252-0027

Einführung in die Programmierung

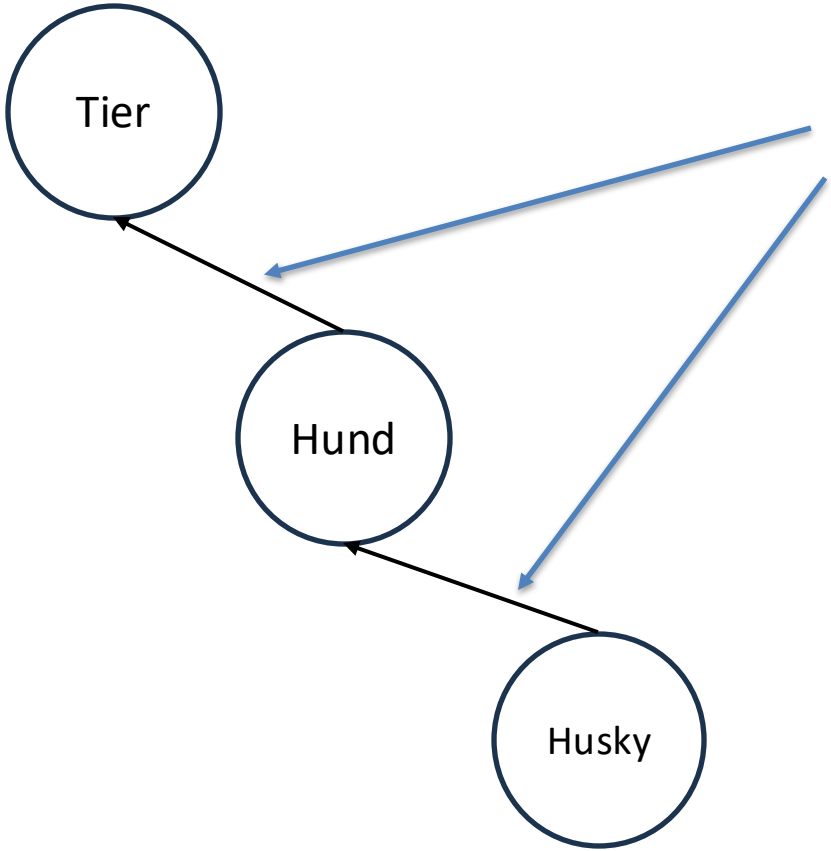
Übungen

Woche 12: Vererbung III

Henrik Pätzold
Departement Informatik
ETH Zürich

Konstruktoren

Konstruktoren



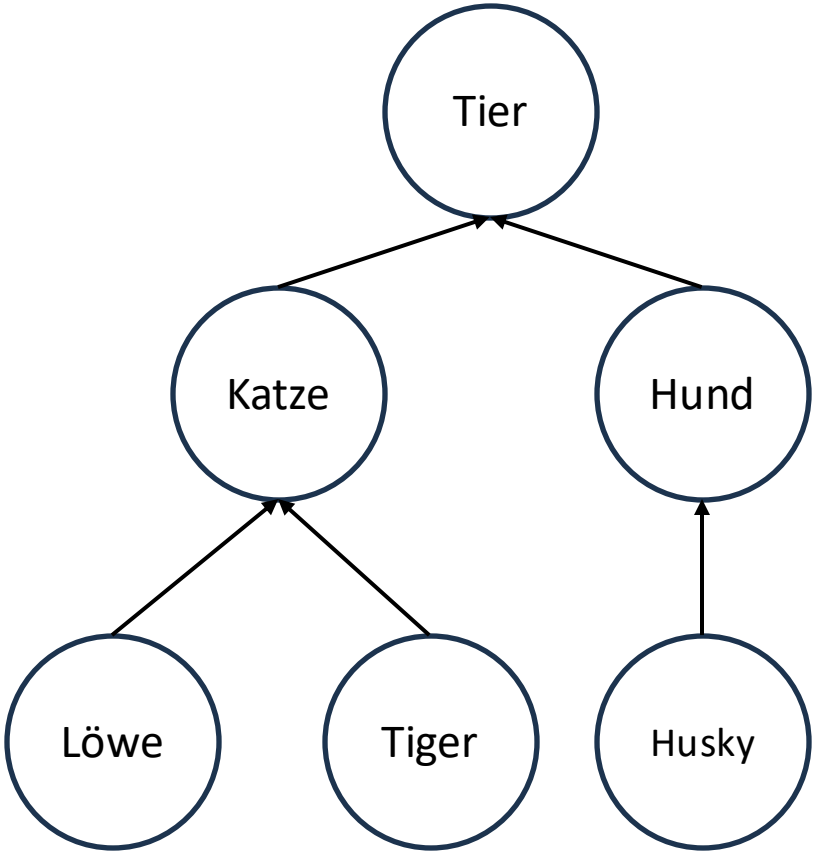
Die Pfeile sind eine “erbt von” - Beziehung

- Ein Hund ist ein Tier.
- Nicht alle Tiere sind ein Hund.

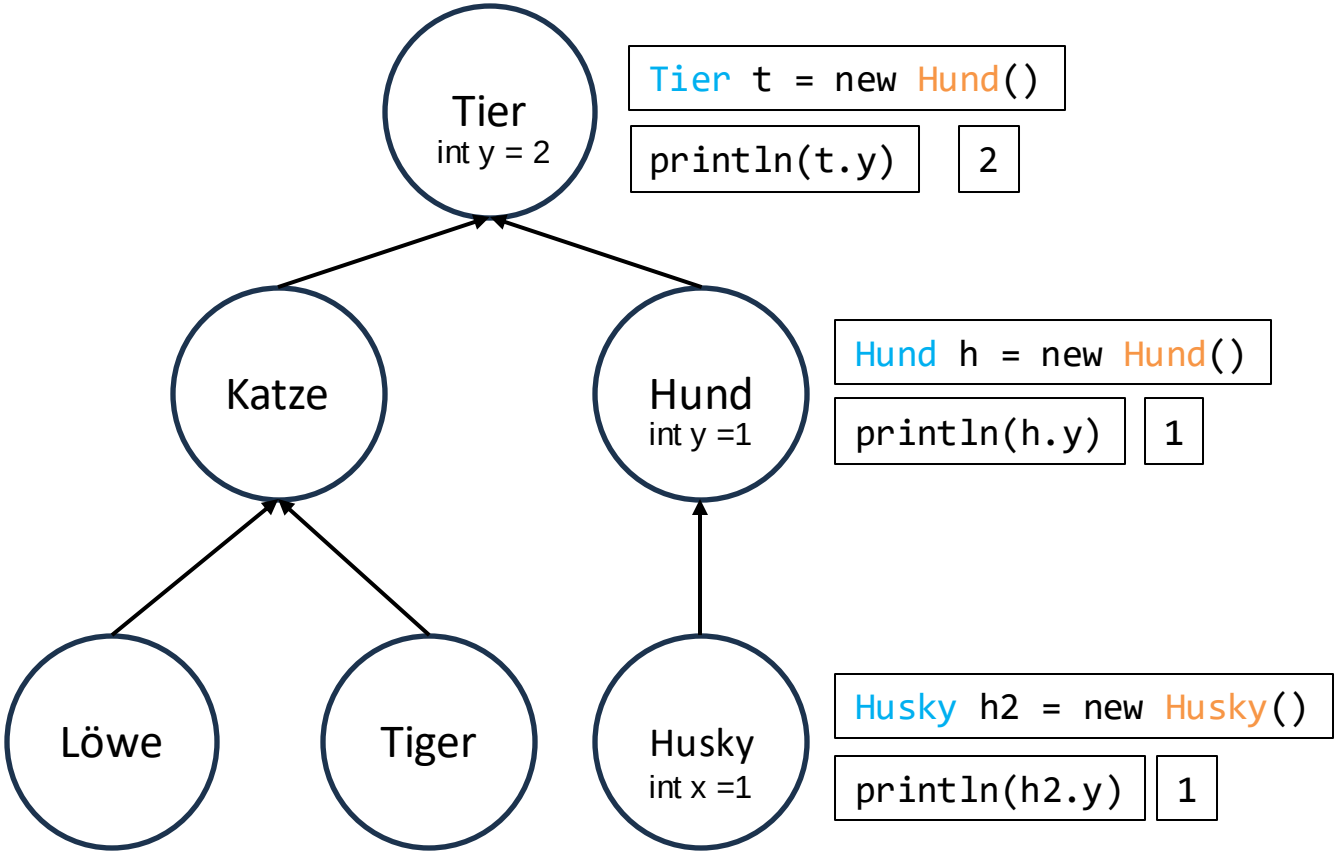
- Ein Husky ist ein Tier und ein Hund.

- Wir wollen sichergehen, dass Attribute richtig vererbt werden

Konstruktoren - Beispiel



Konstruktoren - Beispiel

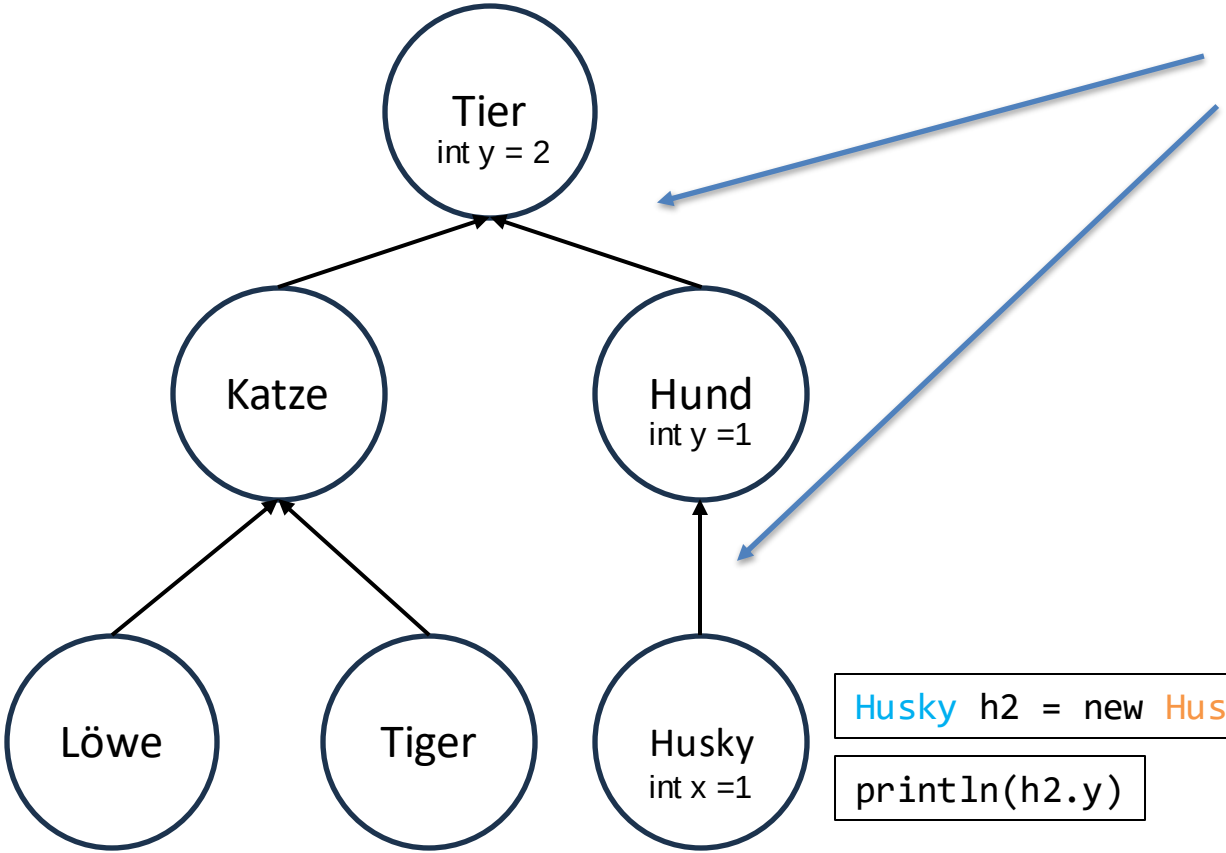



```
public Tier(){
    this.y = 2;
}
```

```
public Hund(){
    this.y = 1;
}
```

```
public Husky(){
    this.x = 1;
}
```

Konstruktoren - Beispiel



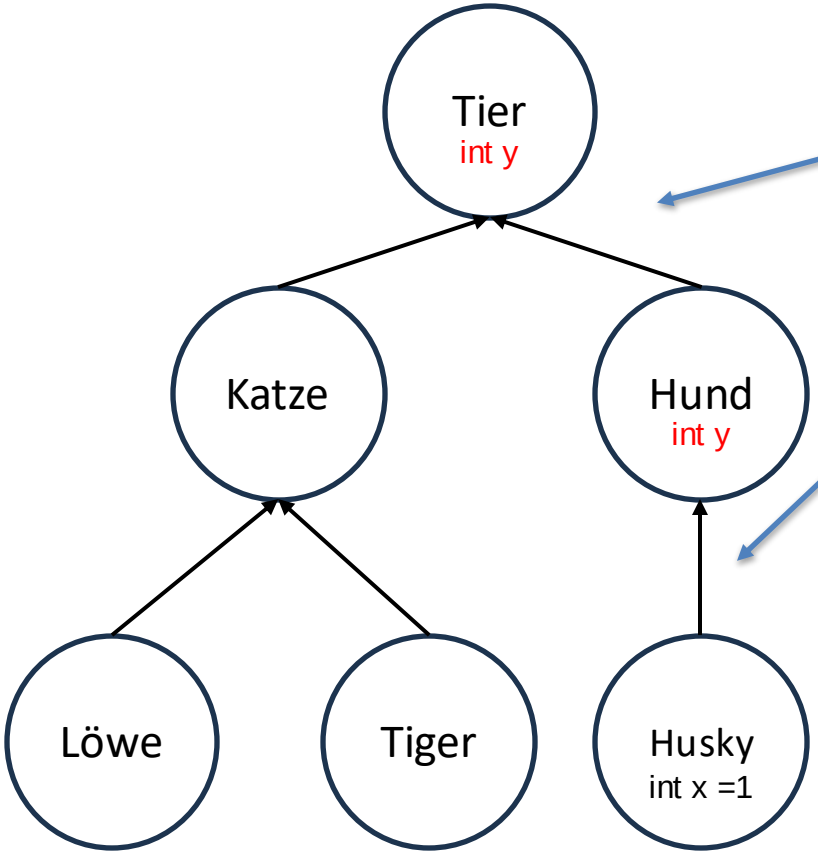
Die Pfeile sind eine “erbt von” - Beziehung

- Wir wollen auf die Variable des statischen Typen zugreifen
- Diese ist hier aber nicht explizit deklariert worden.

```
Husky h2 = new Husky()
println(h2.y)
```

```
public Husky(){
    this.x = 1;
}
```

Konstruktoren - Beispiel



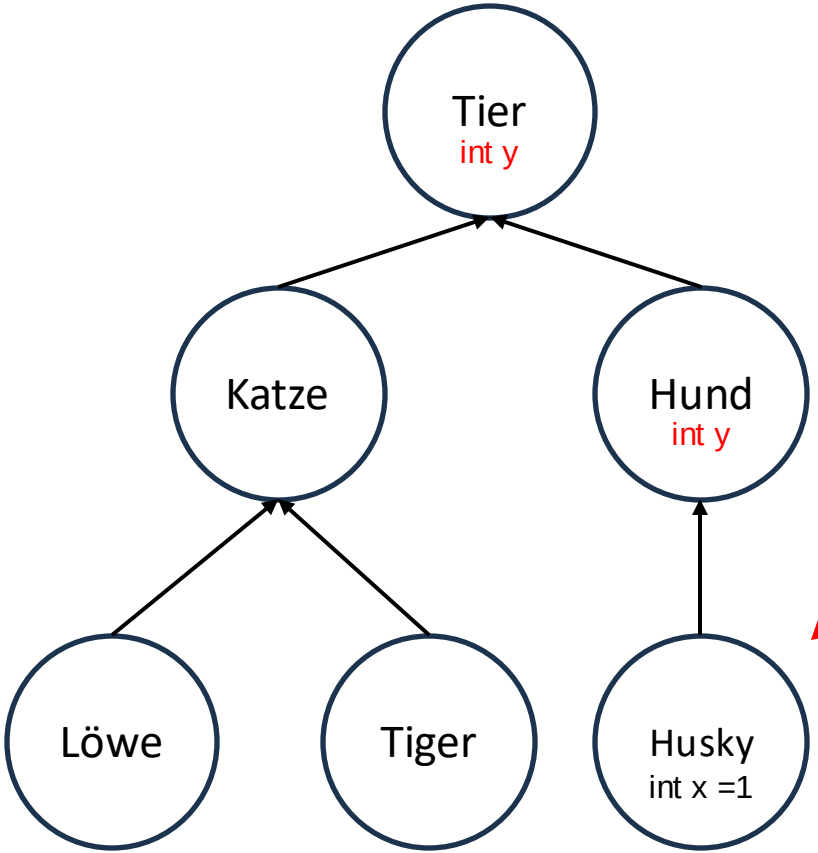
Die Pfeile sind eine “erbt von” - Beziehung

- Wir wollen auf die Variable des statischen Typen zugreifen
- Diese ist hier aber nicht explizit deklariert worden.

- Konstruktoren setzen Variablen auf spezifische, oder im Notfall, Standard werte. (null oder typ-spez.)

```
Husky h2 = new Husky()
println(h2.y)
```

Konstruktoren - Beispiel



Die Pfeile sind eine “erbt von” - Beziehung

- Wir wollen auf die Variable des statischen Typen zugreifen
- Diese ist hier aber nicht explizit deklariert worden.

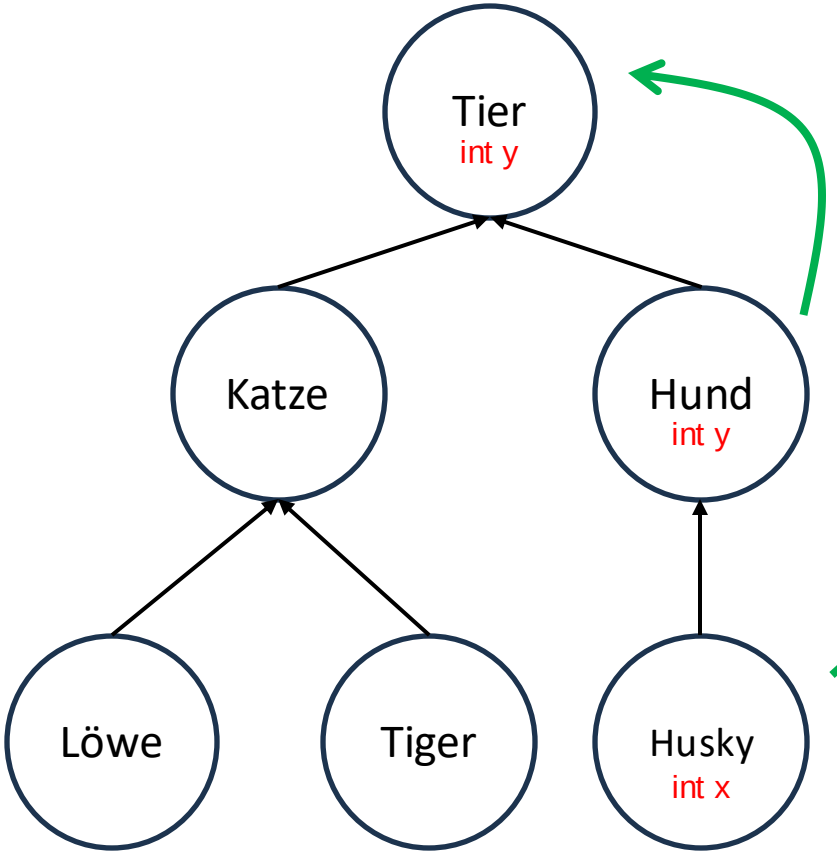
Wie können wir etwas vererben, was nicht gesetzt wurde?

```
Husky h2 = new Husky()
println(h2.y)
```

Konstruktoren

- **Instanziierung von Subklassen:**
  - Erfordert das vorherige Ausführen des Superklassen-Konstruktors.
- **Default-Konstruktor:**
  - Ruft automatisch den Konstruktor der Superklasse auf.
- **Zweck:**
  - Aufrufen der Superklassen-Konstruktoren stellen sicher, dass alle Attribute korrekt initialisiert werden.
  - Sie bereiten das Objekt so vor, dass es direkt genutzt werden kann.

Konstruktoren - Beispiel



super()

super()

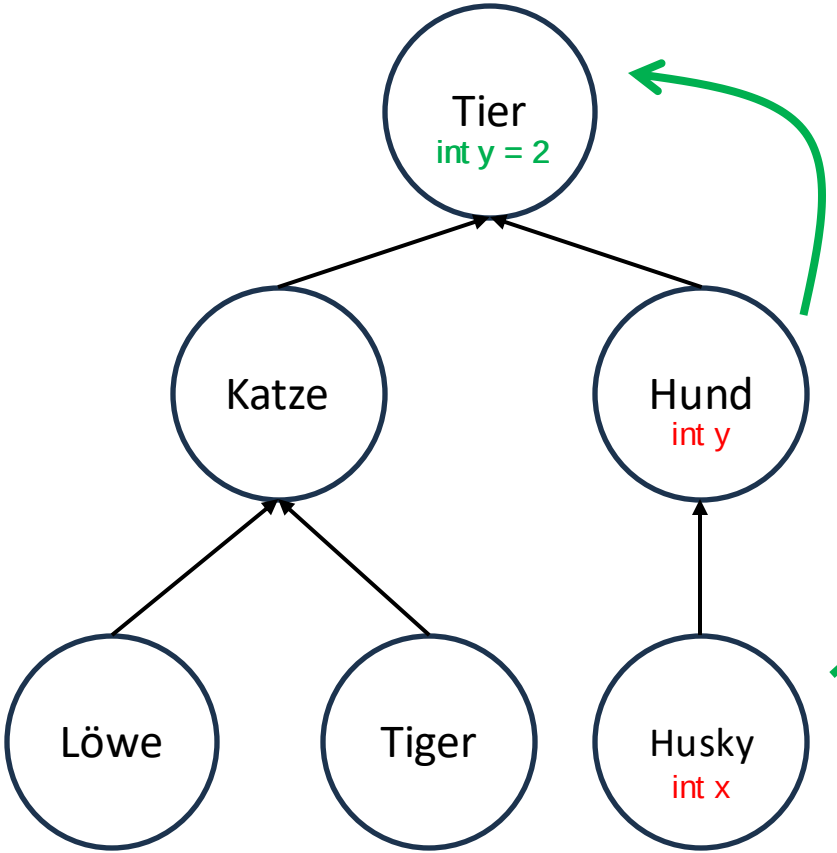
```
Husky h2 = new Husky()
println(h2.y)
```

```
public Tier(){
    this.y = 2;
}
```

```
public Hund(){
    this.y = 1;
}
```

```
public Husky(){
    this.x = 1;
}
```

Konstruktoren - Beispiel



super()

super()

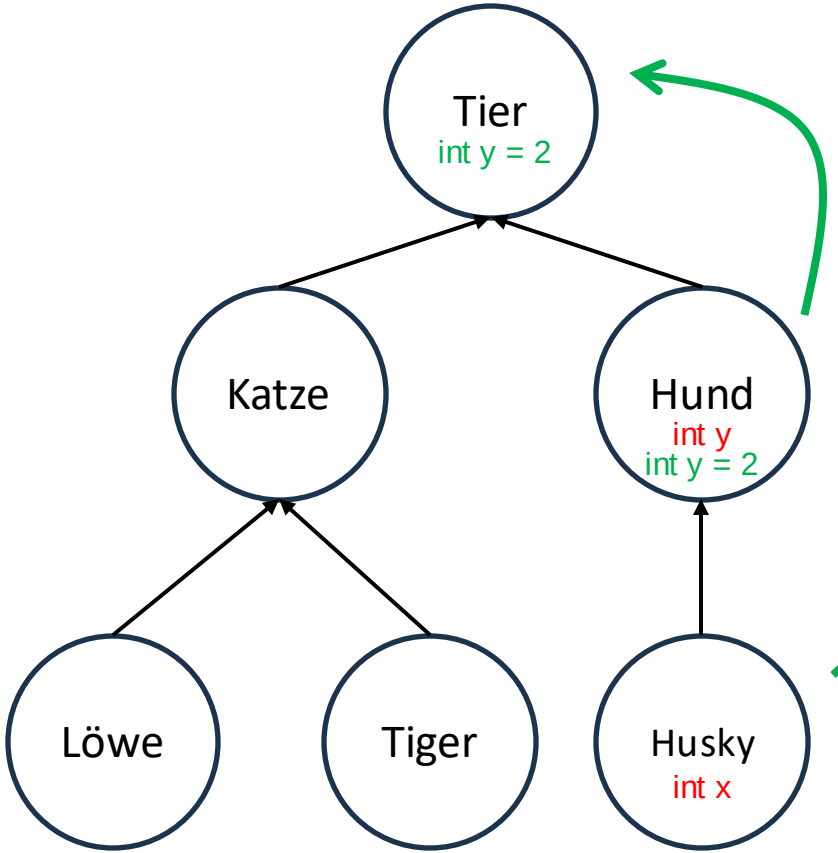
```
Husky h2 = new Husky()
println(h2.y)
```

```
public Tier(){
    this.y = 2;
}
```

```
public Hund(){
    this.y = 1;
}
```

```
public Husky(){
    this.x = 1;
}
```

Konstruktoren - Beispiel



super()

super()

```
Husky h2 = new Husky()
```

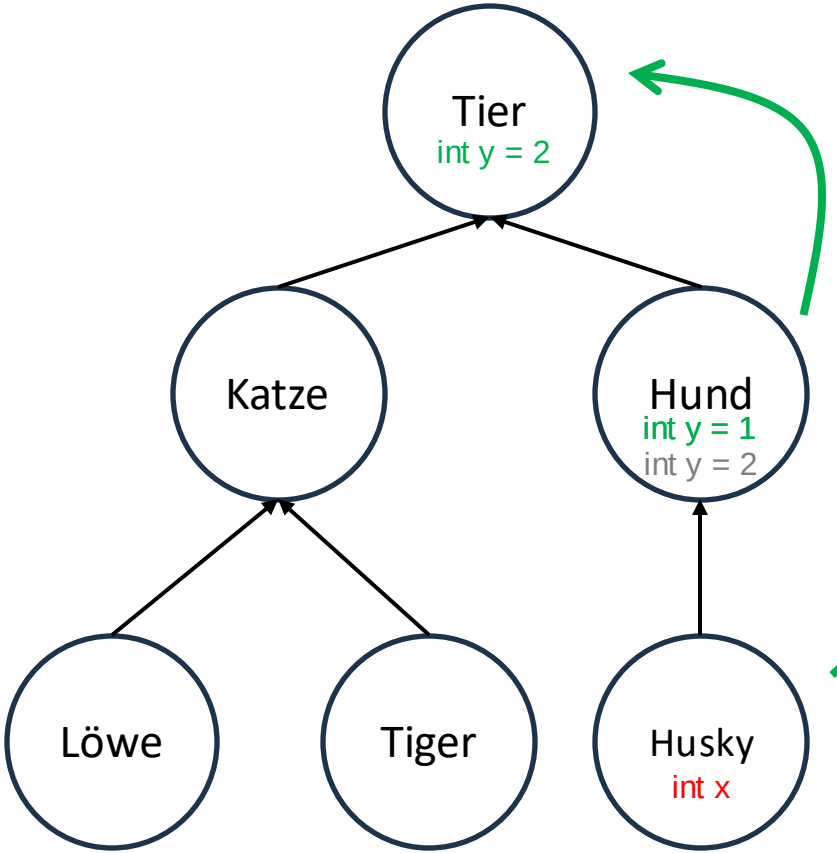
```
println(h2.y)
```

```
public Tier(){
    this.y = 2;
}
```

```
public Hund(){
    this.y = 1;
}
```

```
public Husky(){
    this.x = 1;
}
```

Konstruktoren - Beispiel



super()

super()

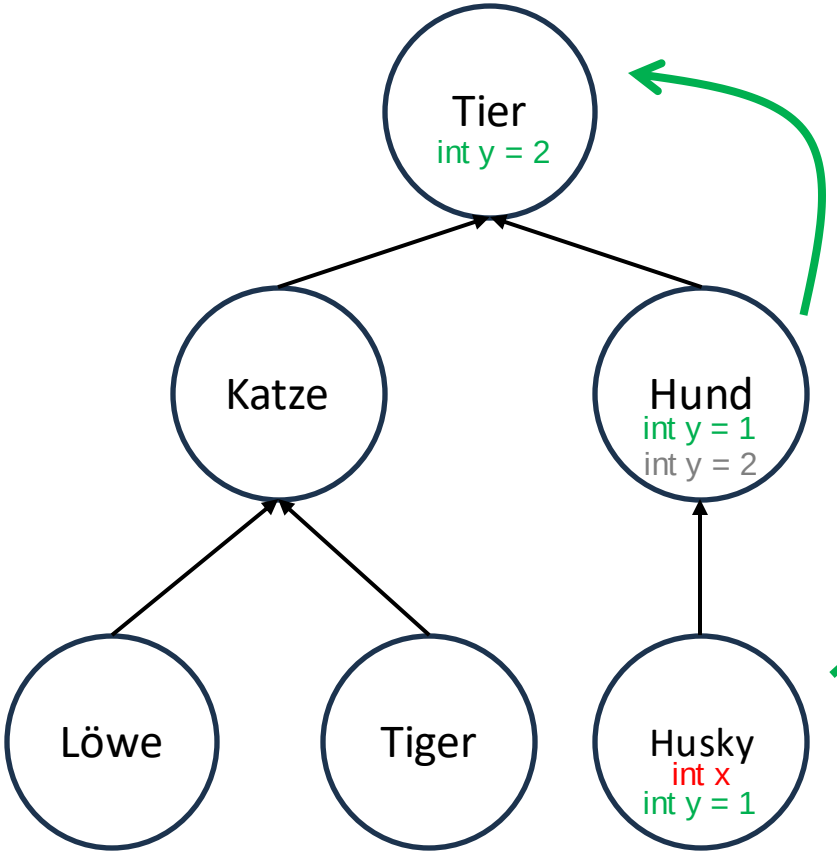
```
Husky h2 = new Husky()
println(h2.y)
```

```
public Tier(){
    this.y = 2;
}
```

```
public Hund(){
    this.y = 1;
}
```

```
public Husky(){
    this.x = 1;
}
```

Konstruktoren - Beispiel



super()

super()

```
Husky h2 = new Husky()
```

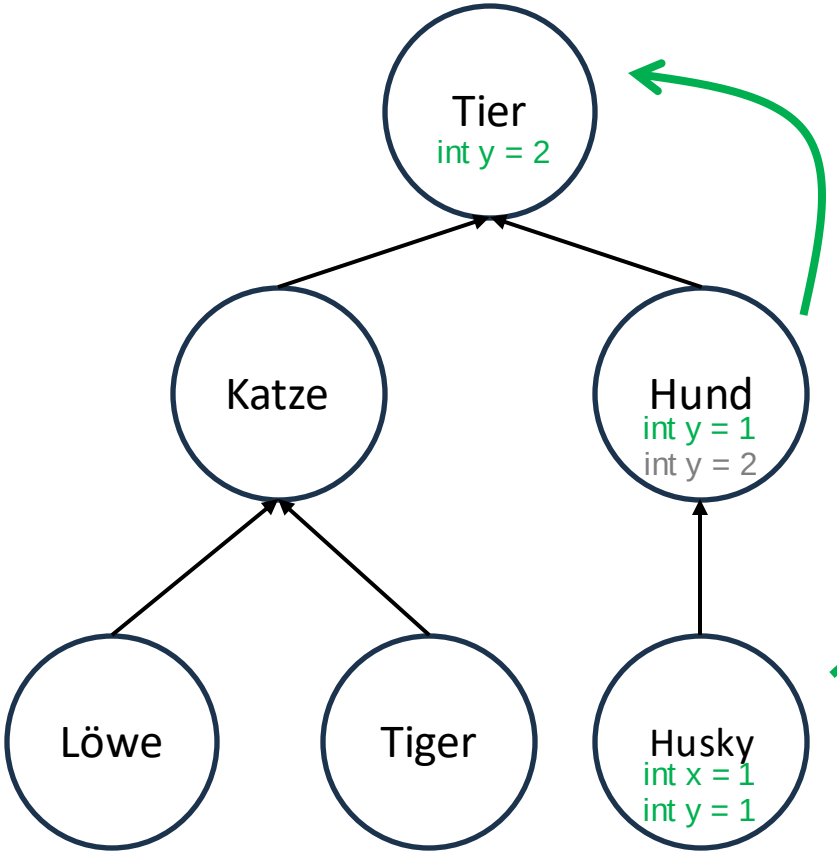
```
println(h2.y)
```

```
public Tier(){
    this.y = 2;
}
```

```
public Hund(){
    this.y = 1;
}
```

```
public Husky(){
    this.x = 1;
}
```

Konstruktoren - Beispiel



super()

```
public Tier(){
    this.y = 2;
}
```

super()

```
public Hund(){
    this.y = 1;
}
```

```
Husky h2 = new Husky()
```

```
println(h2.y)
```

1

```
public Husky(){
    this.x = 1;
}
```

Konstruktoren

- Subklasse instanziieren:
  - Erfordert die Instanziierung der Superklasse.
- Default-Konstruktor:
  - Ruft automatisch den Konstruktor der Superklasse auf.
- Konstruktoren und Vererbung:
  - Konstruktoren werden nicht vererbt.
- Sinn von Konstruktoren:
  - Initialisierung von Attributen bei der Objekterstellung.

Parametrisierte Konstruktoren

- Default-Konstruktor überschrieben:
  - Wenn der Default-Konstruktor der Superklasse durch einen parametrisierten Konstruktor ersetzt wird.
- super(...) erforderlich:
  - super(...) muss explizit aufgerufen werden, um den Konstruktor der Superklasse zu nutzen.
- Parameterreihenfolge beachten:
  - Die Reihenfolge und Anzahl der Parameter in super(...) muss identisch mit der des Superklassenkonstruktors sein.
- Fehler vermeiden:
  - Kein Aufruf von super(...) führt zu einem Kompilierungsfehler.

Beispiel

Wir sind gezwungen explizit super() aufzurufen - auch wenn es keinen Unterschied macht.


```java
public static class Super {
    int y;
    public Super() {
        this.y = 2;
    }
}

public static class Mid extends Super{
    int y;
    public Mid(int y) {
        this.y = 3;
    }
}

public static class Sub extends Mid{
    int x;
    public Sub() {
        super(3);
        this.x = 2;
    }
}
```

```java
public static class Super {
    int y;
    public Super() {
        this.y = 2;
    }
}

public static class Mid extends Super{
    int y;
    public Mid(int y) {
        this.y = 3;
    }
}

public static class Sub extends Mid{
    int x;
    public Sub() {
        super(3);
        this.x = 2;
    }
}
```

```java
public static class Super {
    int y;
    public Super() {
        this.y = 2;
    }
}

public static class Mid extends Super{
    int y;
    public Mid(int y) {
        this.y = 3;
    }
}

public static class Sub extends Mid{
    int x;
    public Sub(int x, int y) {
        super(y);
        this.x = x;
    }
}
```

Interfaces

Interfaces

- **Definition:**
  - Legen das Verhalten fest, das eine Klasse haben muss, um das Interface zu implementieren.
- **Implementierung der Methoden:**
  - Das Interface gibt nur die Methodensignaturen vor – die Implementierung erfolgt in der Klasse.
- **Keine Attribute:**
  - Interfaces enthalten keine Attribute, nur Konstanten.
- **Eigenschaften von Attributen:**
  - Alle Konstanten in einem Interface sind **public**, **static** und **final**.
  - Konstanten gehören zum Interface und sind unveränderlich.

```java
public interface Fahrzeug {
    void start();
    void stop();
    void checkSystem();
    void fahrmodusWechsel();
    Fahrmodus aktuellerFahrmodus();

    enum Fahrmodus{
       P,D,R,N;
    };
}
```

```java
public interface Verbrenner {
    int aktuellerGang();
    void wechsleGang(int gang);
}
```

```java
public interface Schluessel {
    void neuerSchluessel(String id);
    void verriegeln();
    void entriegeln();
    void fenster(boolean hoch);
}
```

```
public class Auto {
}
```

```java
public Auto implements Vehicle {
        // Vorgeschrieben durch Vehicle Implementierung
        public void start() {}
        public void stop() {}
        public void checkSystem() {}
        public void fahrmodusWechsel() {}
        public FahrModus aktuellerFahrmodus() {}
}
```

Wir können auch mehrere Interfaces implementieren

```java
public Auto implements Vehicle, Schluessel {
        // Vorgeschrieben durch Vehicle Implementierung
        public void start() {}
        public void stop() {}
        public void checkSystem() {}
        public void fahrmodusWechsel() {}
        public Fahrmodus aktuellerFahrmodus() {}

        // Vorgeschrieben durch Schluessel Implementierung
        public void neuerSchlüssel(String id) {}
        public void verriegeln() {}
        public void entriegeln() {}
        public void fenster(boolean hoch) {}
}
```

Wir können auch mehrere Interfaces implementieren

Interfaces: Intuition

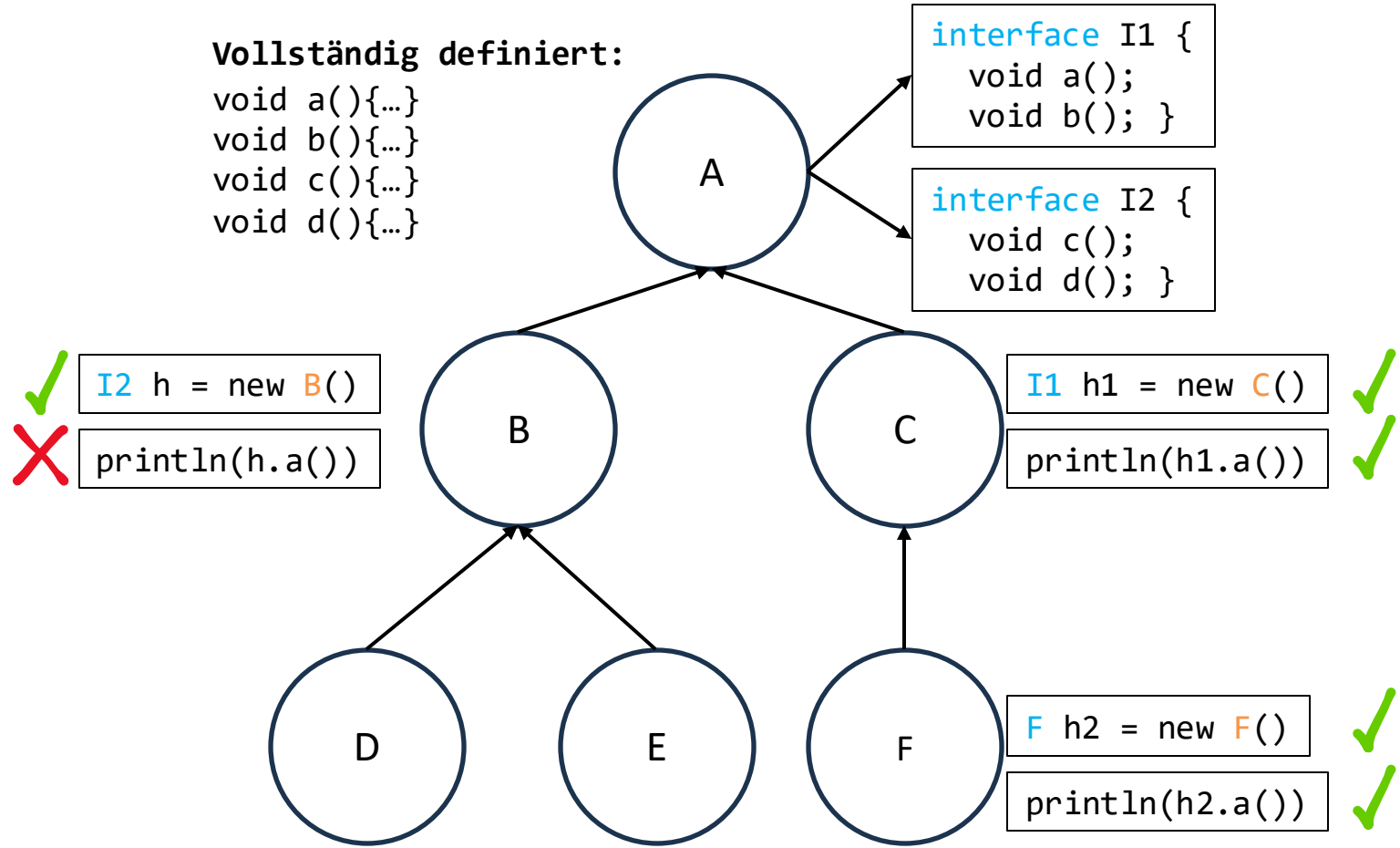
- **Definition:**
  - Ein Interface definiert einheitliche Regeln für Klassen.
- **Grundidee:**
  - Klassen müssen eine vorgegebene Grundstruktur erfüllen.
  - Die Implementierung der Details bleibt der Klasse überlassen.

Interfaces: Aus Sicht des Compilers

- **Pflicht zur Implementierung:**
  - Alle Methoden des Interfaces müssen in der implementierenden Klasse definiert werden.
- **"Vererbung" der Methodennamen:**
  - Nur die Signaturen der Methoden werden übernommen – die Implementierung erfolgt durch die Klasse.
- **Klare Regeln:**
  - Stellt sicher, dass alle Klassen mit dem Interface einheitliche Methoden bereitstellen.
- **Wichtig:**
  - Interfaces sind keine Klassen, sondern reine "Verträge".
  - Eine Klasse kann mehrere Interfaces gleichzeitig implementieren.

Vollständig definiert:

```
void a(){…}
void b(){…}
void c(){…}
void d(){…}
```



Dynamic Binding müsste eigentlich funktionieren, oder?

Der Compiler überprüft, ob die Methode im statischen Typ abrufbar ist:
a ist auf **I2** nicht definiert.

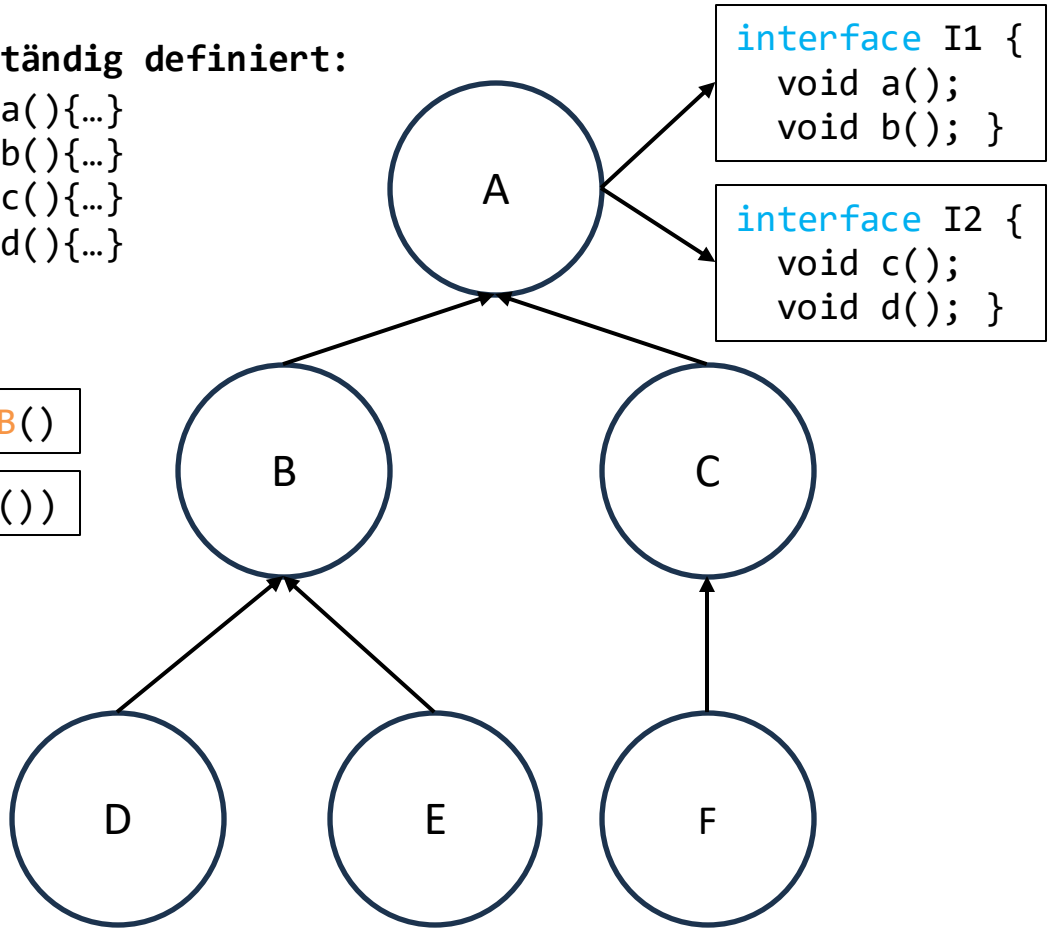
Dies führt zu einem **Compiler-Fehler**!

Vollständig definiert:

```
void a(){…}
void b(){…}
void c(){…}
void d(){…}
```

✓ `I2 h = new B()`

✗ `println(h.a())`



```
interface I1 {
    public void method1();
}

public class Base {
    int x = 100;

    public void method1() {
        System.out.println("B m1 x=" + x);
    }
}

public class T  extends Base implements I1 {
    int x = 200;

    public void method0() {
        System.out.println("T m0 x=" + x);
    }
}

public class Q implements I1 {
    int x = 300;
    void method1() {
        System.out.println("Q m1 x=" + x);
    }
}
```

```
public class R implements I1 {
    int x = 400;

    public  void method1() {
        System.out.println("R m1 x=" + x);
    }
    public void method1(int i) {
        System.out.println("R m1 i=" + i);
    }
}


public class S extends T {

    public void method1() {
        System.out.println("S m1 x=" + x);
    }

    public void method1(int i) {
        System.out.println("S m1 i=" + i);
    }
}

public class X extends Base {
    int x = 600;

    public void method1() {
        System.out.println("X m1 x=" + x);
    }
}
```

```
Base b = new Base();
b.method1();
```

B m1 x=100

```
Base b = new T();
b.method1();
```

B m1 x=100

```
I1 q = new Q();
q.method1();
```

Compile-Fehler

```
I1 t = new T();
t.method1(1);
```

Compile-Fehler

```
R  r = new R();
r.method1(2);
```

R m1 i=2

```
R  r = new R();
r.method1();
```

R m1 x=400

```
S s = new S();
s.method1(3);
```

S m1 i=3

```
I1 s = new S();
s.method1();
```

S m1 x=200

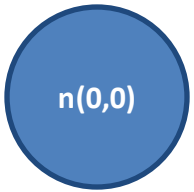
```
I1 x = new X();
x.method1();
```

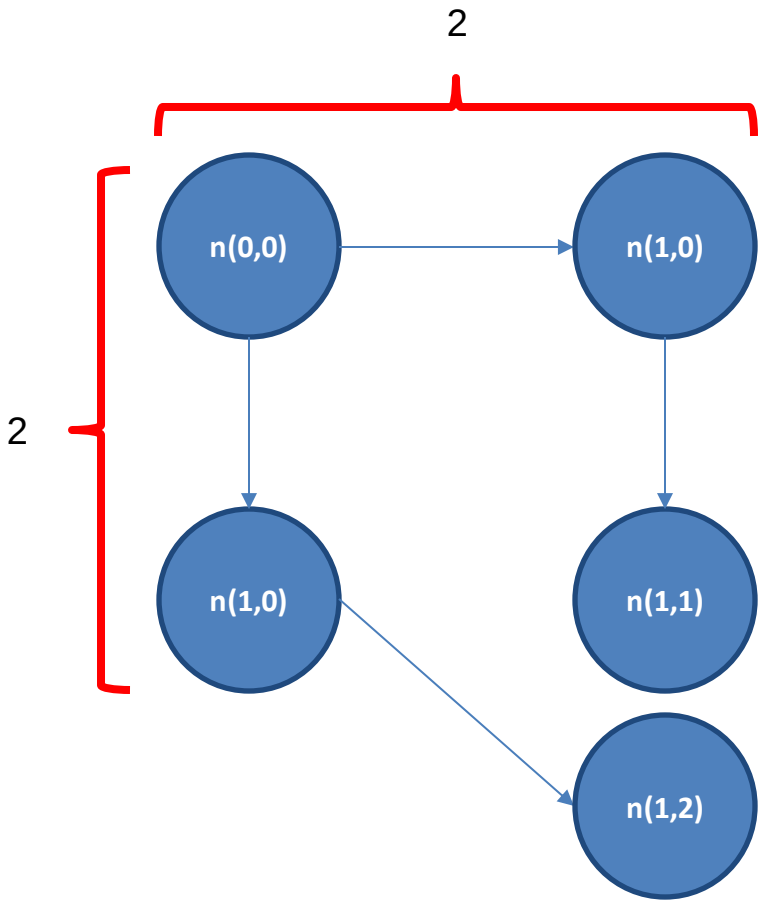
Compile-Fehler

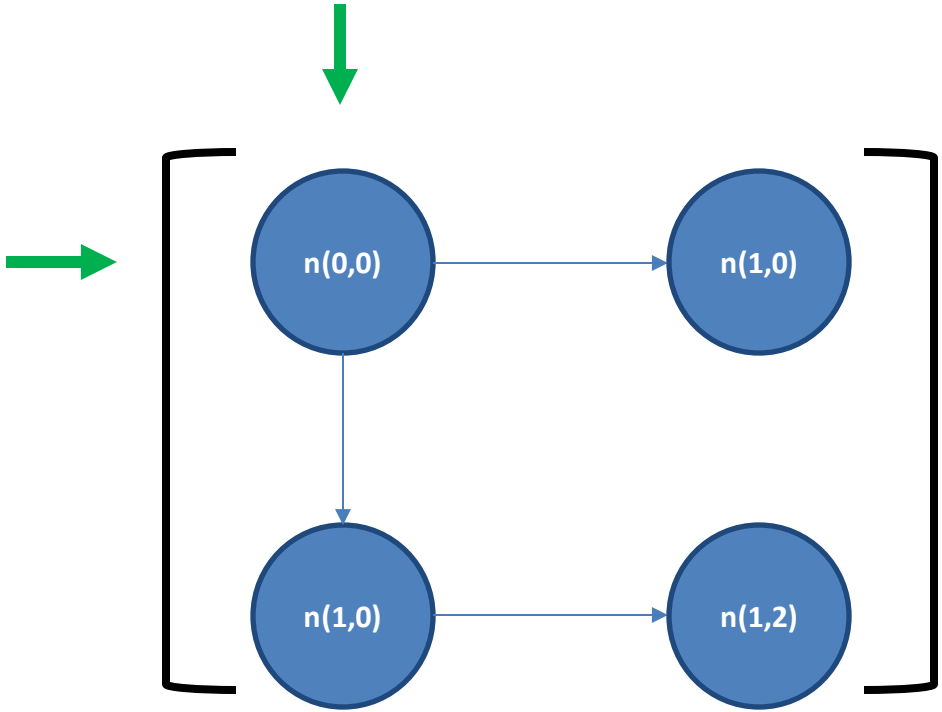
Graphenaufgaben

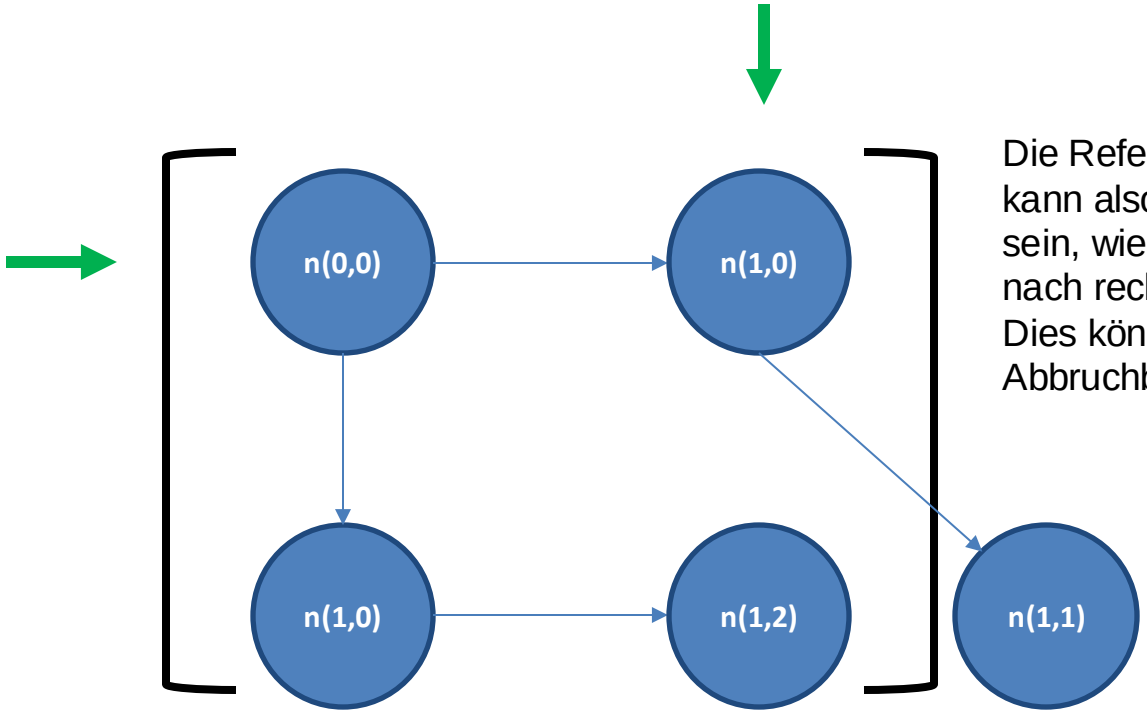
Trick zum lösen *mancher* Graphenaufgaben

- **Rekursion**
  - Gegebene Daten (oft einzelner Knoten) rekursiv abarbeiten und wichtige Merkmale speichern. **Wieso rekursiv?**
- **Daten in richtige Form bringen**
  - Graph in einer Matrix o.Ä. speichern
- **Bedingungen iterativ überprüfen**
  - Die zu überprüfenden Bedingungen stehen in der Aufgabenbeschreibung
  - Iterativ kann ist das jetzt einfacher zu lösen durch konsequentes Überprüfen der Bedingungen

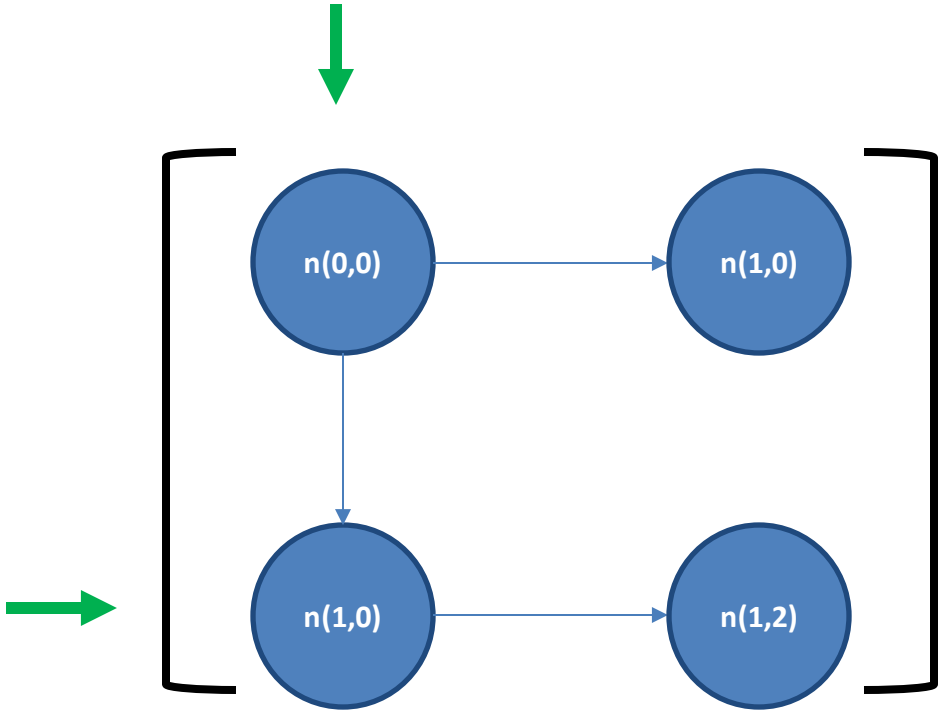

n(0,0)

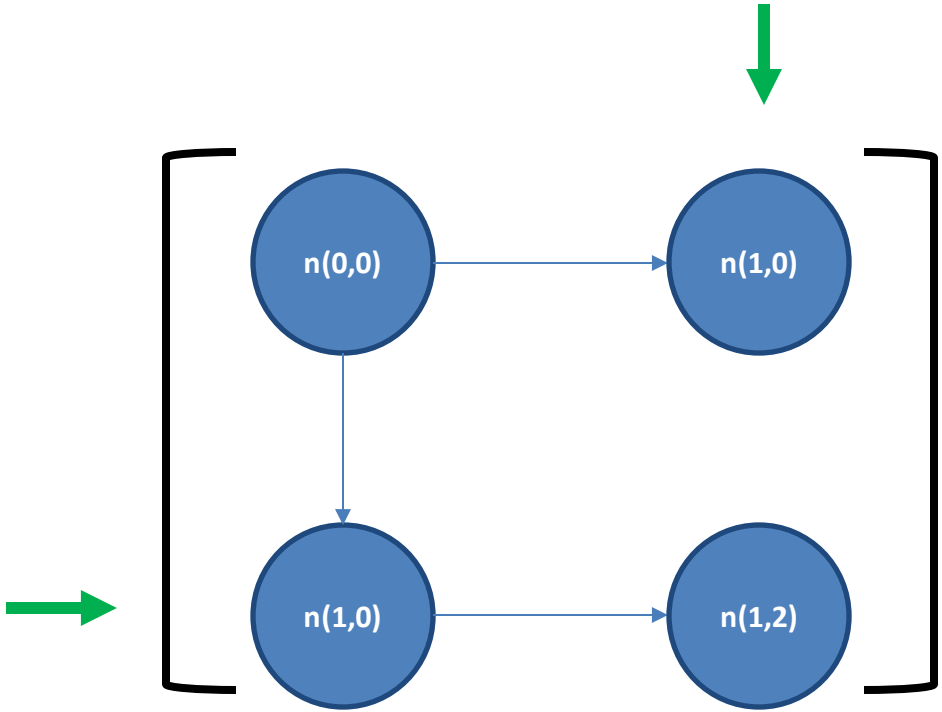

2
2
n(0,0)
n(1,0)
n(1,0)
n(1,1)
n(1,2)


n(0,0)
n(1,0)
n(1,0)
n(1,2)



Die Referenz nach unten kann also nicht dieselbe sein, wie vom Knoten (1,0) nach rechts!
Dies könnte eine Abbruchbedingung sein.


n(0,0)
n(1,0)
n(1,0)
n(1,2)


n(0,0)
n(1,0)
n(1,0)
n(1,2)

Pyramide (U10)

Umfrage 2

umfrage.henrikpaetzold.de

- **Folgendes kommt noch:**
  - Comparator
  - Maps
  - Sets
  - Lists
  - Generics