

# Programmierung 2

## Kapitel 3 Vererbung

# Verständnisfragen



- Wenn alle Tests bestanden sind, kann man davon ausgehen, dass die Software fehlerfrei ist?
- Was ist ein Unit-Test?
- Welche andere Testarten (außer Komponententest) gibt es?
- Welche Fälle testet man in einem Unit-Test?
- Wann sollte man Unit-Tests schreiben: vor oder nach der Implementierung der Komponente (Unit)?
- Was ist TDD?
- Wie geht man beim TDD vor? Zählen Sie die Testschritte auf und erläutern sie (z.B. beim Testen einer Methode).
- Was sind die Vorteile von TDD?

# Überblick Programmierung 2



## Software-Engineering

- Testen: Unit-Tests mit JUnit
- Strukturieren: Pakete
- Kommentieren: Javadoc, Annotationen

## OO- Programmierung

- Vererbung (+ Operatoren)
- super-Operator
- Polymorphie von Objekten
- Finale Klassen
- Abstrakte Klassen und Schnittstellen
- Wrapper- Klassen
- generische Klassen

## Dynamische Datenstrukturen

- Verkettete Listen
- Stack, Queue, Binärbäume

## GUI

- GUI-Programmierung mit Swing

# ++ Operator und Nebeneffekte

# Ausdrücke

## Ausdruck in Java

- **ein Bezeichner** (Name) einer Variable oder einer Konstante (**einfache** Ausdrücke), z.B. 2, false, var, k
  - mehrere Bezeichner durch Operatoren verknüpft (**komplexe** Ausdrücke), z.B. `var+k`, `2*k<(-3)-var`, `k&10`
- hat immer einen **Rückgabewert** (= Wert des Ausdrucks)
- Rückgabewert hat immer einen **Datentyp**

# Einstellige arithmetische Operatoren

- positiver Vorzeichenoperator  $+A \rightarrow$  Rückgabewert  $A$
- negativer Vorzeichenoperator  $-A \rightarrow$  Rückgabewert  $-A$
- Postfix-Inkrementoperator  $A++ \rightarrow$  Rückgabewert  $A$
- Präfix-Inkrementoperator  $++A \rightarrow$  Rückgabewert  $A+1$
- Postfix-Dekrementoperator  $A-- \rightarrow$  Rückgabewert  $A$
- Präfix-Dekrementoperator  $--A \rightarrow$  Rückgabewert  $A-1$

# Postfix-Inkrement: Beispiel

`a++ - a`



Was ist der Rückgabewert?

# Auswertung der Operanden

In Java:

Operanden eines Operators werden **strikt von links nach rechts** ausgewertet.



Jeder Operand eines Operators wird **vor** der Operation **vollständig ausgewertet**.

*Ausnahmen:* Operatoren &&, || und ? :



# Prioritäten der Operatoren

**Bindungsstärke** (auch: **Priorität**) eines Operators  $\rightarrow$  entscheidet über *Auswertungsreihenfolge* der (Teil-)Ausdrücke

Regeln (wie in der Mathematik):

- Punkt bindet stärker als Strich:  $5+2*3 \rightarrow 5+(2*3)$
- unär bindet stärker als binär:  $-5+6 \rightarrow (-5)+6$
- Klammern überschreiben Bindung:  $(5+2)*3 \rightarrow (5+2)*3$
- bei *gleicher* Bindungsstärke (Priorität) – **Assoziativität** eines Operators entscheidend

*Beispiele:*

- $-\sim x \rightarrow -(\sim x)$ , da unäre Operatoren rechtsassoziativ
- $a + b - c \rightarrow (a + b) - c$ , da + und – linksassoziativ

# Postfix-Inkrement: Beispiel

$a++ - a$



Was ist der Rückgabewert?

Angenommen,  $a$  hat den Wert 7.

$a$   
7

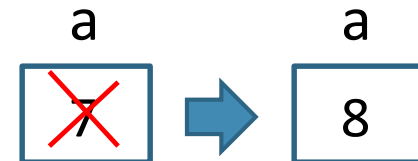
- Bindung:  $(a++) - a$
- Auswertung: zuerst Operand links:  $a++$ , dann rechts:  $a$
- $a++$  vollständig auswerten vor der Subtraktion

1.  $a++$  auswerten

→ Rückgabewert: 7

→ Variable  $a$  der Wert  $a+1$  d.h. 8 zugewiesen

→ **Nebeneffekt !**



2.  $a$  auswerten → Rückgabewert: 8

3. Subtrahieren:  $7-8$  → Rückgabewert: -1

# Prefix-Inkrement: Beispiel

$++a - a$



Was ist der Rückgabewert?

Angenommen, a hat den Wert 7.

a  
7

- Bindung:  $(++a) - a$
- Auswertung: zuerst Operand links:  $++a$ , dann rechts:  $a$
- $++a$  vollständig auswerten vor der Subtraktion

1.  $++a$  auswerten

→ Rückgabewert:  $a+1$  d.h. 8

→ Variable a der Wert  $a+1$  d.h. 8 zugewiesen

→ **Nebeneffekt !**



2.  $a$  auswerten → Rückgabewert: 8

3. Subtrahieren:  $8-8$  → Rückgabewert: 0

# Nebeneffekt

**Nebeneffekt** (auch: **Seiteneffekt**, **Nebenwirkung**):

**Veränderung einer Variable** (→ meist der *Wert* einer (statischen) Variable *im Speicher* verändert), die „nebenbei“ **während der Auswertung** eines Ausdrucks (in dem die Variable vorkommt) stattfindet



**Operatoren mit Seiteneffekten mit Bedacht einsetzen**,  
ansonsten Programm fehleranfällig und schlecht lesbar

# Aktuelle Parameter auswerten

**Auswertungsreihenfolge** der aktuellen Parameter in Java:  
immer **von links nach rechts**

→ beim Aufruf von `methode (param1 , param2 )` :

1. `param1` ausgewertet
2. ggf. Seiteneffekte von `param1` stattgefunden
3. `param2` ausgewertet
4. ggf. Seiteneffekte von `param2` stattgefunden

# Aktuelle Param. auswerten: Bsp.

```
public class Auswertung {  
    public static void methode (int f1, int f2) {  
        System.out.println ("f1: " + f1);  
        System.out.println ("f2: " + f2);  
    }  
  
    public static void main (String[] args) {  
        int aktuell = 1;  
        methode (aktuell++, aktuell);  
        System.out.println ("aktuell: " + aktuell);  
    }  
}
```



Was ist die Ausgabe des Programms?

f1: 1  
f2: 2  
aktuell: 2

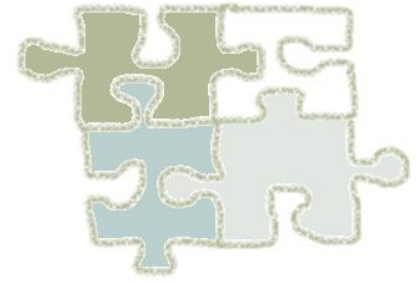
# Verständnisfragen



- Was ist der Unterschied zwischen Postfix- und Prefix-Inkrement? Was ist der Rückgabewert des Ausdrucks `a++` und `++a` für `a = 1`? Was ist der Wert von `a` nach der Anweisung `a++`; bzw. `++a`?
- Was ist ein Nebeneffekt?
- Was ist der Wert von `v` und `u` nach der letzten Anweisung?

```
int u = 1;  
int v;  
v = u++;
```

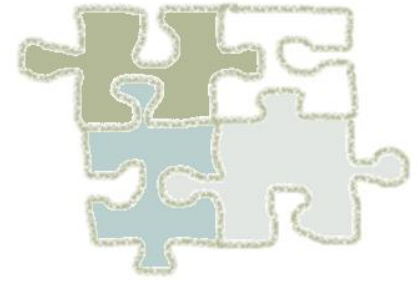
# Zusammenfassung: Operatoren



- Unär:
  - arithmetische z.B. A++
  - logische !B
  - Bit-Operatoren ~A
- Binär
  - arithmetische z.B. A + B
  - relationale z.B. A <= B
  - logische z.B. A && B
  - Bit-Operatoren z.B. A & B
  - Shift-Operatoren z.B. A >> B
  - Zuweisungsoperator, kombinierte Zuweisungsoperatoren  
z.B. A = B   A\* = B
- Ternär
  - Bedingungsoperator A ? B : C



# Zusammenfassung



- bei Auswertung mancher Ausdrücke (z.B.  $A++$ ,  $A += B$ ) – Nebeneffekte
- **Nebeneffekt:** Veränderung einer Variable, die „nebenbei“ während der Auswertung eines Ausdrucks stattfindet

# Überschriebene (polymorphe) und überladene Methoden

# Eindeutigkeit der Methode

**Methodenkopf** (= Methodendeklaration) in derselben Klasse immer **eindeutig**, d.h.

- *ein* Methodenkopf **kommt nur einmal** vor
- mehrere Methodenrumpfe (= Implementierungen) zum gleichen Methodenkopf **nicht erlaubt**

Aber:

gleicher Methodenkopf mit unterschiedlichem Methodenrumpf in anderer Klasse **erlaubt** → Methode **überschrieben**



Wozu kann das gut sein?

# print()-Methode in Person

```
public class Person {  
    private String name;  
    private String vorname;  
    private int alter;  
    // Konstruktor  
  
    public void print() {  
        System.out.println ("Name : " + name);  
        System.out.println ("Vorname : " + vorname);  
        System.out.println ("Alter : " + alter);  
    }  
}
```

# print()-Methode in Bruch

```
public class Bruch {  
    private int zaehler;  
    private int nenner;  
    // Konstruktor  
  
    public void print() {  
        System.out.print ("Wert des Quotienten von " +  
                           zaehler);  
        System.out.print (" und " + nenner + " ist " +  
                           zaehler + " / ");  
        System.out.println (nenner);  
    }  
}
```

# print()-Methode in Test

```
public class Test {  
    public static void main (String[] args) {  
        Bruch b;  
        b = new Bruch(1,2);  
        b.print();  
  
        Person p;  
        p = new Person("Müller", "Fritz", 35);  
        p.print();  
    }  
}
```

Der Wert des Quotienten von 1 und 2 ist 1 / 2  
Name : Müller  
Vorname : Fritz  
Alter : 35

Jedes Objekt „weiß“, zu welcher Klasse es gehört → eindeutige Zuordnung der aufgerufenen Methode möglich



# Polymorphe Methoden

`print()`: sog. **polymorphe** (überschriebene) Methode  
hat in *verschiedenen* Klassen **identischen Methodenkopf** und  
**gleiche Semantik**, jedoch **unterschiedliche Implementierung**

→ **Vorteil:** Verständlichkeit des Programms erhöht:

z.B. klar, dass `print()` in jeder Klasse zur Ausgabe des Objektes  
auf dem Bildschirm verwendet

**Polymorphie** (gr. *Vielgestaltigkeit*) von Methoden:

*Konzept*, nach dem verschiedene Implementierungen *einer*  
Methode (d.h. gleicher Methodenkopf und gleiche Semantik) in  
verschiedenen Klassen erlaubt sind



# Anzahl der Parameter

*Aufgabe:* Gesucht zwei Methoden, die den Durchschnitt von zwei bzw. drei int-Werten berechnen

```
public static double avg(int x, int y){  
    return (x + y) / 2;  
}
```

```
public static double avg(int x, int y, int z){  
    return (x + y + z) / 3;  
}
```

→ gleicher Name der Methode, **Anzahl** der (formalen) Parameter **unterschiedlich** → Methodename avg **überladen**



# Datentyp der Parameter

*Aufgabe:* Gesucht drei Methoden, die den Absolutwert einer long-, float- bzw. double-Zahl berechnen

vgl. `java.lang.Math`

```
public static long abs (long a) {  
    return (a >= 0) ? a : -a;  
}
```

```
public static float abs (float a) { ... }
```

```
public static double abs (double a) { ... }
```

→ gleicher Name der Methode, **Datentypen** der (formalen) Parameter **unterschiedlich** → Methodennamen **abs** **überladen**

# Überladene Methoden



`avg()`, `abs()`: sog. **überladene Methode**

hat in *derselben* Klasse die **gleiche Semantik** und **denselben Namen**, aber **verschiedene Parameterliste** (*Anzahl* oder *Datentypen* der Parameter verschieden) – und abweichende Implementierung

→ **Vorteil:** Verständlichkeit des Programms erhöht

z.B. klar, dass `abs()` immer den Absolutwert berechnet – für verschiedene DT

# Signatur der Methode

- bei überladenen Methoden(namen): die richtige Methode beim Aufruf anhand der Parameterliste vom Compiler erkannt
- Methode durch ihre **Signatur** eindeutig identifizierbar:

**Signatur = Methodenname + Parameterliste**

→ Rückgabetyp gehört in Java **nicht** zur Signatur






# Parameterliste variabler Länge

Parameterliste hat zwei Teile:

- Liste fester Länge – immer am Anfang
- **varargs**: Liste variabler Länge (aber von **gleichem** Datentyp) – immer am Ende, Syntax: **datentyp... variable**

*Beispiel:*

```
public void metV(int k, byte b,String... str){  
    System.out.println(k + " " + b);  
    for(int i=0; i < str.length ; i++)  
        System.out.print(str[i]);  
}
```

wie String[]  
behandelt

Aufruf in main():

```
metV(10,(byte)3,"Das ", "ist ", "ein ", "Test.");
```

10 3

Das ist ein Test.

# Vorteil von varargs

**varargs** entspricht einem **Array**:

**datentyp... x**      $\approx$      **datentyp[] x**

Vorteil von **varargs** gegenüber **Array**?



komfortabler, da

- Übergabewerte einfach als aktuelle Parameter beim Methodenaufruf angeben
- nicht nötig, ein Array zuvor anzulegen und zu initialisieren

# Parameter von main()

main()-Methode:

```
public static void main(String[] args)
```

wozu?

→ in Java möglich, Parameter (als String-Objekte) über die Kommandozeile an ein Programm zu übergeben

# Parameter von main(): Beispiel

// Datei: StringTest.java

```
public class StringTest {
```

```
    public static void main (String[] args){
```

```
        String a = "HTW";
```

```
        String b = args[0];
```

```
        if (a.equals (b))
```

```
            System.out.println ("OK");
```

```
        else
```

```
            System.out.println ("Nicht OK");
```

```
    }
```

```
}
```

überprüft, ob der 1. per  
Kommandozeile angegebener  
Argument gleich „HTW“ ist

Aufruf des Programms: `java StringTest HTW`

OK

Parameterübergabe in Eclipse:

Run → Run Configurations... → <sup>(x)</sup>=Arguments → Program arguments

# Verständnisfragen



- Was ist eine **polymorphe** Methode? Was sind die Vorteile der Polymorphie?
- Was ist eine **überladene** Methode? Wozu ist Überladen gut?
- Wodurch wird beim Aufruf eine Methode eindeutig identifizierbar?
- Muss in Java eine Methode konstant viele Parameter haben?
- Was ist bei varargs zu beachten?

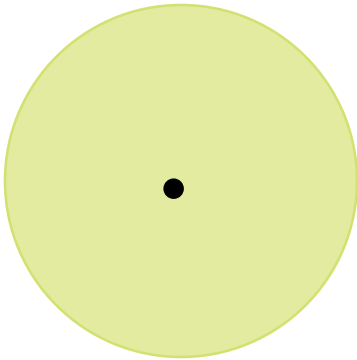


# Vererbung

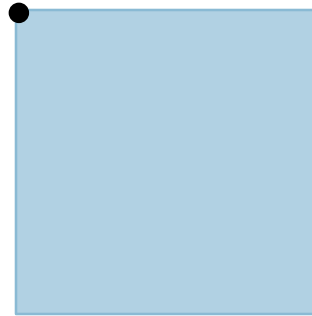
# Klassen gesucht

vorhanden:  
Klasse Punkt  
• (x,y)

Kreis



Quadrat



Rechteck



# Klasse Kreis

```
public class Kreis {  
    private Punkt anker;  
    private int radius;
```

```
    public Kreis(int radius, Punkt anker) {  
        this.radius = radius;  
        this.anker = anker;  
    }
```

```
    public void setAnker(Punkt p) { anker = p; }  
    public Punkt getAnker(){ return anker; }  
    public void verschiebeAnker(int x, int y) {  
        anker.verschiebe(x,y);  
    }  
    public double berechneFlaeche(){...}
```

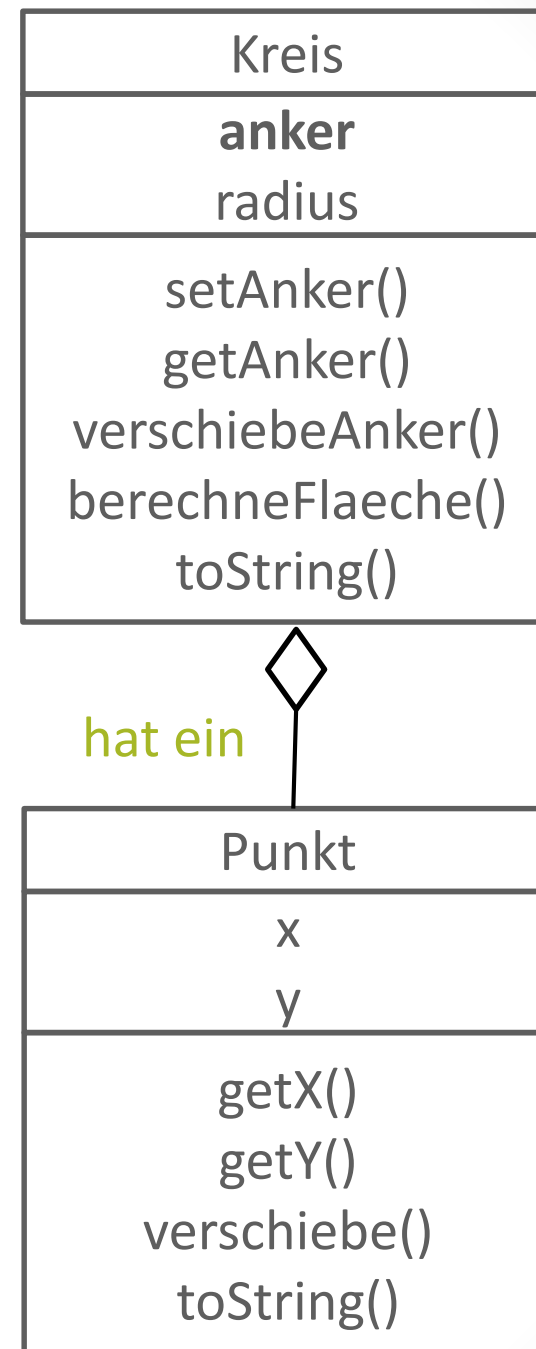
```
public class Punkt {  
    private int x;  
    private int y;  
    public Punkt(int u, int v){...}  
    public int getX(){return x;}  
    public int getY(){return y;}  
    public void verschiebe(int x,  
                           int y){...}  
    public String toString() {...}  
}
```

# Aggregation

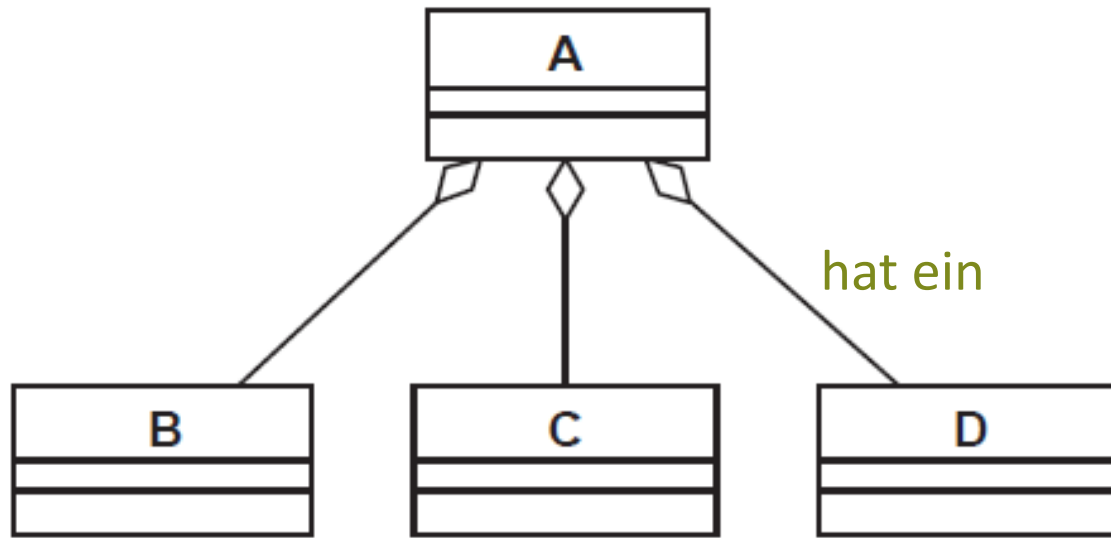
Aggregation:

„Groß“-Objekt hat als Datenfeld  
eine **Referenz** auf „Klein“-Objekt

*Beispiel:* Kreis hat ein Punkt



# Aggregation: Schema



# Aggregation: Lebensdauer

```
public class KreisTest {  
    public static void main (String[] args) {  
        Punkt p = new Punkt(1,2);  
        Kreis k = new Kreis(5,p);  
    }  
}
```

→ bei der **Aggregation**: **Lebensdauer** der Objekte **p** und **k** **entkoppelt**, d.h. wenn Objekt **k** nicht mehr existiert, kann Objekt **p** trotzdem weiter leben

# Exkurs: Komposition

## Komposition:

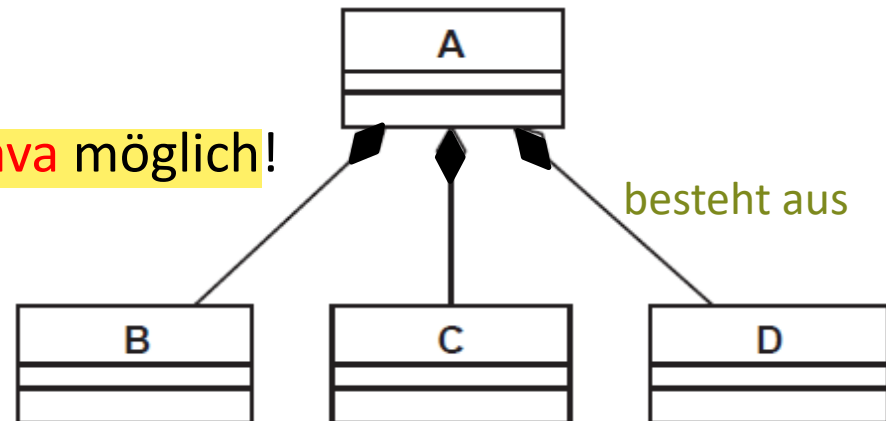
„Groß“-Objekt **gross** hat als Datenfeld ein **Objekt klein** vom Typ „Klein“-Objekt

→ bei der **Komposition**: **Lebensdauer** der Objekte **gross** und **klein gekoppelt**, d.h. wenn Objekt **gross** nicht mehr lebt, dann lebt Objekt **klein** auch nicht mehr

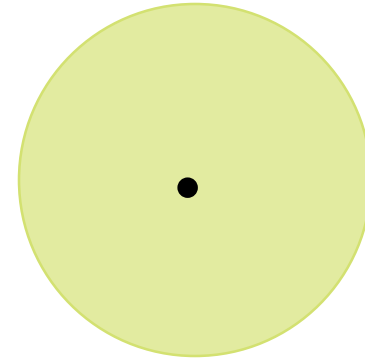


## Komposition:

in C++, aber **nicht in Java** möglich!



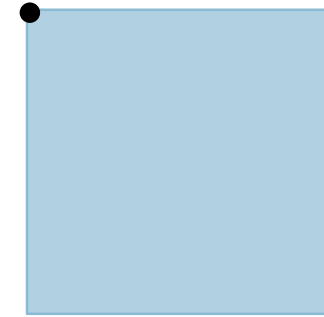
# Klasse Kreis



```
public class Kreis {  
    private Punkt anker;  
    private int radius;  
  
    public Kreis(int radius, Punkt anker) {  
        this.radius = radius;  
        this.anker = anker;  
    }  
  
    public void setAnker(Punkt p) { anker = p; }  
    public Punkt getAnker(){ return anker; }  
    public void verschiebeAnker(int x, int y) {  
        anker.verschiebe(x,y);  
    }  
  
    public double berechneFlaeche(){...}
```



# Klasse Quadrat



```
public class Quadrat {  
    private Punkt anker;  
    private int breite;  
  
    public Quadrat(int st, Punkt p) {  
        breite = st;  
        anker = p;  
    }
```

```
    public void setAnker(Punkt p) { anker = p; }  
    public Punkt getAnker(){ return anker; }  
    public void verschiebeAnker(int x, int y) {  
        anker.verschiebe(x,y);  
    }
```


```
    public double berechneFlaeche(){...}
```

# Klasse Figur

```
public class Figur {  
    private Punkt anker;  
  
    public Figur(Punkt p) {  
        anker = p;  
    }  
}
```

```
public void setAnker(Punkt p) { anker = p; }  
public Punkt getAnker(){ return anker; }  
public void verschiebeAnker(int x, int y) {  
    anker.verschiebe(x,y); }  
}
```

# Klasse `Kreis` abgeleitet

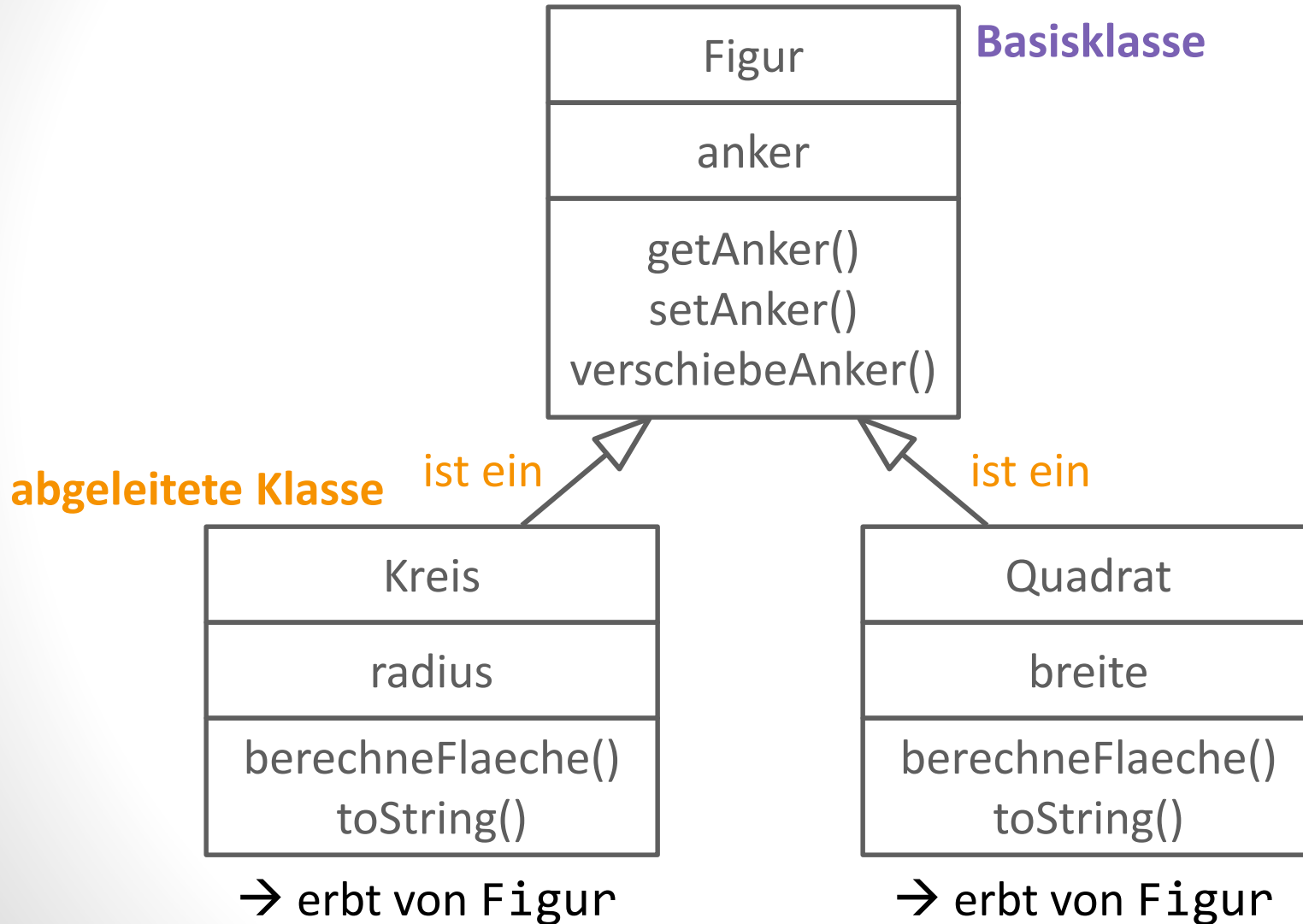
```
public class Kreis extends Figur {  
    private Punkt anker;  
    private int radius;  
  
    public Kreis(int radius, Punkt anker) {  
        super(anker);  Aufruf des Konstruktors  
        this.radius = radius; der Superklasse  
    }  
  
    public void setAnker(Punkt p) { anker = p; }  
    public Punkt getAnker(){ return anker; }  
    public void verschiebeAnker(int x, int y) {  
        anker.verschiebe(x,y);  
    }  
  
    public double berechneFlaeche(){...}
```

# Klasse Quadrat abgeleitet

```
public class Quadrat extends Figur {  
    private Punkt anker;  
    private int breite;  
  
    public Quadrat(int st, Punkt p) {  
        super(p);  
        breite = st;  
    }  
  
    public void setAnker(Punkt p) { anker = p; }  
    public Punkt getAnker(){ return anker; }  
    public void verschiebeAnker(int x, int y) {  
        anker.verschiebe(x,y);  
    }  
  
    public int berechneFlaeche(){...}
```

← Aufruf des Konstruktors der Superklasse

# Vererbung

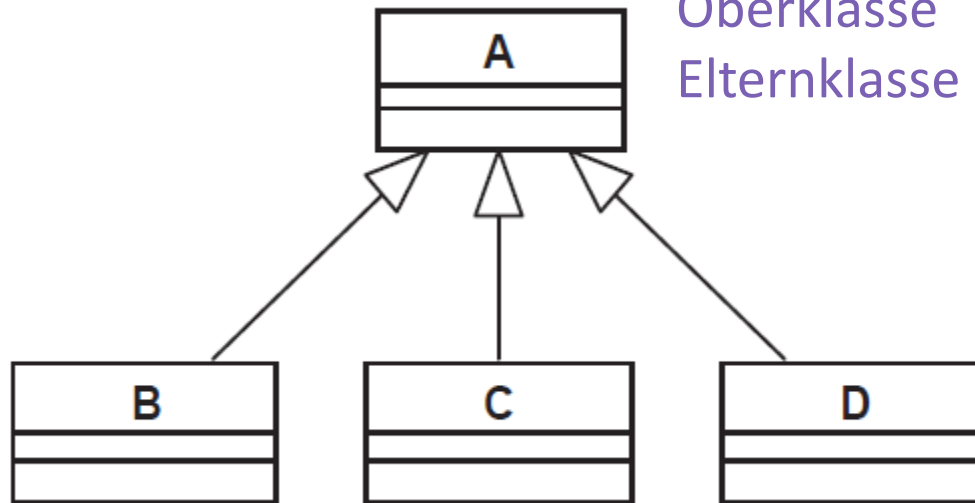


# Vererbung: Schema

**Basisklasse**

auch: **Superklasse**

Oberklasse  
Elternklasse



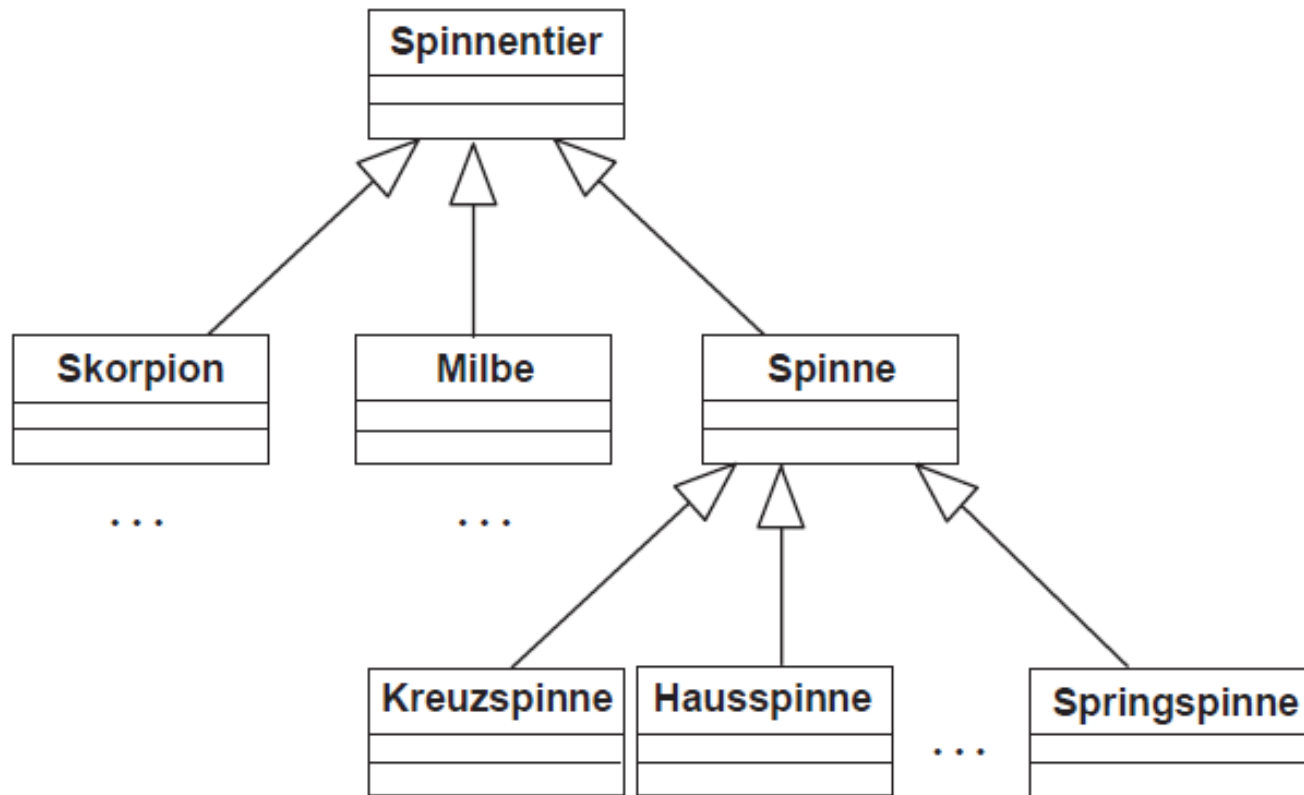
**abgeleitete Klasse**

auch: **Subklasse**

Unterklasse

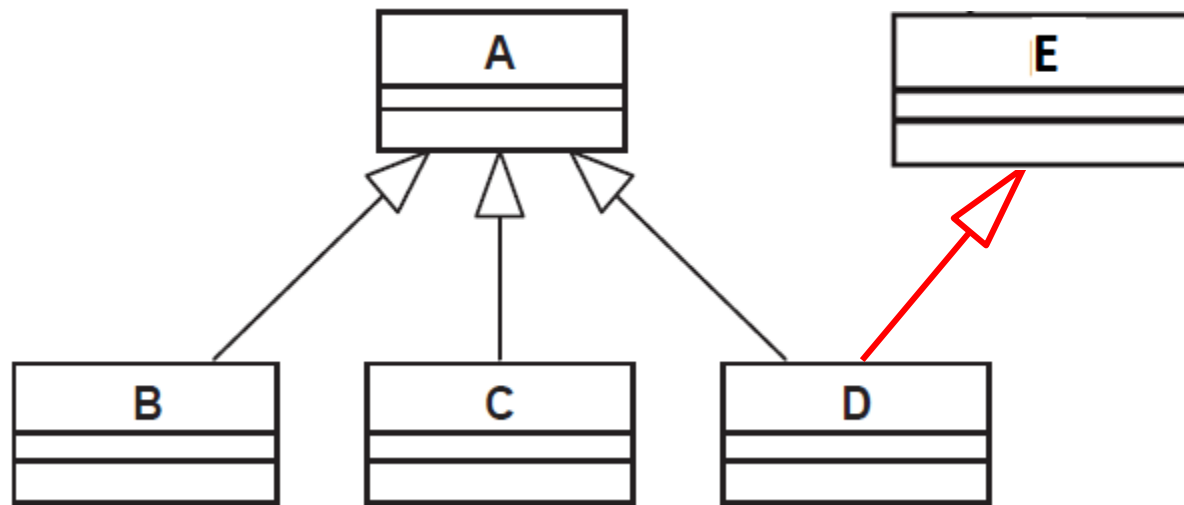
Kindklasse

# Vererbung: Beispiel



*Beispiel:* eine Kreuzspinne **ist eine** Spinne **ist ein** Spinnentier

# Keine Mehrfachvererbung



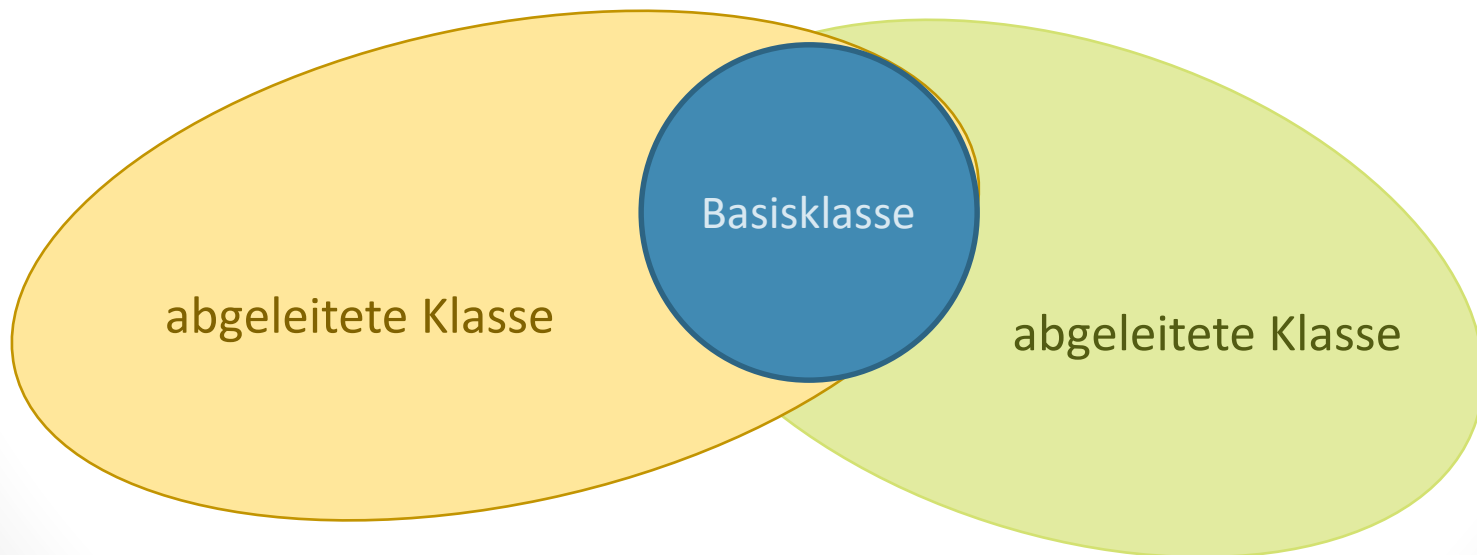
in Java: keine Mehrfachvererbung möglich!



# Vererbung: abgeleitete Klasse

abgeleitete Klasse:

- **erbt** Struktur (Attribute) und Verhalten (Methoden) von seiner Basisklasse
- **erweitert** die Basisklasse durch Definition zusätzlicher Attribute und Methoden



# Vererbung: Vorteile

Vererbung = gemeinsame Eigenschaften (Struktur und Verhalten) mehrerer Klassen – in gemeinsame Oberklasse ausgelagert →

Vorteile:

- Wiederholung des Programmcodes vermieden → Teile des Codes *wiederverwendet* → Komplexität reduziert
- Programmstruktur übersichtlicher

# Verständnisfragen



- Was ist Aggregation?
- Was ist der Unterschied zwischen Aggregation und Komposition? Wie sieht Komposition in Java aus?
- Mit welchem Schlüsselwort wird die Vererbung in Java definiert?
- In welcher Beziehung steht die Subklasse (abgeleitete Klasse) zur Superklasse (Basisklasse)?
- Kann in Java eine Klasse von zwei unterschiedlichen Klassen erben?
- Hat eine abgeleitete Klasse mehr oder weniger Attribute und Methoden als ihre Basisklasse?
- Was sind die Vorteile des Vererbungskonzeptes?

# Subtyping

- Basisklasse A: definiert einen *Datentyp* A
- abgeleitete Klasse: erbt von Basisklasse, definiert einen *Untertyp* von A

→ **Subtyping**:

durch Vererbung wird ein **Untertyp** eines vorhandenen *Datentyps* definiert

Objekt der abgeleiteten Klasse **ist** auch ein Objekt der Basisklasse

→ Objekt der abgeleiteten Klasse **polymorph**

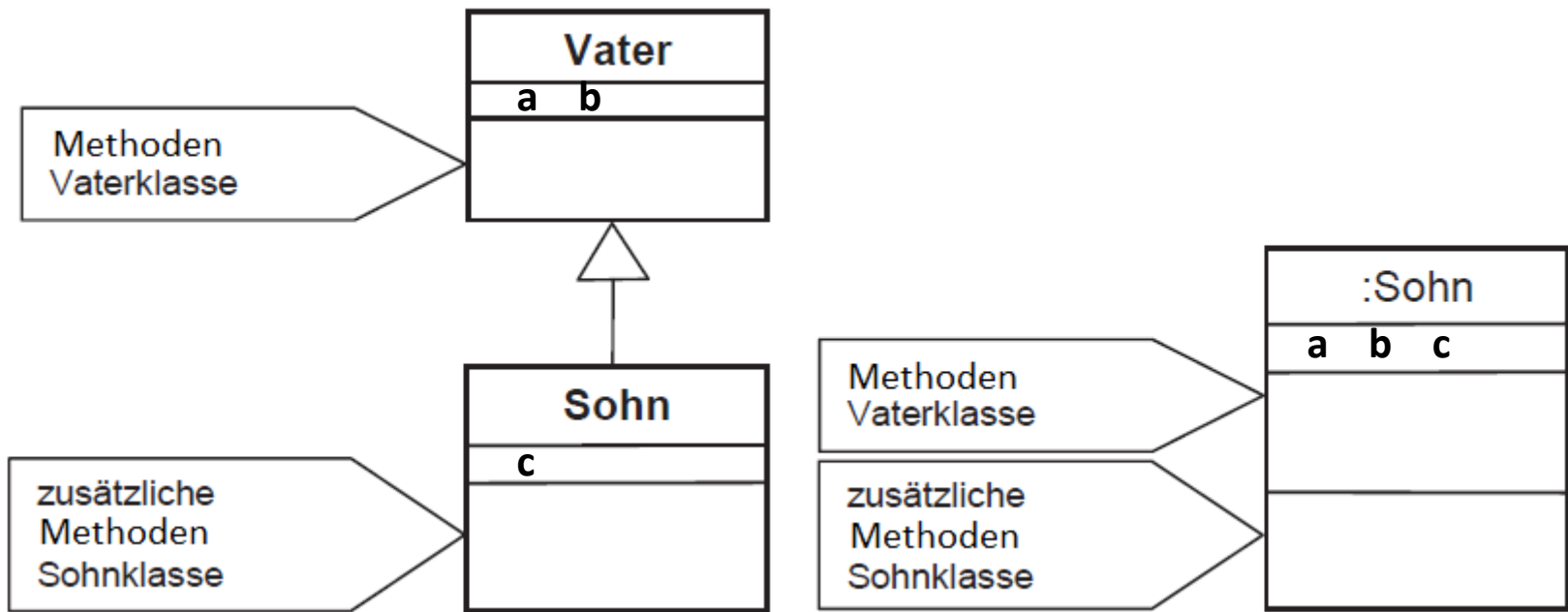
# Was wird vererbt?



Sohnklasse **erbt** von der Vaterklasse (fast) **ALLES**:

- Instanzvariablen + Instanzmethoden
- Klassenvariablen + Klassenmethoden

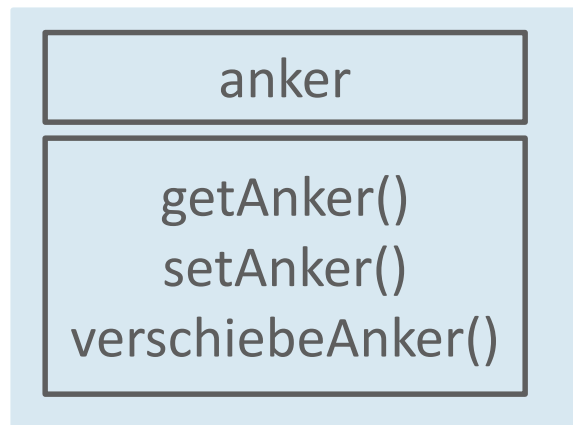
→ **Ausnahme: Konstruktor wird nicht vererbt**



# Was wird vererbt: Beispiel

## Beispiel:

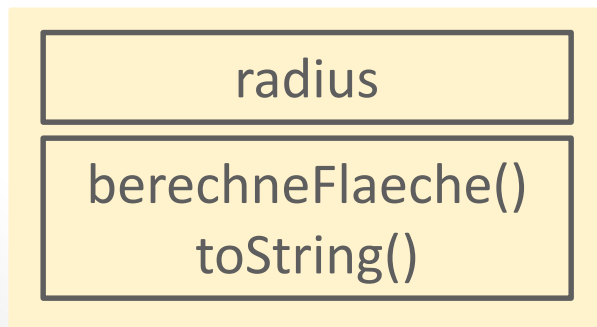
jedes Objekt der Klasse `Kreis` hat:



von Klasse `Figur`  
geerbte Datenfelder

von Klasse `Figur`  
geerbte Methoden

durch  
Konstruktor  
von `Figur`  
initialisiert



eigene Datenfelder

eigene Methoden

durch *eigenen*  
Konstruktor  
initialisiert

# Konstruktor

```
public class Kreis extends Figur {  
  
    private int radius;  
  
    public Kreis(int radius, Punkt anker) {  
        super(anker);  
        this.radius = radius;  
    }  
    ...  
}
```

← Aufruf des Konstruktors  
der Superklasse **Figur**

```
public class Figur {  
    private Punkt anker;  
  
    public Figur(Punkt p) {  
        anker = p;  
    }  
    ...  
}
```

# Beispiel: vererbte Methoden

```
public class TestFigur {  
    public static void main (String[] args) {  
        Punkt p = new Punkt(1,2);  
        Kreis k = new Kreis(5,p);
```

Zugriff auf eigene Methode: über **Kreis**-Objekt

```
        System.out.println(k.berechneFlaeche());
```

Zugriff auf von **Figur** geerbte Methoden: über **Kreis**-Objekt

```
        k.verschiebeAnker(2,2);
```

```
        System.out.println(k.getAnker());
```

```
    }
```

```
}
```



# Beispiel: vererbte Attribute

```
public class Kreis extends Figur {  
    private int radius;  
    ...  
}
```

direkter Zugriff auf eigene Datenfelder:

**radius**

**k.radius**

wenn **k** ein Objekt vom Typ **Kreis**



direkter Zugriff auf geerbte Datenfelder:

**anker**

**k.anker**

wenn **k** ein Objekt vom Typ **Kreis**  
möglich?



→ **nicht möglich**, da **anker** in **Figur** als **private** definiert

# Vererbung und private

Tochterklasse erbt von der Mutterklasse **ALLES**  
→ auch **private** Attribute und Methoden, aber:



**nicht alles** Vererbte in der  
Tochterklasse automatisch  
**sichtbar** (= zugreifbar)!

d.h. **private** Attribute und Methoden der  
Mutterklasse in der Tochterklasse **nicht sichtbar**

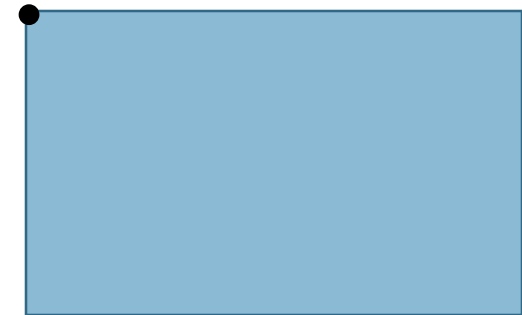
# Was ist mit dem Rechteck?



Datenfelder von Rechteck?

- was gemeinsam mit Figur?
- was gemeinsam mit Quadrat?

Rechteck



Rechteck
<del>anker</del> <del>breite</del> laenge
<del>getAnker() setAnker()</del> <del>verschiebeAnker()</del> berechneFlaeche() toString()

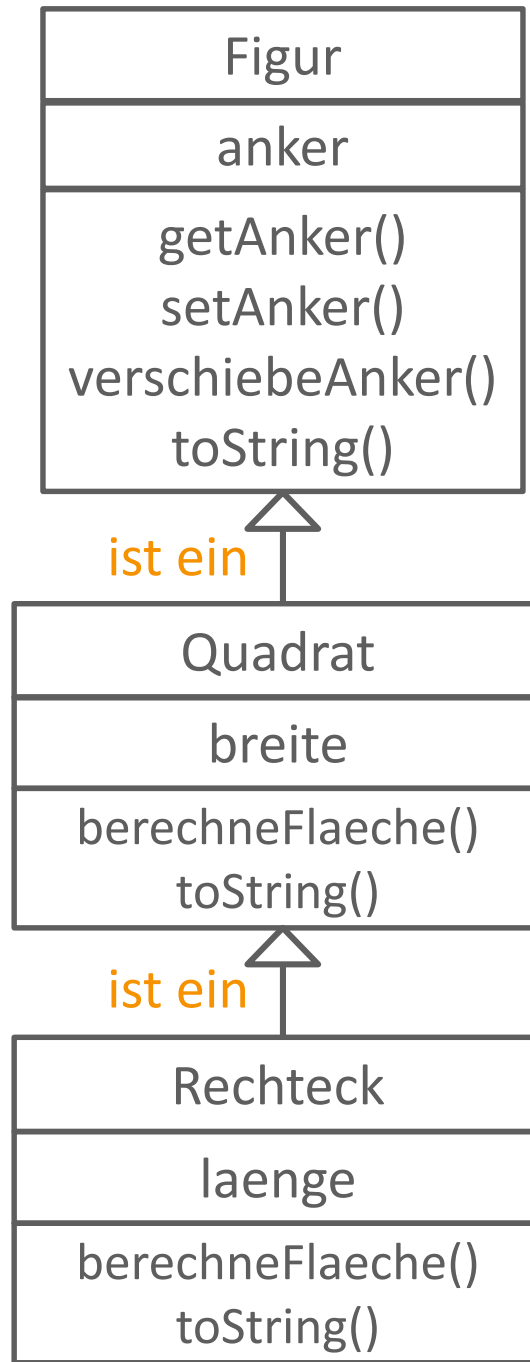
Rechteck hat ein weiteres Attribut laenge → erste (naive) Implementierung: Rechteck erbt vom Quadrat

# Klasse Rechteck abgeleitet

```
public class Rechteck extends Quadrat {  
    private int laenge;  
    //Konstruktor  
    ...  
    public int berechneFlaeche(){...}  
    public String toString(){...}  
}
```

# Vererbungshierarchie

Generalisierung



Spezialisierung

hier:  
Rechteck ist ein Quadrat...

# Klasse Object

**Object** = Mutter aller Klassen

→ *jede* Klasse und *jedes* Array automatisch von Klasse **Object** abgeleitet → erbt alle Methoden der Klasse **Object**

- `public String toString()`
- `public boolean equals (Object obj)`  
liefert bei `x.equals(y)` den Wert `true` zurück, wenn `x` und `y` Referenzen auf dasselbe Objekt sind
- `protected Object clone() throws CloneNotSupportedException`  
erlaubt es, eine Kopie eines Objektes zu erzeugen

# Verständnisfragen



- Was sind die Vorteile von Vererbung?
- Was ist Subtyping?
- Was wird genau an die Kindklasse vererbt?
- Wird der Konstruktor der Elternklasse an die Kindklasse vererbt?
- Wie greift man in der Kindklasse auf die (geerbten) privaten Methoden der Elternklasse zu?
- Was ist eine Vererbungshierarchie?
- Welche Java-Klasse ist die erste in der Vererbungshierarchie?
- Ist eine abgeleitete Klasse eine Spezialisierung oder Generalisierung der Basisklasse?

# Links

- Java Basics - Increment Decrement Operators  
<https://www.youtube.com/watch?v=o8PkhmEZ5AI>
- Java Tutorial -14- für Anfänger "Inkrement Dekrement Operatoren" [HD] Deutsch  
[https://www.youtube.com/watch?v=S8\\_DYMG173c](https://www.youtube.com/watch?v=S8_DYMG173c)
- Vererbung in Java für Anfänger  
<https://www.youtube.com/watch?v=XICQ8TFOgUM>
- Java Crashkurs für Anfänger in 3 Std [15/21] | VERERBUNG  
<https://www.youtube.com/watch?v=CErVXcyTIZ8>
- This Keyword in Java - How to use "this"  
<https://www.youtube.com/watch?v=CSWrefLneXE>