



Objektorientierte Programmierung (OOP)

Grundlagen

Praxiseinstieg Zähler



Klassen und Objekte

Klasse

Abstrakte Beschreibung
von Eigenschaften
(Attributen) und
Aktionen (Methoden).

= Schablone

Eigenschaften

Name:
Haarfarbe:
Alter:

Aktionen

laufen
sitzen
schlafen
essen



Objekt

Konkrete Instanz einer
Klasse.

Eigenschaften

Name: Sascha Müller
Haarfarbe: blond
Alter: 18

Aktionen

laufen
sitzen
schlafen
essen



Klassen und Objekte in C#

- Definition einer Klasse

```
class <Klassenname> { Attribute und Methoden }
```

- Erzeugung von Objekten

```
<Klassenname> <Objektname> = new <Klassenname> ();
```

- Aufruf einer Methode eines Objektes

```
<Objektname> .<Methodenname> (<Parameter_1> , ..., <Parameter_n> );
```

Beispiel

Klasse definieren, Objekte erzeugen

```
class Mensch {  
    public string Name;  
    public string Haarfarbe;  
    public int Alter;  
  
    public void Schlafen() {CW(name+" schläft");}  
    public void Sitzen() {CW(name+" sitzt");}  
}
```

**Definition
neue Klasse**

Attribute

Methoden

```
class TesteMensch {  
    /** Startmethode der Klasse */  
    public static void Main(string[] args) {  
        Mensch patrick = new Mensch();  
        patrick.Name="Patrick";  
        Mensch tim = new Mensch();  
        tim.Name="Tim";  
        patrick.Schlafen();  
        tim.Sitzen();  
    }
```

Testklasse

**Objekte
erzeugen
und
Methoden
aufrufen**

Tim



Patrick



Sitzen

wecken()

```
if (zustand.equals("schlafen"))  
    zustand = "wach"
```

```
}
```

tim.weckt(patrick);

```
public void weckt(Mensch schlafenderMensch) {  
    schlafenderMensch.wecken();  
}
```

Beispiel für eine Klasse und das Erzeugen eines Objektes

```
class Rechteck {  
    public float Laenge;  
    public float Breite;  
  
    public float BerechneF() {  
        float a = Laenge * Breite;  
        return a;  
    }  
  
    public void Print() {  
        CW("Reckteck (" + Laenge  
            + Breite  
            + ") ");  
    }  
}
```

```
class Test {  
    public static void Main(string[] arg) {  
  
        Rechteck r1 = new Rechteck();  
        r1.Laenge=10f;  
        r1.Breite=1f;  
  
        Rechteck r2 = new Rechteck();  
        r2.Laenge=5f;  
        r2.Breite=20f;  
  
        float f1 = r1.BerechneF();  
        float f2 = r2.BerechneF();  
  
        CW("Flächeninhalt r1:" + f1);  
        CW("Flächeninhalt r2:" + f2);  
    }  
}
```

Aufgabe Klasse CD



- Eigenschaften:
 - Interpret, Album, Erscheinungsjahr, ...
 - Titel der Lieder; 5 Lieder , ein Attribut pro Lied
 - Profiaufgabe: beliebig viele Lieder (Array)
- Aktionen:
 - `public void Print()`: Alle wichtige Daten
 - `public string LiedTitel(int nr)`: Gib den Namen des n-ten Liedes zurück
- Objekte:
 - 3 vollständig instantiierte CDs erstellen
 - Von jeder CD die Namen der Lieder 1-3 ausgeben.

Aufgabe Klasse CD (2)



Ergänzungen:

- Jedes Lied erhält eine Laufzeit in Sekunden.
- Schreiben Sie eine Methode `LaufzeitCD`, diese berechnet die Gesamt-Laufzeit der CD und gibt sie als `int`-Wert zurück.
- Verwenden Sie diese Methode für Ihre Objekte.

Aufgabe Klasse Bruch



- Definieren Sie eine Klasse Bruch zum Speichern und Rechnen mit Brüchen
- Attribute: Zähler, Nenner
- Methoden
 - Kürzen: $2/4 \rightarrow 1/2$
 - Addieren, Subtrahieren, Multiplizieren, ...
- Beispiel



In UML
und
in C#

```
Bruch b1 = new Bruch(1,2);  
Bruch b2 = new Bruch(2,4);  
Bruch b3 = b1.mult(b2).kuerzen();  
sout(b3) -> 1/4
```

Aufgabe Klasse Bruch



- **Eigenschaften:**

- Zähler
- Nenner

- **Objekte:**

- Verschiedene Brüche erstellen und mit ihnen rechnen.
- Ergebnisse ausgeben.

- **Zusatzaufgabe**

- Schreiben Sie eine statische **eval-Methode** zum Rechnen von Brüchen:

```
Bruch.Eval("1/2 + 1/4"); // 3/4
```

- **Aktionen:**

- **Ausgabe**, implementiere die `ToString()`-Methode

```
Bruch b1 = new Bruch(2,4);
```

```
Console.WriteLine(b1) // 2/4
```

- **Rechenoperationen:** + - * /

```
Bruch b1 = new Bruch(2,4);
```

```
Bruch b2 = new Bruch(1,4);
```

```
Bruch b3 = b1.Add(b2); // b3 = 3/4
```

- **Dezimalwert**

```
b1.Dezimalwert() // 0.5
```

- **Kürzen**

```
b1.Kürzen() // b1=1/2
```

Aufgabe



Modellieren Sie folgende Szenarien mit Klassen und Objekten

- Auto

- Eigenschaften: Besitzer, Marke, Modell, Baujahr, Leistung, ...
- Aktionen: Ausgabe, fahren, tanken, beschleunigen, bremsen, ...

- Zoo

- Klassen: Papagei, Pinguin, Löwe, ...
- Eigenschaften: Name, sprechen, schwimmen, brüllen, ...

- Computer-Adventure

- Klassen: Held, Monster
- Eigenschaften: lebenspunkte, angriffswert
- Aktionen: kämpfen, heilen, ...

- Eigenes Szenario



Sichtbarkeit (Zugriffsmodifizierer)

Sichtbarkeiten

Schlüsselwörter

Sichtbarkeiten werden durch Schlüsselwörter angegeben:

C#

- **public**
 - Zugriff für alle Klassen
- **protected**
 - für die eigene Klasse und Subklassen und im package
- **internal (keins)**
 - für Klassen im assembly
- **private**
 - Zugriff nur für die eigene Klasse

UML

- **+**
- **#**
- **-**


Sichtbarkeiten

Beispiel

```
1 ▼ public class Papagei {  
2     public String name;  
3 }
```

```
1 ▼ public class Pinguin {  
2     private String name;  
3 }
```

```
1 ▼ public class Zoo {  
2 ▼     public static void main(String[] args) {  
3         Papagei alex = new Papagei();  
4         Pinguin tux = new Pinguin();  
5  
6         alex.name = "Alex";  
7         tux.name = "Tux";  
8     }  
9 }
```





Konstrukturen

Konstrukturen

```
class Rechteck {  
    public float laenge;  
    public float breite;  
  
    public Rechteck(float l1, float b1) {  
        laenge=l1;  
        breite=b1;  
    }  
  
    public Rechteck(float laenge1) {  
        laenge=laenge1;  
    }  
  
    public Rechteck() {}  
  
    public float berechneF() {  
        float a = laenge * breite;  
        return a;  
    }  
}
```

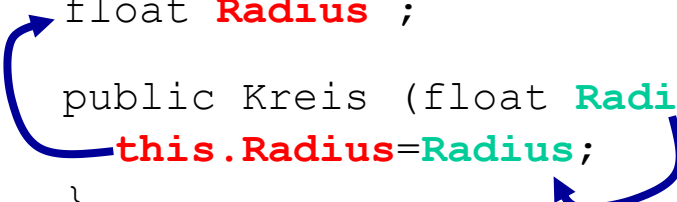
Konstruktor

```
class Test {  
    public static void Main(string[] arg) {  
  
        Rechteck r1 = new Rechteck(10,1);  
  
        Rechteck r2 = new Rechteck(5);  
        r2.Breite=20;  
  
        float f1 = r1.BerechneF();  
        float f2 = r2.BerechneF();  
  
        CW("Flächeninhalt r1:"+f1);  
        CW("Flächeninhalt r2:"+f2);  
  
    }  
}
```

Der Konstruktor ohne Parameter (Default-Konstruktor) ist immer definiert, falls sonst kein Konstruktor angegeben ist.

Der this-Operator

```
class Kreis{  
    static readonly float pi = 3.14127f;  
    float Radius ;  
    public Kreis (float Radius) {  
        this.Radius=Radius;  
    }  
}
```

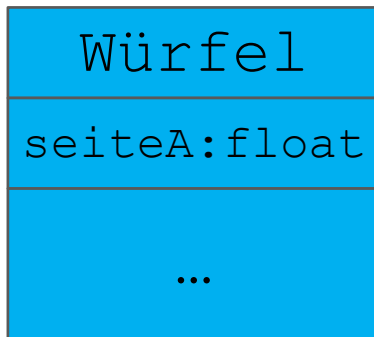


Konflikt:
lokale Variable und **Attribut**.
this. verweist auf das Attribut.

Anlegen eines Objektes im Speicher

①

```
class Würfel {  
    float seite A;  
    ...  
}
```



Definition der Klasse

②

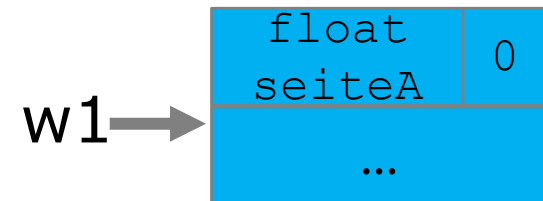
```
Würfel w1;
```



W1 ist Referenz (Variable), die auf kein Objekt zeigt.

③

```
w1 = new Würfel();
```



W1 verweist auf ein neues Objekt der Klasse Würfel.



Getter/Setter Properties

Warum Getter und Setter?

```
class Rechteck {  
    public float Laenge;  
    public float Breite;  
}
```

Die Attribute können von allen verändert werden, es gibt keine Kontrolle.

```
class Rechteck {  
    private float Laenge;  
    private float Breite;  
}
```

Schränke den Zugriff ein, jetzt darf nur noch die Klasse Rechteck sie ändern.

```
class Rechteck {  
    private float Laenge;  
    private float Breite;  
  
    public float GetLaenge() {  
        return Laenge;  
    }  
  
    public void SetLaenge(float laenge) {  
        Laenge=laenge;  
    }  
    ...  
}
```

Erlaube den Zugriff über Methoden.

```
class Test {  
    ...  
    Rechteck r2 = new Rechteck(5);  
    //r2.Breite=20;  
    //funktioniert nicht mehr  
    r2.SetBreite(20);  
    ...  
}
```

Getter/Setter-Methoden bieten Kontrolle

```
class Rechteck {  
    private float Laenge;  
    private float Breite;  
  
    ...  
    public void setLaenge(float laenge1) {  
        if (laenge1 >= 0)  
            Laenge = laenge1;  
        else  
            CW("Warnung: Länge darf nicht negativ sein!");  
    }  
    ...  
}
```

Getter/Setter als Properties Kurzform

```
class Rechteck {  
    public float Laenge {get; set;}  
    public float Breite {get; set;}  
}
```

Kein Semicolon nach {...}

Jeder kann wieder zugreifen

**get oder set kann auch
weggelassen werden**

```
class Test {  
    ...  
    Rechteck r2 = new Rechteck(5);  
    r2.Breite=20f;  
    //funktioniert wieder, ist  
    //allerdings ein Methodenaufruf  
    ...  
}
```

Getter/Setter als Properties Langform

```
class Rechteck {  
    private string farbe;  
    public string Farbe {  
        get {  
            if (farbe != "")  
                return farbe;  
            else  
                return "Keine Farbe";  
        }  
        set {  
            farbe = value;  
        }  
    }  
}
```

Privates Attribut **farbe**
Öffentliche Property **Farbe**

**value bezeichnet den
zugewiesenen Wert (hier
"blau")**

```
} class Test {  
    ...  
    Rechteck r2 = new Rechteck(5f);  
    r2.Farbe="blau";  
    ...  
}
```

**Spezialität von C#:
Properties bieten den
komfortablen Zugriff wie auf
Attribute (r2.Farbe=...),
jedoch mit voller Kontrolle wie
bei Getter/Setter-Methoden.**



Statische Attribute

Statische Attribute

```
class Kreis{  
  
    static readonly float PI = 3.1415927f;  
    float Radius;  
  
    public Kreis (float radius1) {  
        Radius=radius1;  
    }  
  
    public Kreis() {}  
  
    public float berechneFlaecheninhalt() {  
        float x = PI * radius * radius;  
        return x;  
    }  
}
```

Statisches Attribut:
Wird nur einmal für die Klasse
(nicht für jedes Objekt)
definiert.

Zugriff auf ein
statisches Attribut
bleibt innerhalb der Klasse
gleich.

Zugriff auf eine
statisches Attribut
ist ohne Objekt möglich:
Klassenname.Attribut

```
class Test {  
    public static void Main(string[] arg) {  
  
        Kreis k1 = new Kreis(7);  
        float f1=k1.BerechneFlaecheninhalt();  
        CW("Flächeninhalt k1:"+f1);  
        CW("Wert von PI:"+Kreis.PI);  
  
    }  
}
```



Die ToString()-Methode

Objekte „printable“ machen: **ToString()**

```
class Kreis {  
    ...  
  
    public override string ToString()  
        return $"Kreis({radius})";  
    }  
}
```

```
class Test {  
    public static void Main(string[] args) {  
  
        Kreis k1 = new Kreis();  
        k1.SetRadius(7);  
        float f1 = k1.BerechneF();  
        CW($"Flächeninhalt von {k1} ist {f1}");  
    }  
}
```

Ohne ToString()

Flächeninhalt von **OOP.Kreis** ist 153,92223

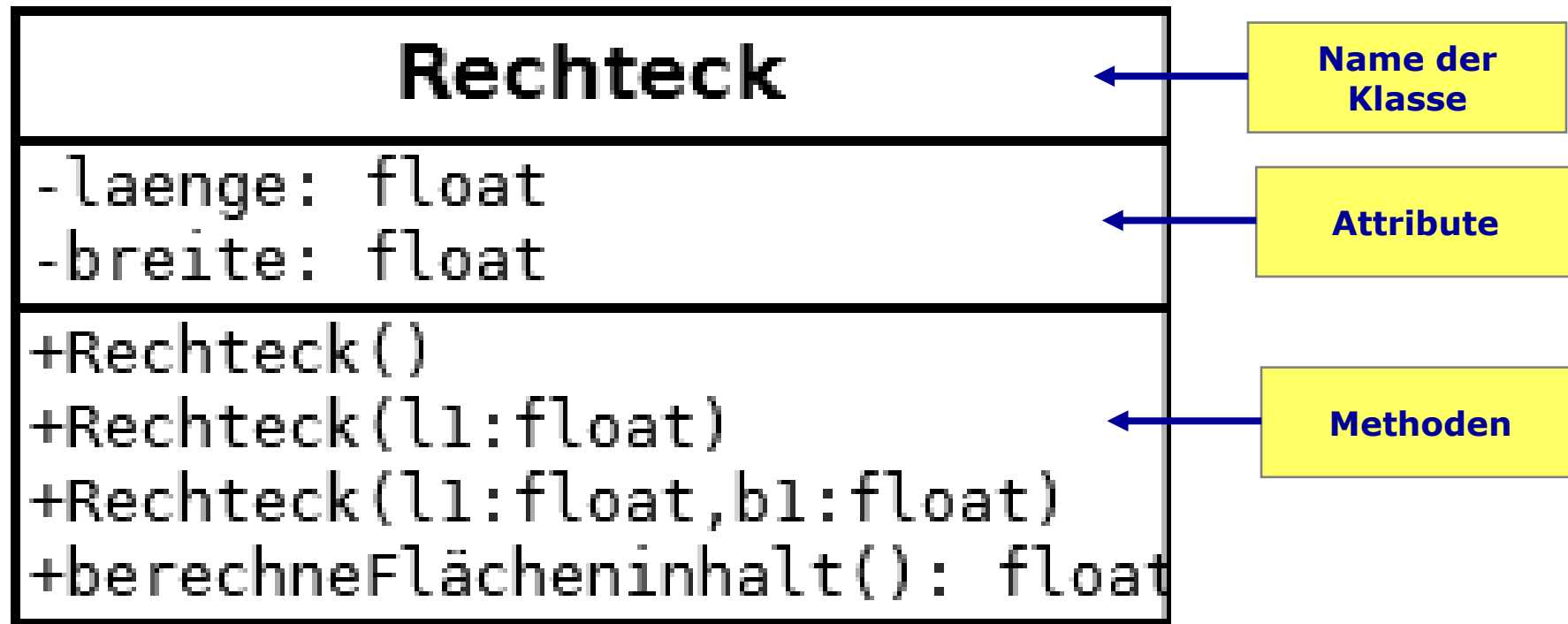
Mit ToString()

Flächeninhalt von **Kreis(7)** ist 153,92223

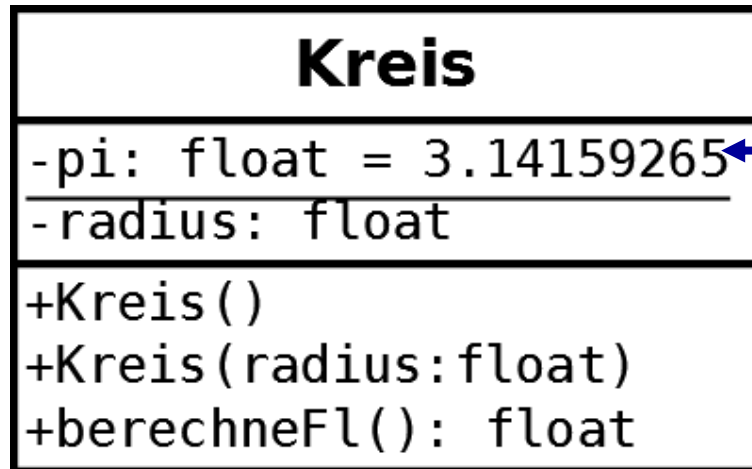


UML

UML-Diagramm

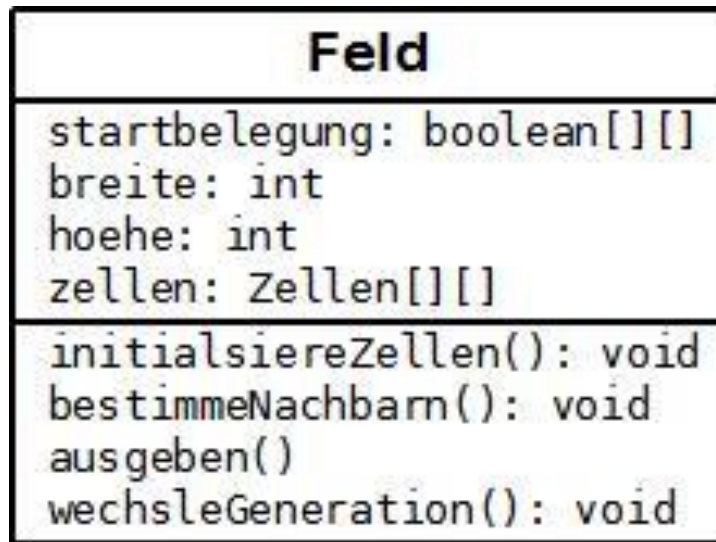


Statische Attribute in UML




Statische Attribute
werden in UML unterstrichen.

Vom Diagramm zum Code



```
1 public class Feld {  
2     boolean startbelegung[][];  
3     int hoehe;  
4     int breite;  
5  
6     Zelle zellen[][];  
7  
8  
9  
10    void initialisiereZellen() { }  
11  
12    void bestimmeNachbarn() { }  
13  
14    void ausgeben() { }  
15  
16    void wechsleGeneration() { }  
17 }
```



Editor: Dia

UML-Symbole

Eigenschaften,
Aktionen
eingeben

The screenshot displays the Dia UML editor interface. The main window shows a class diagram for a class named "Feld". The class has four attributes: "startbelegung: boolean[][]", "breite: int", "hoehe: int", and "zellen: Zellen[][]". It also has four methods: "initialisiereZellen(): void", "bestimmeNachbarn(): void", "ausgeben()", and "wechsleGeneration(): void". The "UML" category is selected in the left toolbar. The "Eigenschaften: UML - Class" panel on the right shows the class's properties, with the "Attribute" tab selected. The attribute "breite: int" is highlighted, and its details are shown in the "Attributdaten" section: Name: "breite", Typ: "int", Wert: "", and Kommentar: "". The "Sichtbarkeit" (Visibility) is set to "Implementierung" (Implementation).

Diagramm1.dia (C:\Users\bal\Pictures) - diaw.exe

Datei Bearbeiten Ansicht Ebenen Objekte Auswahl Werkzeuge Eingabemethoden Hilfe

Diagramm1.dia x

UML

Feld

- startbelegung: boolean[][]
- breite: int
- hoehe: int
- zellen: Zellen[][]

- initialisiereZellen(): void
- bestimmeNachbarn(): void
- ausgeben()
- wechsleGeneration(): void

Eigenschaften: UML - Class

Klasse Attribute Operationen Vorlagen Stil:

startbelegung: boolean[][]

breite: int

hoehe: int

zellen: Zellen[][]

Attributdaten

Name: breite

Typ: int

Wert:

Kommentar:

Sichtbarkeit: Implementierung

UML erstellen



Erstellen Sie ein UML-Diagramm für die Klassen

Rechteck, Quadrat, Kreis



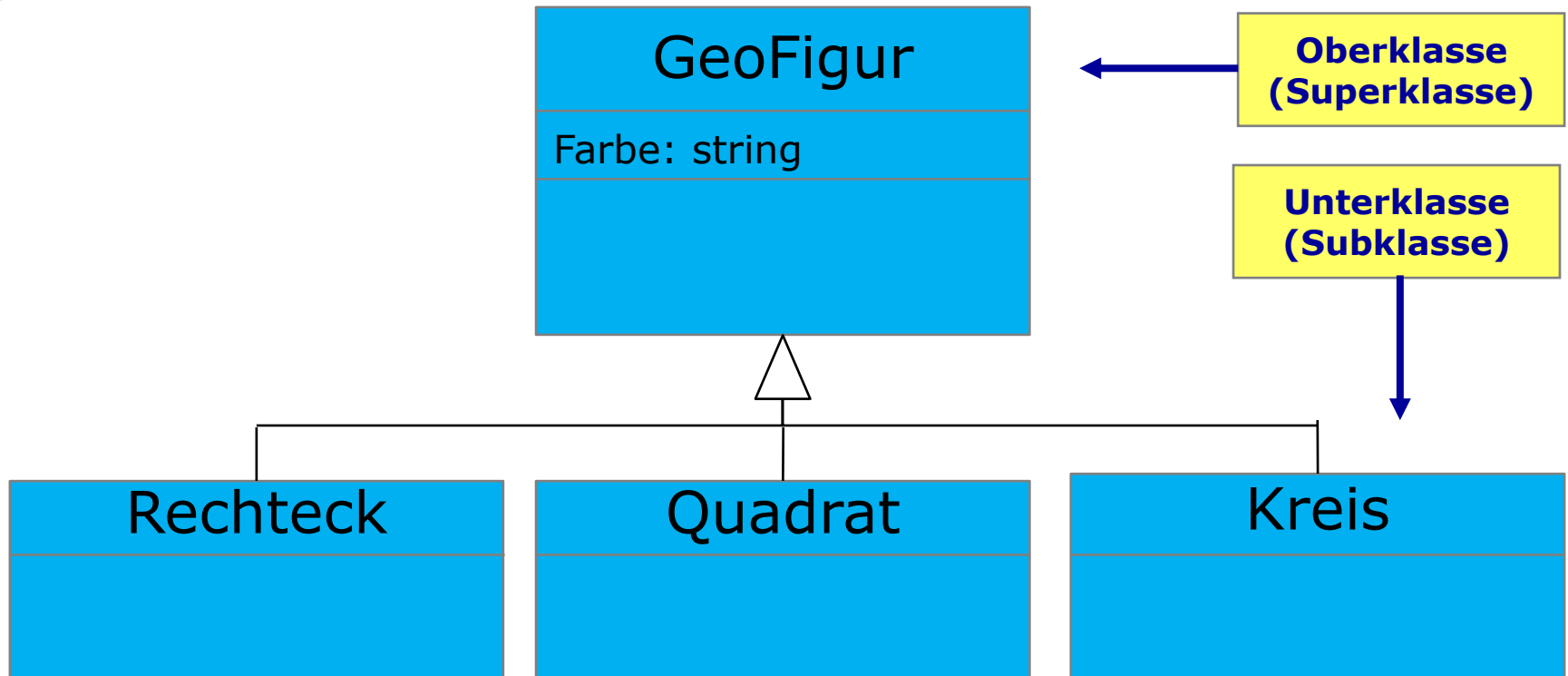
mit folgenden Eigenschaften:

- Definition aller Attribute
- Verschiedene Konstruktoren
- Getter-/Setter-Methoden
- Methode berechneFlaecheninhalt
- Methode berechneUmfang



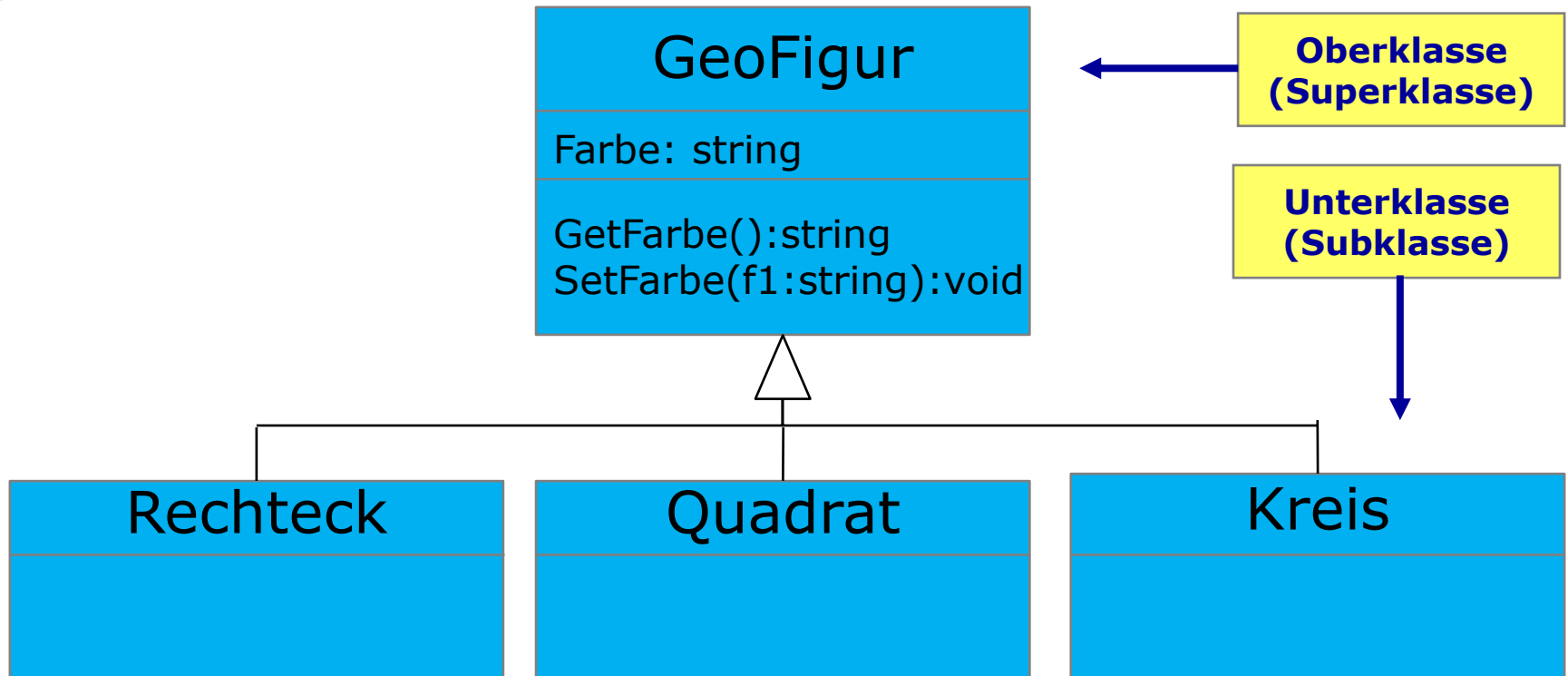
Vererbung

Vererbung: Attribute werden vererbt



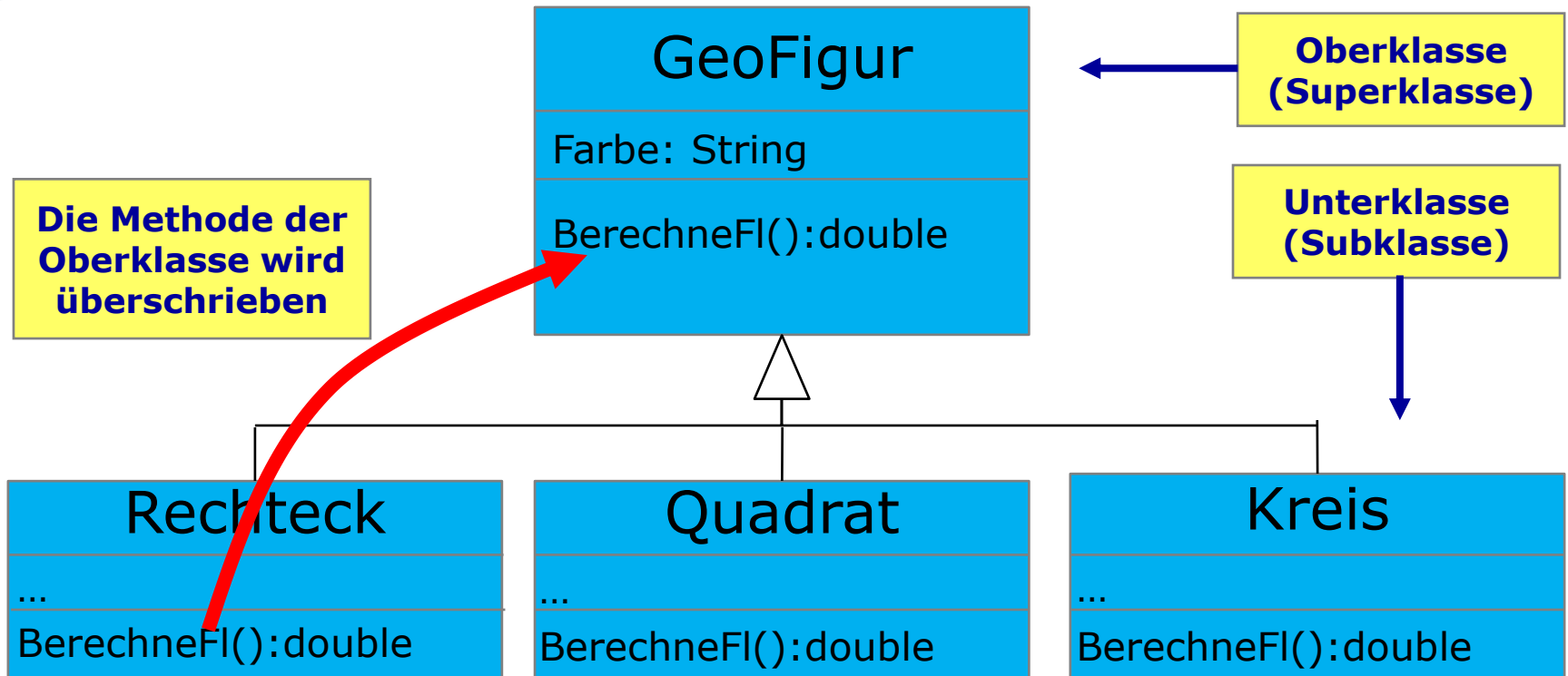
```
public class GeoFigur{...}
public Rechteck : GeoFigur {...}
Rechteck r1 = new Rechteck();
r1.Farbe="rot";
```

Vererbung: Methoden werden vererbt



```
public class GeoFigur{...}
public Rechteck : GeoFigur {...}
Rechteck r1 = new Rechteck();
r1.SetFarbe ("rot");
```

Vererbung: Methoden der Oberklasse können überschrieben werden



```
public class GeoFigur{...}
public Rechteck : GeoFigur {...}
Rechteck r1 = new Rechteck();
double flaeche = r1.BerechneFl();
```

In C#: Schlüsselworte **virtual** und **override** notwendig

```
class GeoFigur{
    ...
    public virtual float BerechneFl() {
        CW("Warnung: Methode noch in
            Unterklasse definieren!");
        return -1;
    }
}
```

```
class Rechteck{
    ...
    public override float BerechneFl() {
        return Laenge*Breite;
    }
}
```



Polymorphie

Polymorphie: verschieden Sichtweisen auf Objekte

```
class Program{  
    public static void Main(string[] a) {  
        Rechteck f1 = new Rechteck(10,20);  
        Kreis f2 = new Kreis(15);  
        Quadrat f3 = new Quadrat(100);  
        GeoFigur[] figuren = {f1,f2,f3};  
        float erg=0;  
        for(int i=0;i<figuren.length;i++){  
            erg = erg + figuren[i].berechneFl();  
        }  
        CW(erg);  
    }  
}
```

Ein Objekt ist
sogleich vom Typ der
Unterklasse als auch
der **Oberklasse**

Die Methode der
Unterklasse wird
aufgerufen, da die
der **Oberklasse**
überschrieben wird

Typ des Arrays:
GeoFigur
(Oberklasse)


Typ der Elemente:
Rechteck/Kreis/Quadrat
(Unterklasse)

Ein Rechteck ist sowohl
ein **Rechteck** als auch
eine **geometrische Figur**

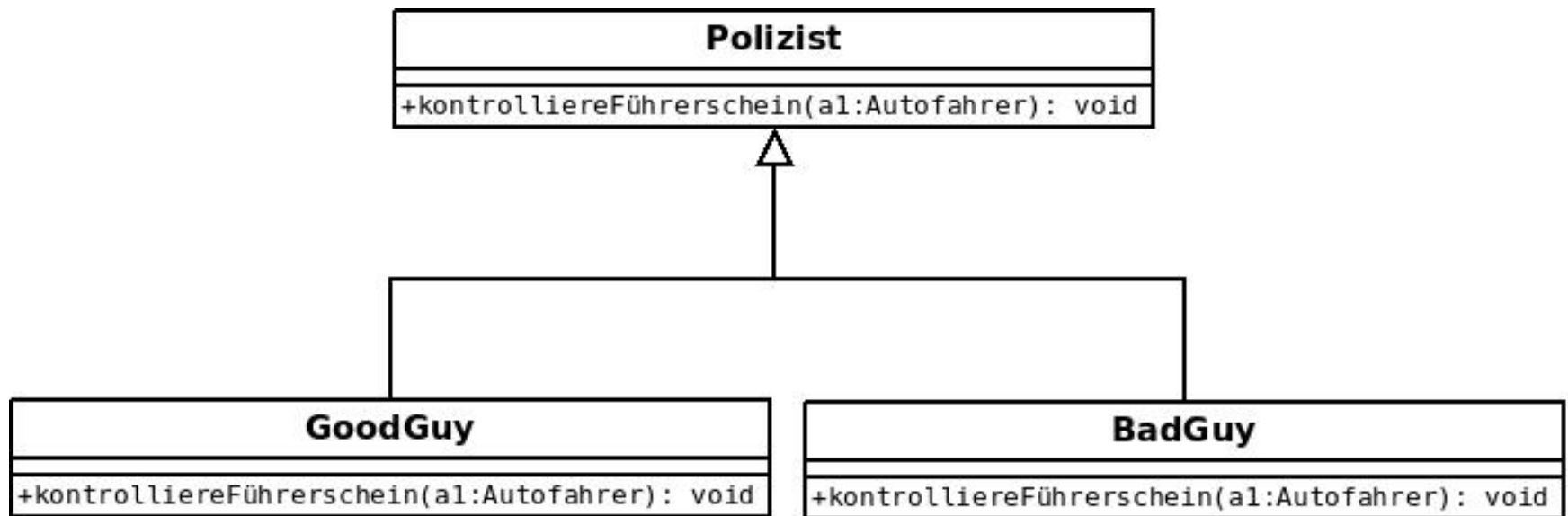
Polymorphie: verschieden Sichtweisen auf Objekte

```
class Program{  
    public static void Main(string[] a) {  
        GeoFigur f1 = new Rechteck(10,20);  
        GeoFigur f2 = new Kreis(15);  
        GeoFigur f3 = new Quadrat(100);  
        GeoFigur[] figuren = {f1,f2,f3};  
        float erg=0;  
        for(int i=0;i<figuren.length;i++){  
            erg = erg + figuren[i].berechneFl();  
        }  
        CW(erg);  
    }  
}
```

Objekte der Unterklasse können als Objekte der Oberklasse definiert werden.



Polymorphie: Beispiel Polizist



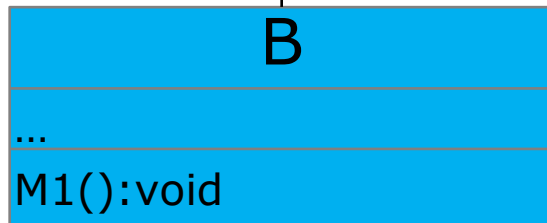
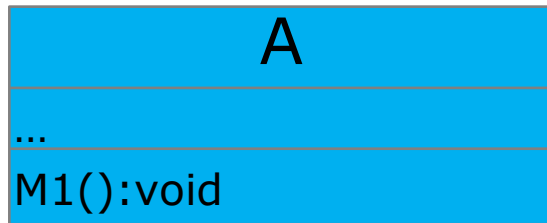
Objekte erzeugen:

```
Polizist p1 = new Polizist();  
Polizist p2 = new GoodGuy();  
Polizist p3 = new BadGuy();
```



Überschreiben und Überladen von Methoden

Überschreiben von Methoden



```
class A {  
    public virtual void M1() {  
        CW("A");  
    }  
}
```

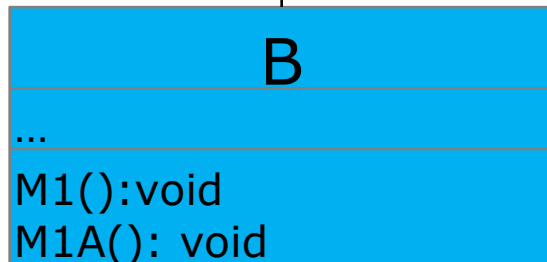
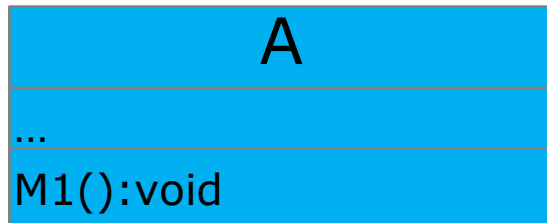
```
class B : A {  
    public override void M1() {  
        CW("B");  
    }  
}
```

```
B b = new B();  
b.M1();
```

Ausgabe "B" // Methode M1() wird in B überschrieben

Überschreiben von Methoden

Aufruf der überschriebenen Methode mit **base**



```
class A {  
    public void M1() {  
        CW("A");  
    }  
}
```

```
class B : A {  
    public void M1() {  
        CW("B");  
    }  
    public void M1A() {  
        base.M1();  
    }  
}
```

```
B b = new B();
```

```
b.M1();
```

```
b.M1A();
```

Ausgabe "B" // Methode m1() wird überschrieben

Ausgabe "A" // m1() aus A wird aufgerufen

Die Operatoren `base()` und `this()` bei Konstruktoren

```
class GeoFigur{  
    private string farbe;  
  
    public GeoFigur(string farbe){  
        this.farbe=farbe;  
    }  
  
    public GeoFigur() {}  
  
}
```

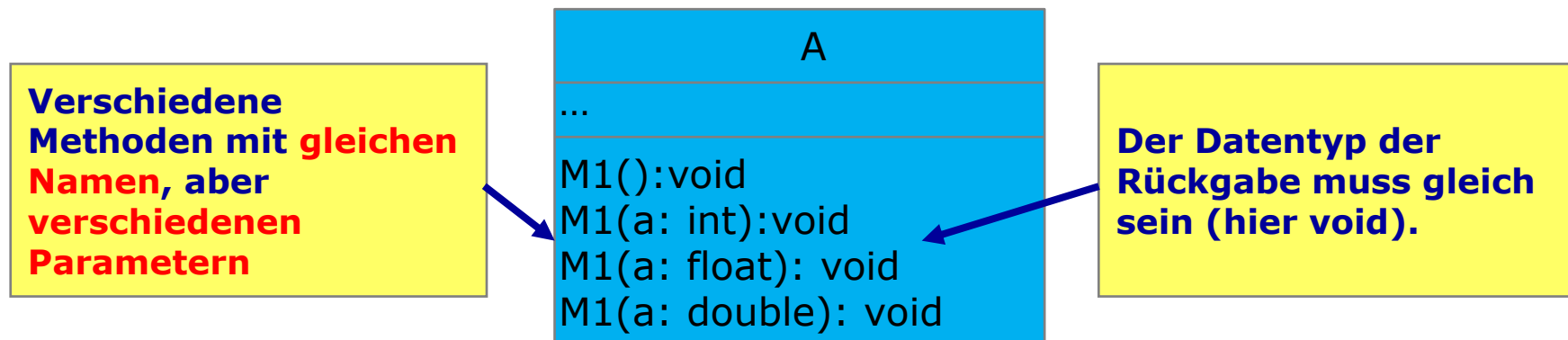
```
class Kreis : GeoFigur{  
    private float radius;  
  
    public Kreis() : base("Vom Kreis  
erzeugt!") {}  
    public Kreis(float radius):this() {  
        this.radius=radius;  
    }  
  
}
```

**Explizierter Aufruf des
Konstruktors der **Oberklasse**
(Ansonsten wird `GeoFigur()`
implizit aufgerufen)**

**Aufruf eines anderen
Konstruktors der **eigenen**
Klasse (Dieser Befehl muss an
erster Stelle stehen)**

Überladen von Methoden

Gleicher Methodenname, unterschiedliche Parameter



```
A a = new A();  
a.M1(3);  
a.M1(3.5f);  
a.M1(3.5);
```


Aufruf eines Konstruktors aus einem Konstruktor

A

Value:int

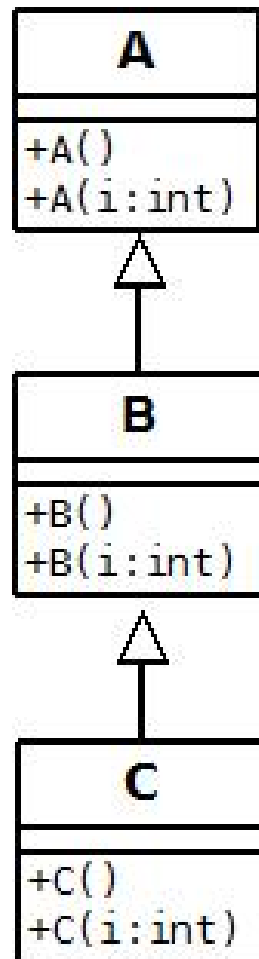
A()

A(i:int)

```
public class A {  
    public A() {  
        //Do initialisation stuff  
    }  
  
    public A(int v) : this() {  
        //Do initialisation stuff  
        //and set Value  
        value=v;  
    }  
}
```

**Ruft zuerst
Konstruktor A() auf**

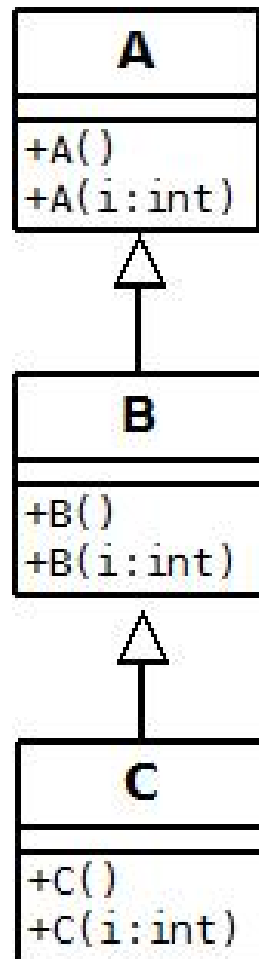
In welcher Reihenfolge werden die Konstruktoren aufgerufen?



```
C c1 = new C();
```

```
C c2 = new C(2);
```

Aufrufreihenfolge von Konstruktoren



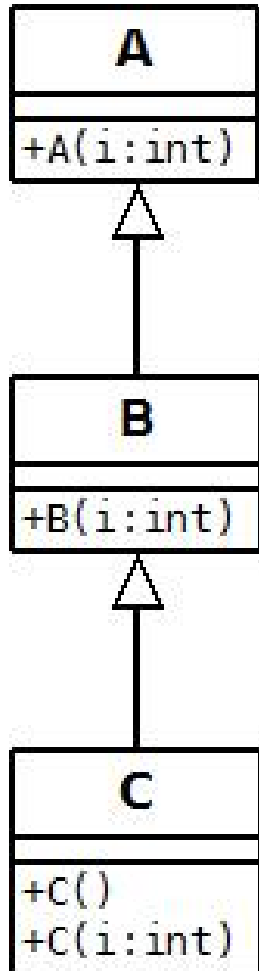
```
C c1 = new C();
```

Konstruktor A
Konstruktor B
Konstruktor C

```
C c2 = new C(2);
```

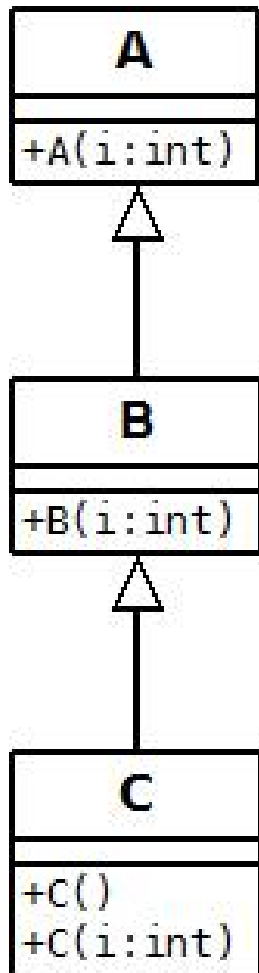
Konstruktor A
Konstruktor B
Konstruktor Cint

Was machen, wenn der Default-Konstruktor fehlt?!? (Und man ihn nicht hinzufügen darf...)



```
C c1 = new C(); //error
```

Was machen, wenn der Default-Konstruktor fehlt?!?



```
C c1 = new C();

public class B : A {
    public B(int i) : base(i) {
        CW("Kons B mit int");
    }
}

public class C : B {
    public C() : base(0) {
        CW("Kons C");
    }
    public C(int i) : base(i) {
        CW("Kons C mit int");
    }
}
```



Übergabe von Objekten

Übergabe von Objekten

Beispiel: **Polizist** kontrolliert **Autofahrer**

```
class Mensch {string name;}
```

```
class Autofahrer : Mensch {...}
```

```
class Polizist : Mensch{
```

```
    public void kontrolliereFührerschein(Autofahrer a1) {
```

```
        if (a1.hatFührerschein()==true) {
```

```
            CW("Polizist "+name+" kontrolliert Führersch
```

```
            CW("Autofahrer "+a1.name+"hat einen Führerschein");
```

Der Zugriff auf
Attribute und Methoden
von **Autofahrer**
geschieht über das
übergebene Objekt **a1**

Der Zugriff auf
Attribute und Methoden
von **Polizist**
geschieht direkt
(es wird kein Objekt übergeben)



Dem **Polizist** wird ein
Objekt **a1** von **Autofahrer**
übergeben.



Interfaces



OOP Prinzipien

Objektorientiertes Programmieren

Prinzipien

Das Objektorientierte Programmieren ermöglicht durch Klassen und Objekte die Welt zu modellieren. Folgende Prinzipien werden dabei verwendet:

- **Kapselung**
 - Daten und Methoden werden dort definiert, wo sie auch gebraucht werden, nämlich in der Klasse. Zugriffsrechte regeln Sichtbarkeit nach außen.
- **Vererbung**
 - Die Weitergabe von Daten und Methoden an Unterklassen ermöglicht den Aufbau von Klassen-Hierarchien.
- **Polymorphie**
 - Ein Objekt kann auf verschiedene Weisen betrachtet werden, z.B. ist ein Schüler ein Mensch und ein Lernender. Dadurch ist es möglich, Methoden sehr allgemein zu formulieren.

Verwendung von Interfaces

- `List<String> a1 = new ArrayList<>();`
- `List<String> a2 = new LinkedList<String>();`
- Interface List
 - `Add()`
 - `Remove()`