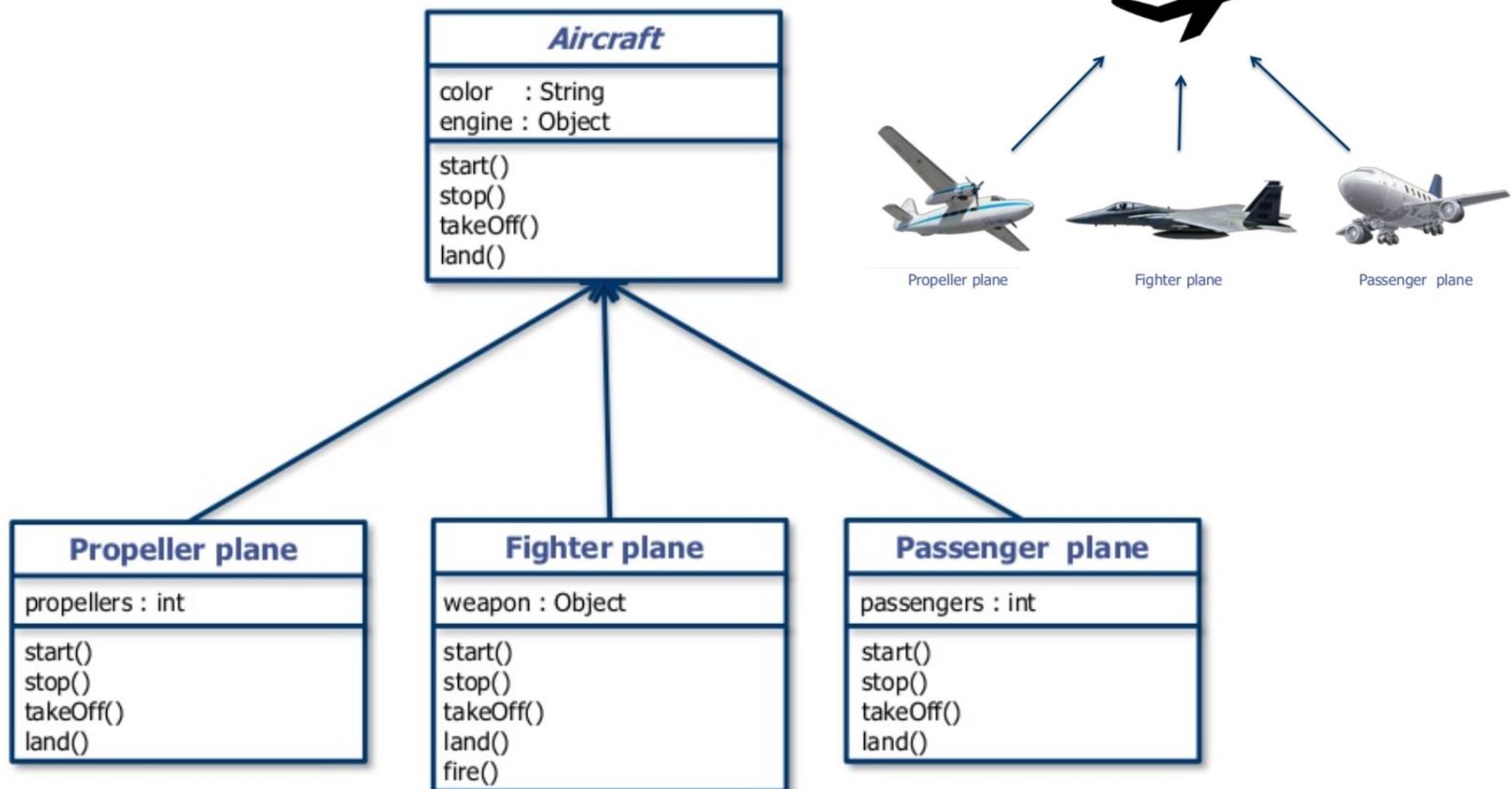
Abstrakte Klasse Vehicle



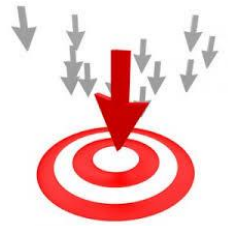
Abstrakte Klasse Animal



Abstrakte Klasse Aircraft

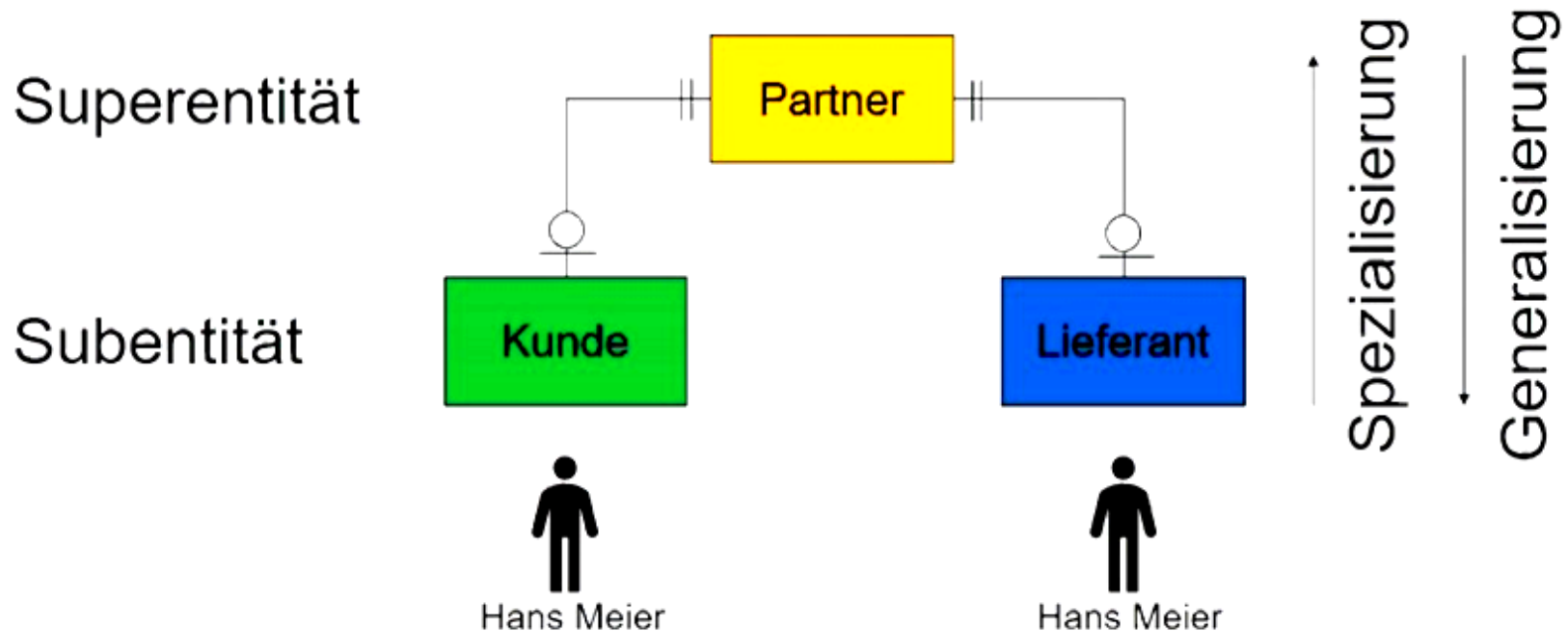


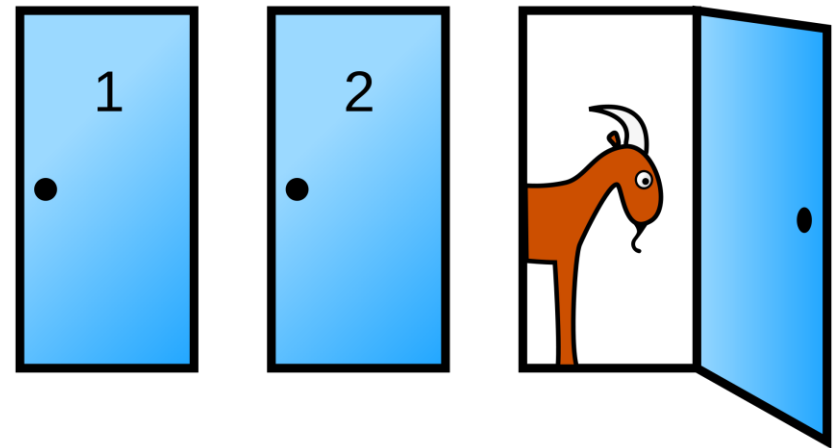
Generalisieren vs Spezialisieren



- Generalisieren:
 - Überbegriff für verschiedene Begriffe finden
 - Basisklasse für Subklassen finden
 - Hund Katze Pferd -> Tiere
- Spezialisieren:
 - Von einem Überbegriff mehrere Subbegriffe finden
 - Konkrete Subklassen finden die alle von einer Basisklasse erben
 - Kleidung: T-Shirt, Hose, Hemd, Bluse, Pullover, ...

Generalisierung vs Spezialisierung



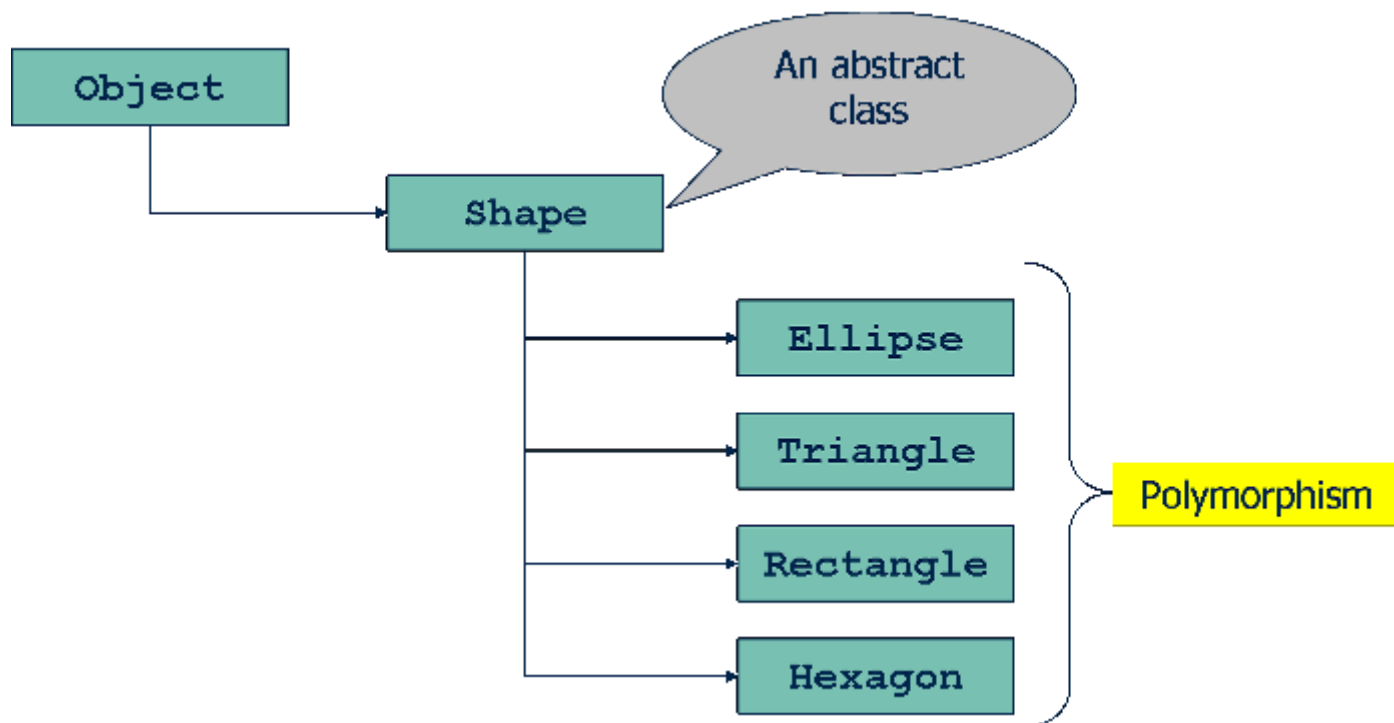
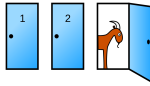


Polymorphie

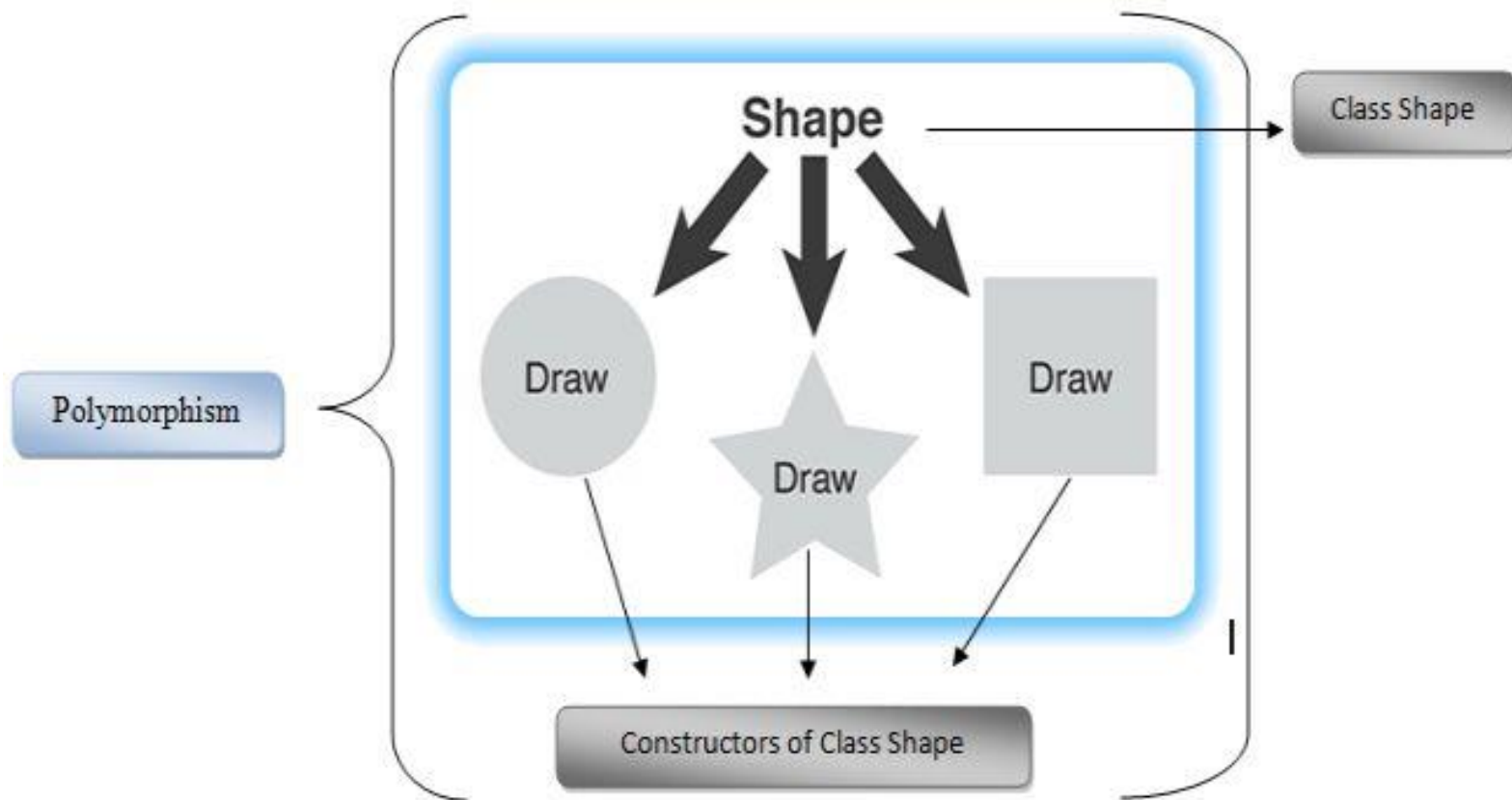
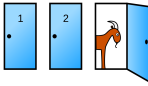
Dynamische Bindung

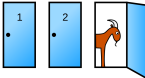
Typ des Objekts zur Laufzeit

Shapes als abstrakte Klasse

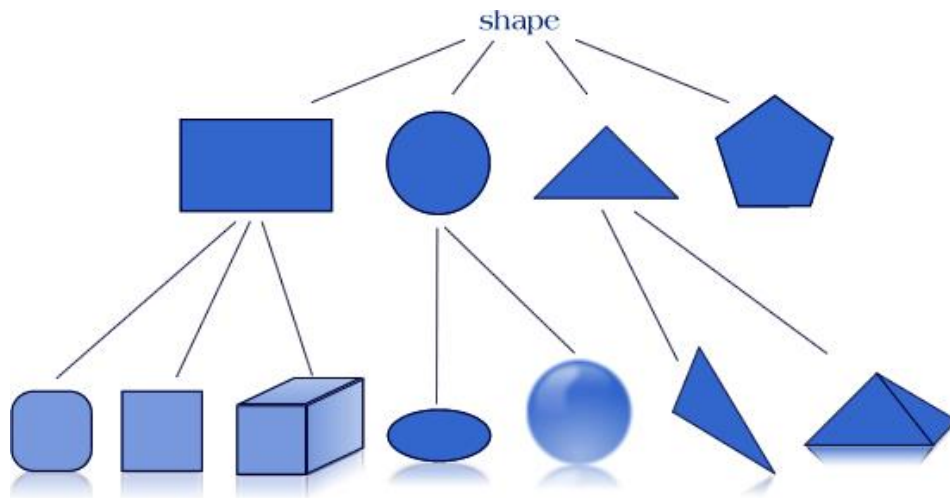


Shapes - Anzeigen...














Array of Shapes

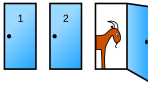


- Erstelle ein Array of Shapes, befülle diese mit unterschiedlichen konkreten Formen.
- Rufe die Methode Anzeigen() beim Durchlaufen des Arrays auf.
- Erkläre deine Beobachtungen...

- Erstelle eine Klasse Shape
- mit der Methode Anzeigen() mit einer Konsolenausgabe.
- Erzeuge abgeleitete Klassen: Rechteck, Kreis, Dreieck, Stern, ...

Schlüsselwörter

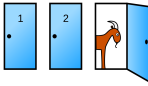


- **protected** - um auf Eigenschaften/Methoden der Basisklasse in abgeleiteten Klassen zugreifen zu können
- **sealed** - Klassen die nicht weiter abgeleitet werden
- **base** - um auf die Basisklasse zugreifen zu können
- **new** - zum Verdecken

für die Polymorphie:

- **virtual** - zum Überschreiben in abgeleiteten Klassen deklariert
- **override** - zum Überschreiben der deklarierten Methode

base Schlüsselwort



- base leitet an die Basisklasse, oder an eine der Basisklassen einer Instanz weiter.
- base ist eine implizite Referenz als solche an eine konkrete Instanz gebunden



Erben von Klasse Object in C#

- Jeder Baum hat einen **Stamm**.
- Genauso sind auch alle Klassen von .NET auf **eine allen gemeinsame Klasse** zurückzuführen: **Object**.
- Diese Klasse ist damit auch die einzige, die **selbst keine Basisklasse hat**.
 - Geben Sie bei einer Klassendefinition ausdrücklich keine Basisklasse an, *beerbt die zu implementierende Klasse immer Object*.
 - Deshalb finden Sie in der Intellisense-Liste von Referenzen immer die Methoden *Equals*, *GetType*, *ToString* und *GetHashCode*.



Referenztypen

Wertetyp vs Referenztyp

Beispiel:

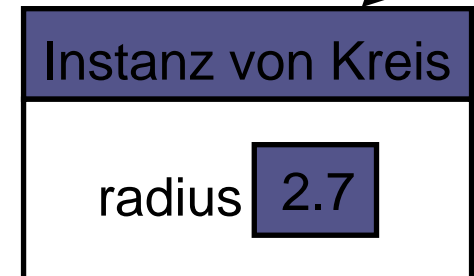
Eine Schule hat ein Array von Schüler

Referenzvariablen

- Variablen für **einfache Datentypen** (z.B. int, double)
 - Stehen als Platzhalter für Werte
 - Variablen **beinhaltet** ihren Wert
- **Referenzvariablen zeigen** auf Objekte; sie **enthalten nicht selber das Objekt** (wie Array-Variablen)
 - `Kreis kreis1 = new Kreis();`
`kreis1.radius = 2.7;`

radius 2.7

kreis1 ●



Referenzvariablen

- Variablen für **einfache Datentypen** (z.B. int, double)
 - Stehen als Platzhalter für Werte
 - Variablen **beinhaltet** ihren Wert

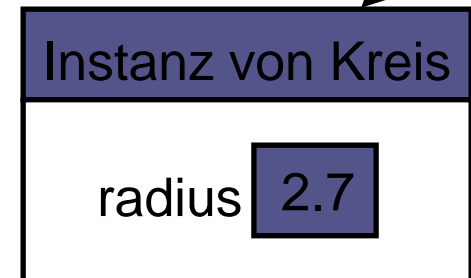
radius 2.7

- **Referenzvariablen zeigen** auf Objekte;
sie **enthalten nicht selber das Objekt**
 - Referenzvariablen, die auf kein konkretes Objekt verweisen, sollte der Wert *null* zugewiesen werden (z.B. direkt bei der Deklaration)

Beispiel:

```
Kreis kreis2 = null;
```

kreis1 ●



kreis2 ●
null

The diagram shows a variable 'kreis2' with a pointer icon (a square box containing a dot) next to it. A curved arrow points from this icon to the word 'null'.

Referenzvariablen - Fazit:

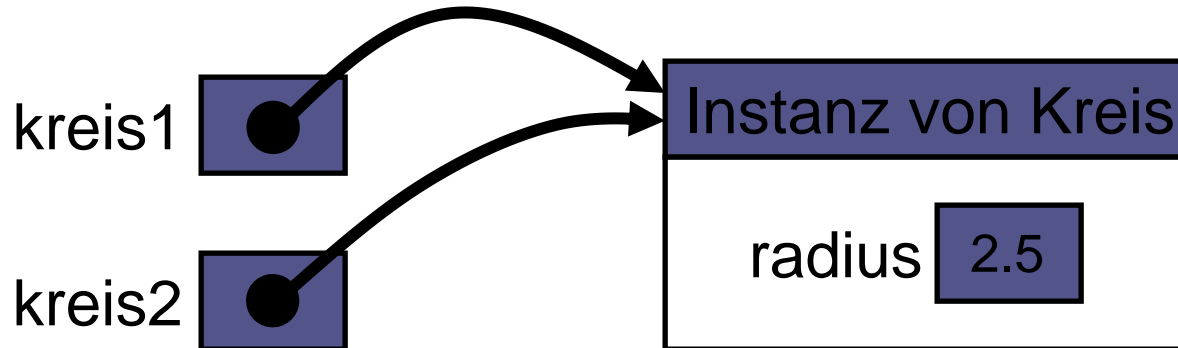
Beispiel:

```

□ Kreis kreis1, kreis2;
  kreis1 = new Kreis();
  kreis1.radius = 1.0;
  kreis2 = kreis1;
  kreis2.radius = 2.5;
    
```

Fazit:

- Zum Kopieren von Objekten oder Erzeugen neuer Objekte reicht es nicht, die Referenzen zu kopieren
- Zum Vergleich zweier unterschiedlicher Objekte reicht es nicht auf Gleichheit der Referenzen zu prüfen





Object_INITIALIZER

Object Initializer Syntax

- New way to initialize an object of a class
- Assign values to properties
 - at the time of creating an object
 - without invoking a constructor

```
Student std = new Student() { StudentID = 1,  
                               StudentName = "Bill",  
                               Age = 20,  
                               Address = "New York"  
};  
  
public class Student  
{  
    public int StudentID { get; set; }  
    public string StudentName { get; set; }  
    public int Age { get; set; }  
    public string Address { get; set; }  
}
```



Indexer

```
public object this[int index]
{
    get { /* return the specified index here */ }
    set { /* set the specified index to value here */ }
}
```

- Spezielle Art von Properties
- Erlaubt einer Klasse benutzt zu werden wie ein Array
 - Sammlung von mehreren Elementen eines Datentyps
- Wie Properties, nur:
 - Nutzt das this Schlüsselwort
 - Eckige Klammern und Parameter
 -

Syntax:

```
public <return type> this[<parameter type> index] {
    get{    // return the value from the specified index }
    set{    // set values at the specified index }
}
```

Nutzung des Indexers

```
public class StringDataStore
{
    // internal data storage
    private string[] strArr = new string[10];

    public StringDataStore() { }

    public string this[int index]{
        get
        {
            if (index < 0 && index >= strArr.Length)
                Console.WriteLine("Cannot store more than 10 objects");
            return strArr[index];
        }
        set
        {
            if (index < 0 && index >= strArr.Length)
                Console.WriteLine("Cannot store more than 10 objects");
            strArr[index] = value;
        }
    }
}
```

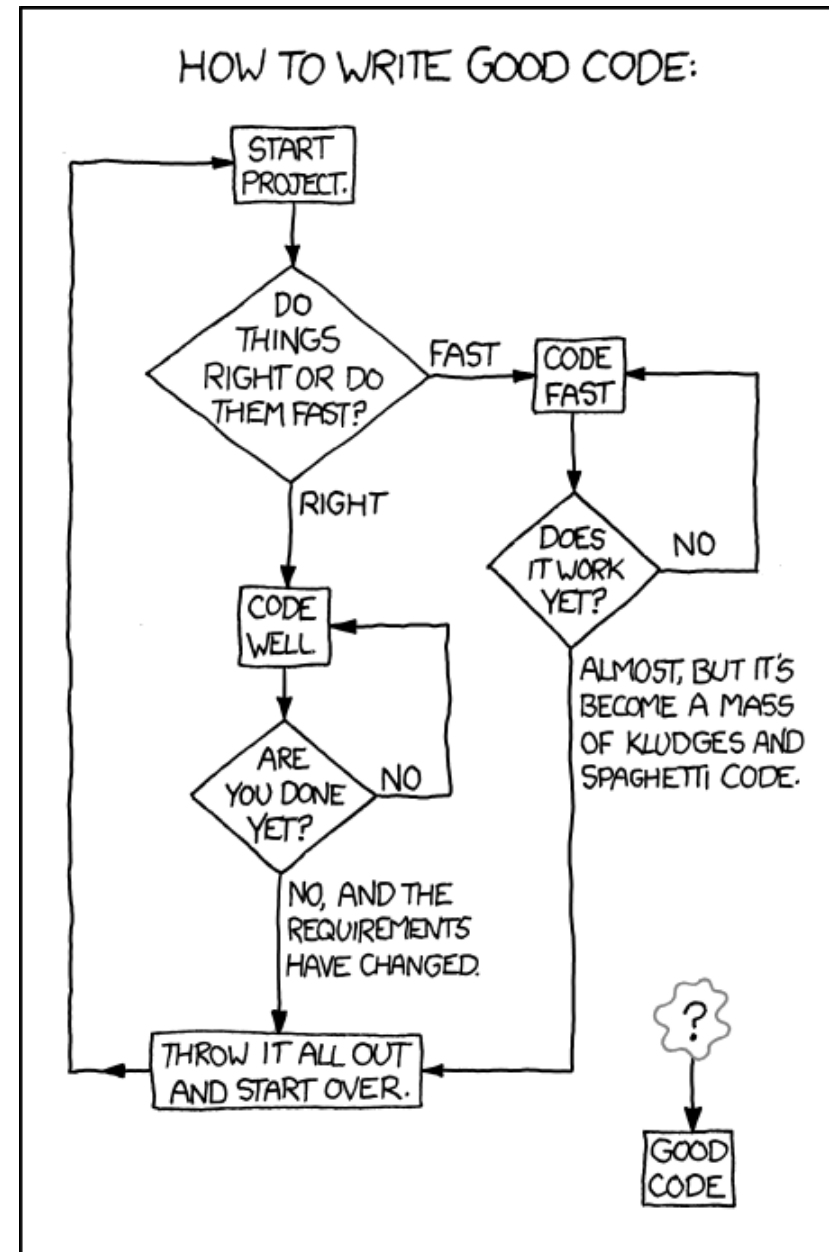
```
public static void Main()
{
    StringDataStore strStore = new StringDataStore();

    strStore[0] = "One";
    strStore[1] = "Two";
    strStore[2] = "Three";
    strStore[3] = "Four";

    for(int i = 0; i < 10 ; i++)
        Console.WriteLine(strStore[i]);
}
```

SOLID Prinzipien

Programmieren ...
... in guter Qualität



SOLID Prinzipien

- *Prinzip einer einzigen Verantwortung*
- *Trennung der Anliegen*
- *Wiederholungen vermeiden*

- *Offen für Erweiterung, geschlossen für Änderung*
- *Trennung der Schnittstelle von der Implementierung*

- *Umkehr der Abhängigkeiten*
- *Umkehrung des Kontrollflusses*

- *Mach es Testbar*

Solid-Prinzipien

