

252-0027

Einführung in die Programmierung
Übungen

Woche 11: Vererbung II

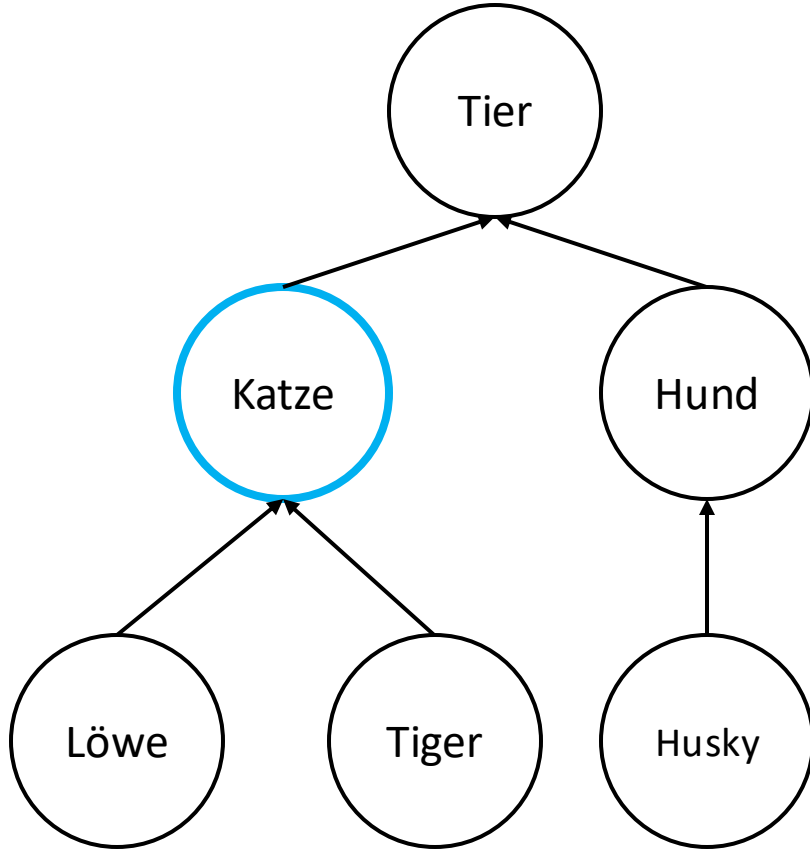
Henrik Pätzold
Departement Informatik
ETH Zürich

Heutiger Plan

- Timed Bonus Hinweis
- Theorie Simulation nächste Woche
- Vererbung - Recap
- Konstruktoren
- Interfaces
- Theorieaufgaben
- Kahoot

Recap

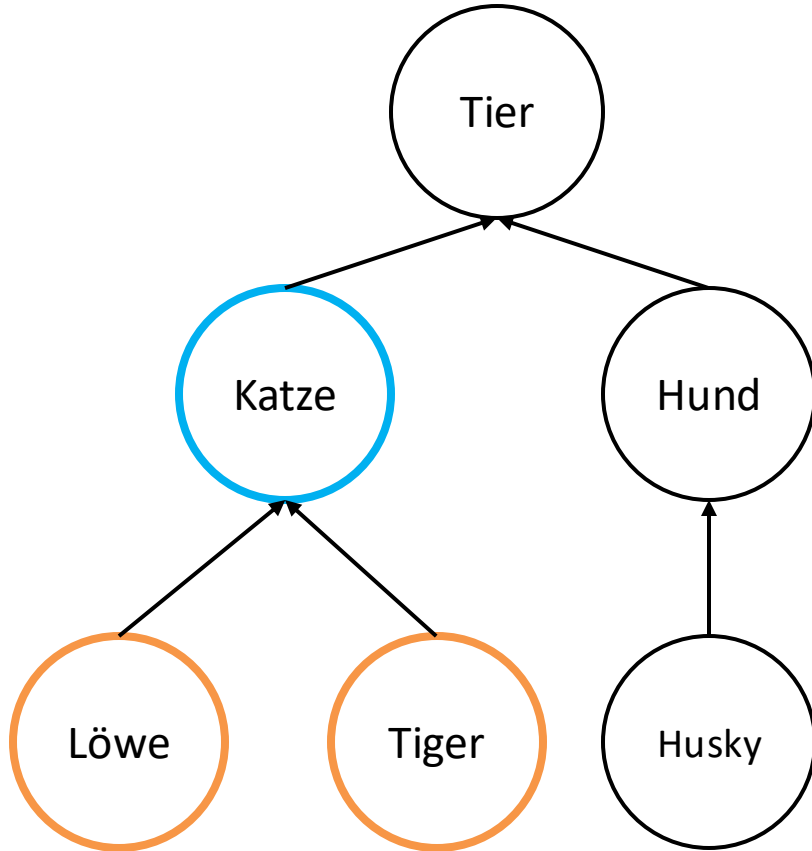
Verdeutlicht – Was für Typen akzeptiert eine Variable?



Statischer Typ

Möglicher Dynamischer Typ

Verdeutlicht – Was für Typen akzeptiert eine Variable?



Funktionieren diese Zuweisungen?

`Katze k1 = new Katze()`



`Tier t1 = new Tiger()`



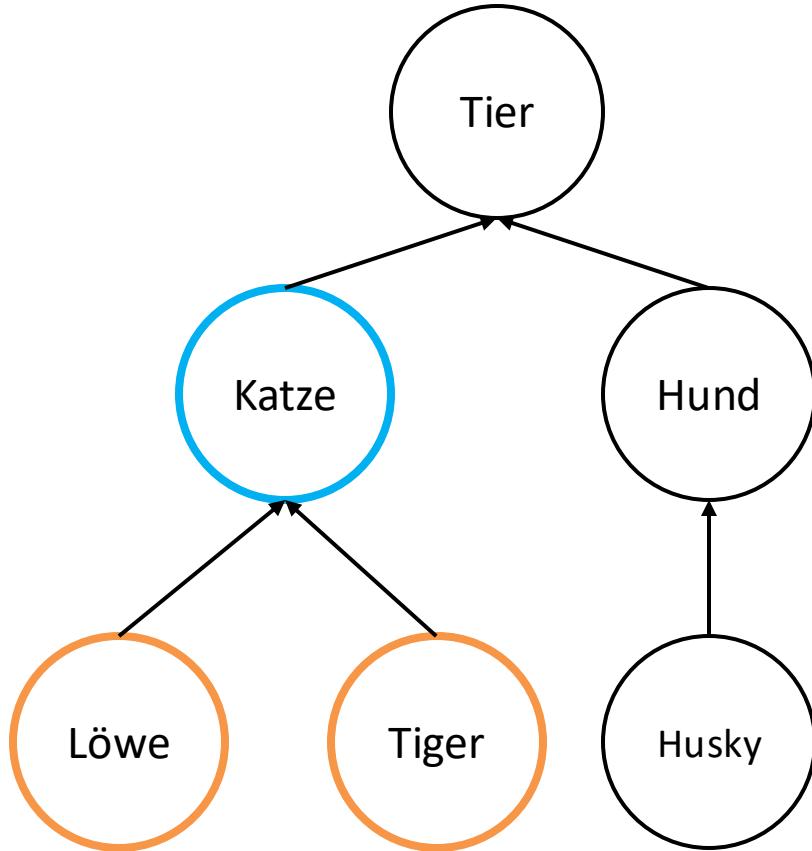
`Katze k3 = t1`



Statischer Typ

Möglicher Dynamischer Typ

Verdeutlicht – Was für Typen akzeptiert eine Variable?



Warum ist das problematisch?

```
Tier t1 = new Tiger()
```



```
Katze k3 = t1
```



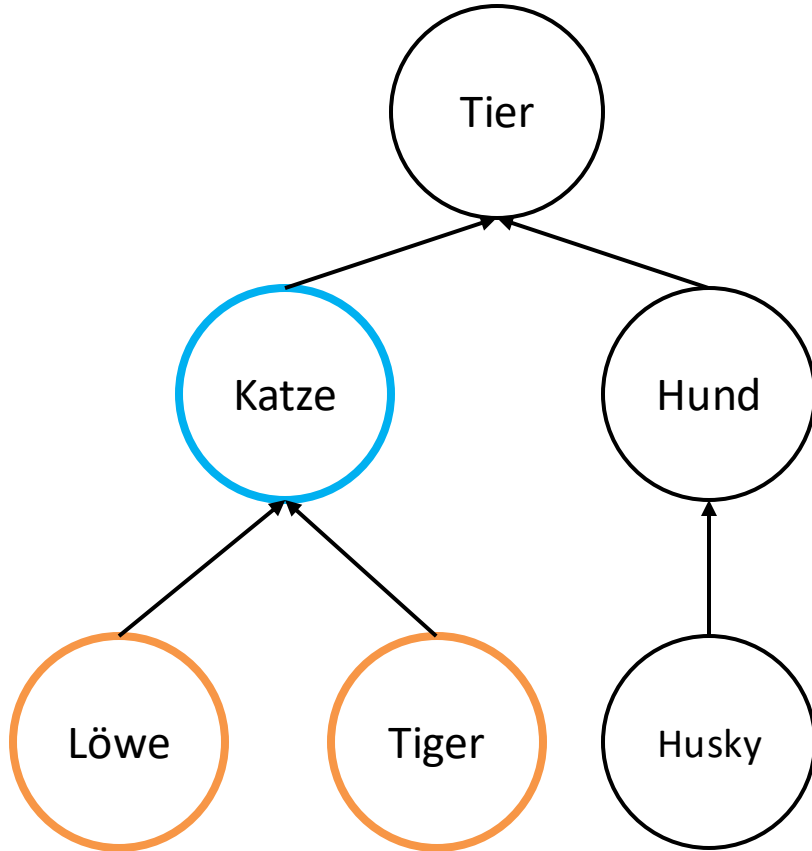
Was sieht der Compiler? 

```
Katze = Tier
```

Statischer Typ

Möglicher Dynamischer Typ

Verdeutlicht – Was für Typen akzeptiert eine Variable?



Was ist aber mit:

```
Tier t1 = new Tiger()
```

```
Katze k3 = (Tiger) t1
```

Was sieht der Compiler? 

```
Katze = Tiger
```



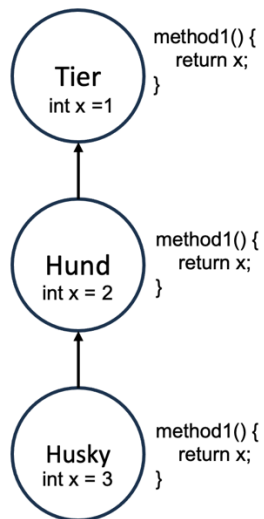
Statischer Typ

Möglicher Dynamischer Typ

Methoden und Attribute

Attribute

- Werden durch statischen Typen gewählt
- Werden sie initialisiert, wird die der superklasse *überdeckt*



Methoden

- Werden durch dynamischen Typen gewählt
- Ist sie nicht definiert wird sie weiter oben in der Vererbungshierarchie gesucht

```
Husky t = new Husky();
System.out.println(t.x);
System.out.println(((Hund)t).x);

Tier t = new Husky();
System.out.println(t.method1());
```

3
2
1

super und instanceof

Keywords bei Objekten

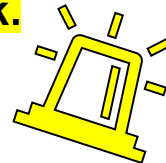
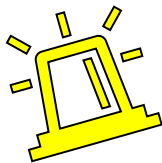
- **new** MyClass(...)
 - Ruft einen Konstruktor von MyClass mit der entsprechenden Argumentenliste auf
- **super**(...)
 - Ruft einen Konstruktor der Superklasse mit der entsprechenden Argumentenliste auf

super-Keyword

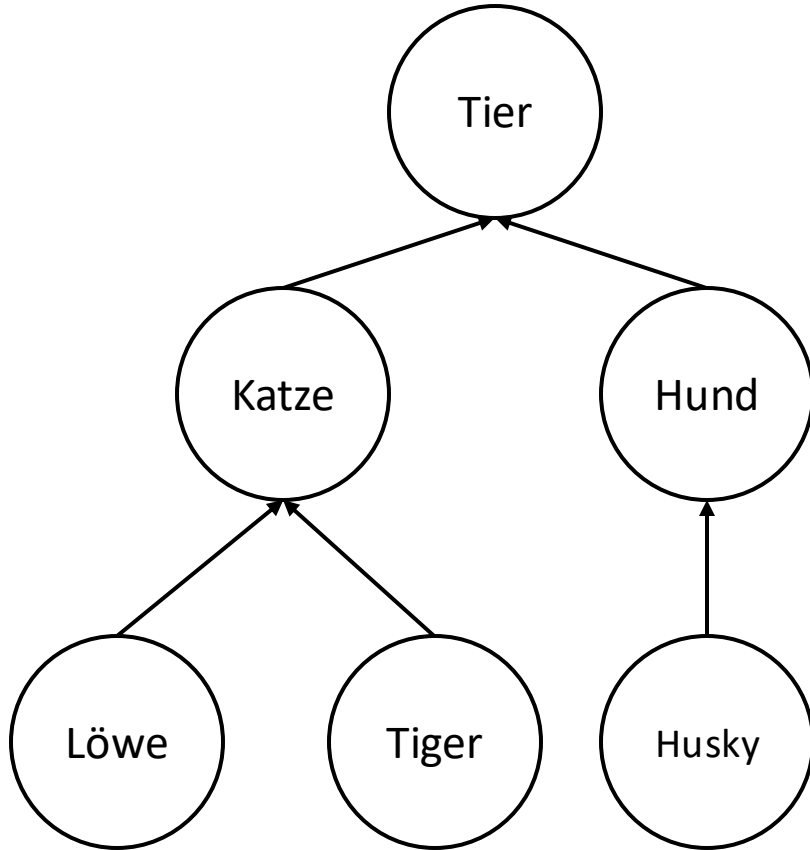
- `super(...)` ruft den Konstruktor der Superklasse auf
 - sie muss dieselbe Methodensignatur wie der Superklassen-Konstruktor haben
- `super` allein kann mittels dot-notation auf Attribute und Methoden der superklasse zugreifen
 - `super.var1 = 1`
 - `super.method()`

konkretesObjekt instanceof Klasse

- prüft, ob der **dynamische** Typ von **konkretesObjekt** eine Unterklasse von **Klasse** ist oder **Klasse** selbst und gibt **true** zurück, falls ja
 - Wenn **konkretesObjekt** **null** ist, gibt **instanceof** immer **false** zurück.
- wenn statischer Typ des Objekts und zu prüfende Klasse **keine gemeinsame Vererbungshierarchie** haben, erkennt Compiler, dass Prüfung sinnlos ist, und **gibt einen Compile-Fehler zurück.**



instanceof



true oder false?

```
Tier hd = new Hund()
```

```
hd instanceof Husky
```

```
hd instanceof Tier
```

```
Hund h2 = new Hund()
```

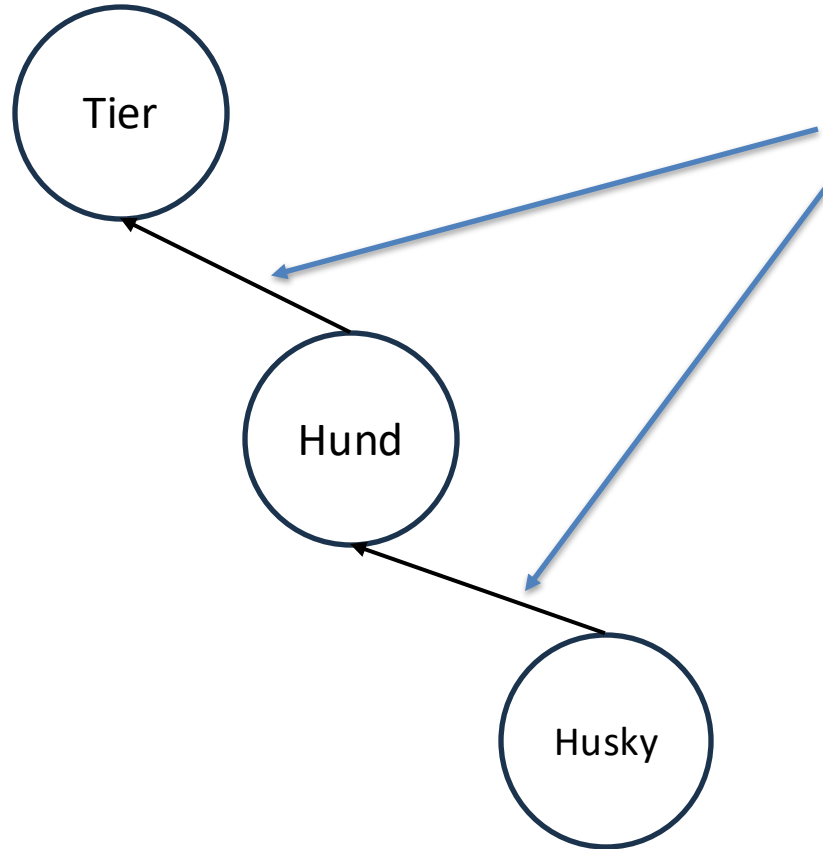
```
h2 instanceof Katze
```

Statischer Typ
Dynamischer Typ



Konstrukturen

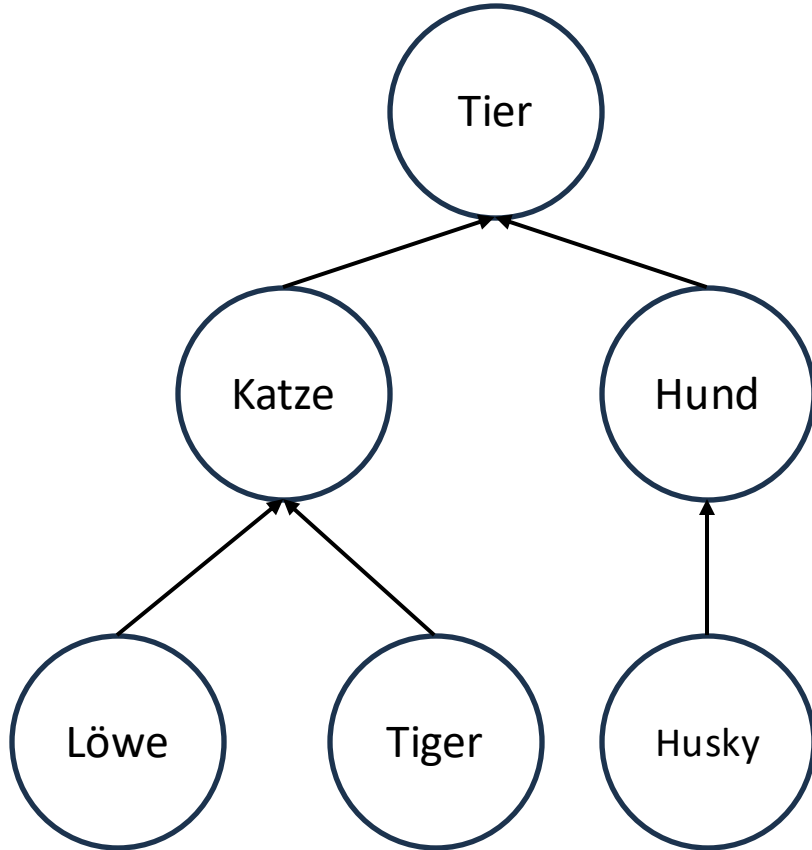
Konstrukturen



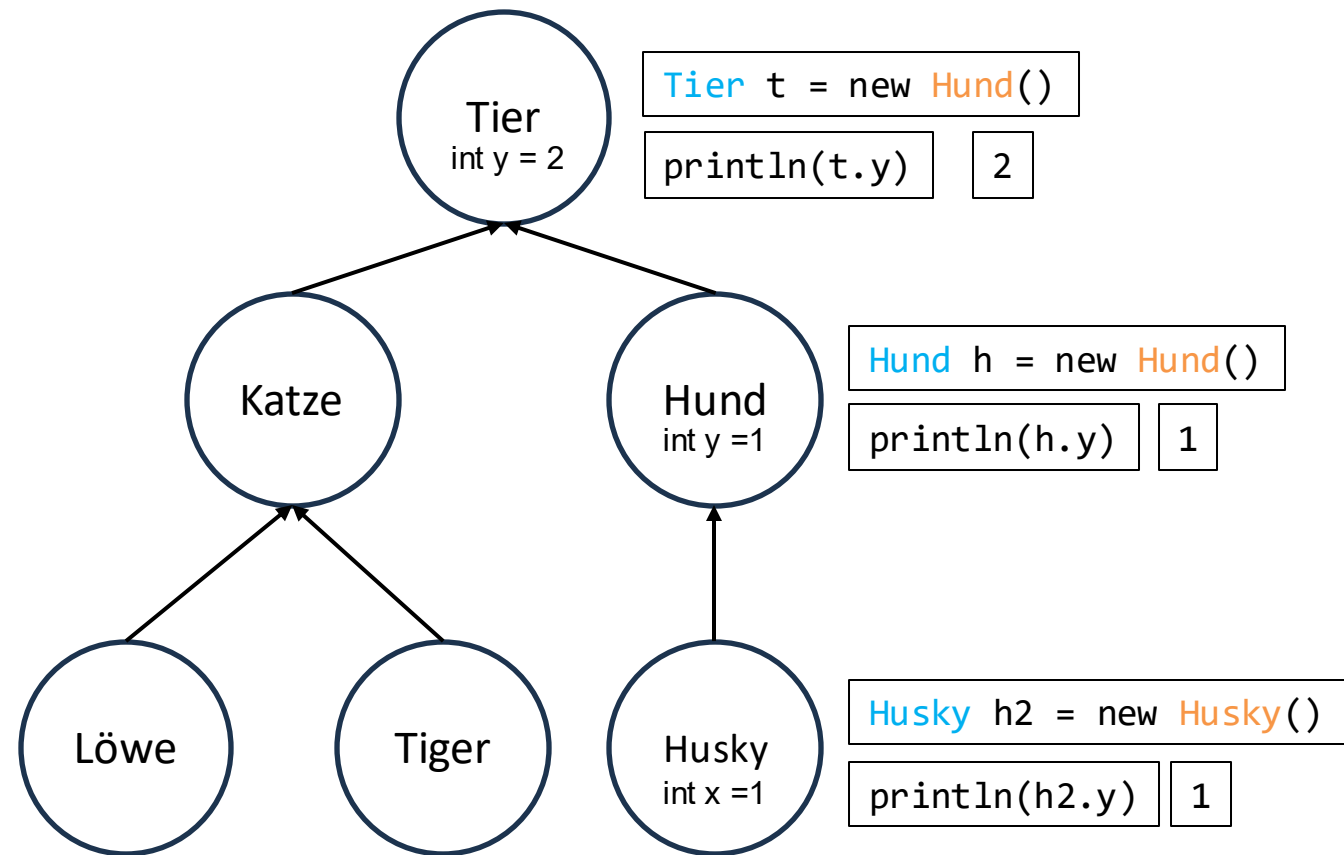
Die Pfeile sind eine “erbt von” - Beziehung

- 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

Konstrukturen - Beispiel



Konstrukturen - Beispiel



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

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

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

Konstrukturen - Beispiel

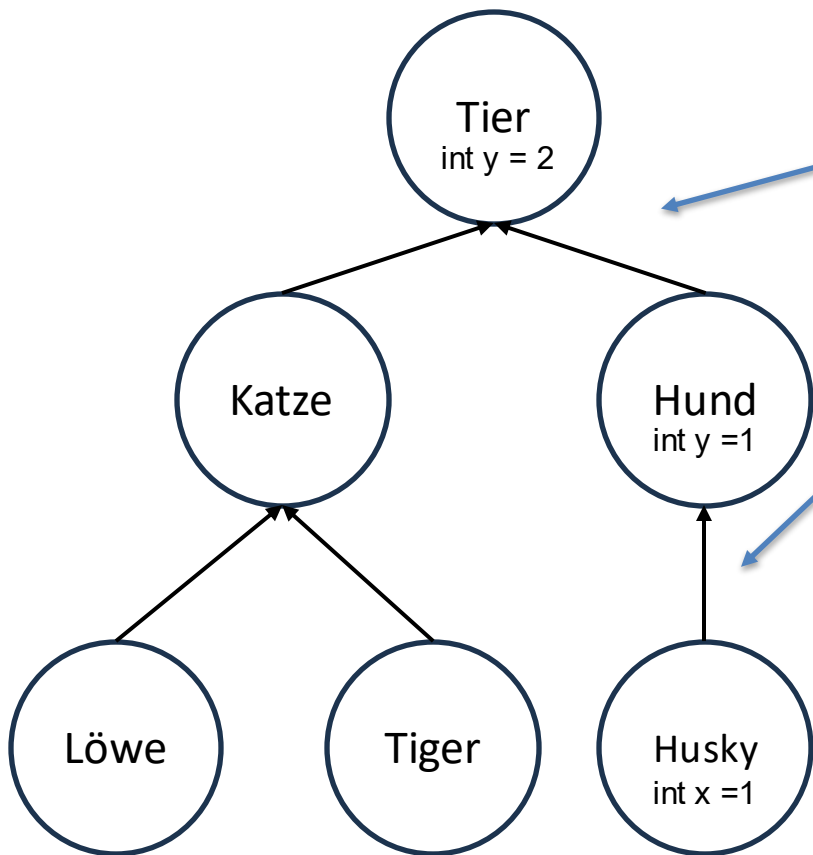
Die Pfeile sind eine “erbt von” - Beziehung

- 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;  
}
```



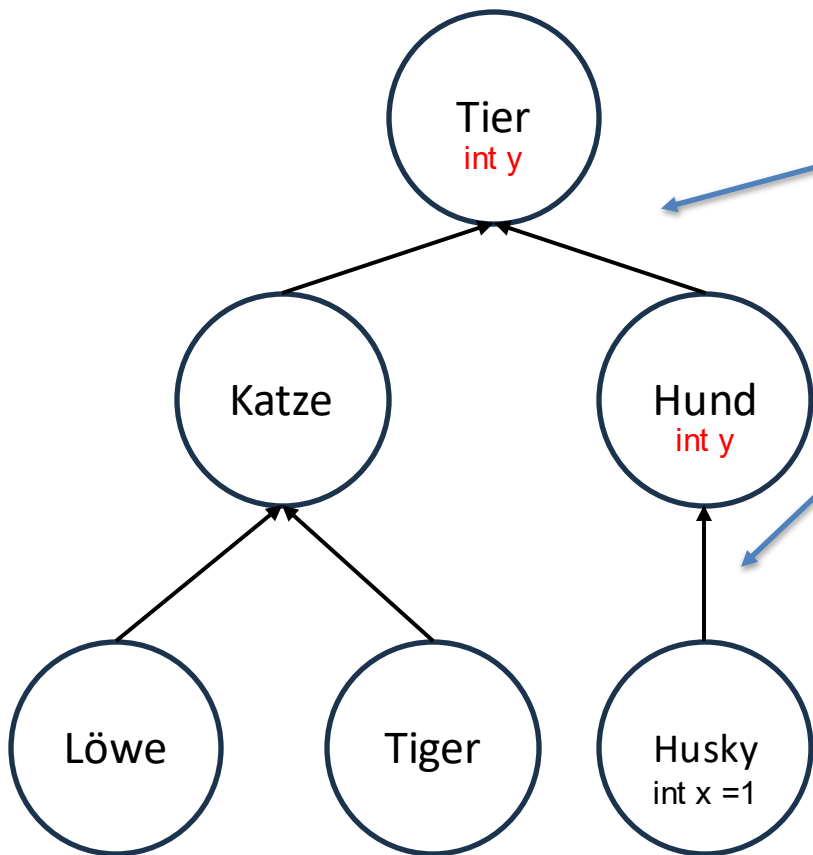
Konstrukturen - Beispiel

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, Standardwerte. (null oder typ-spez.)

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

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



Konstrukturen - Beispiel

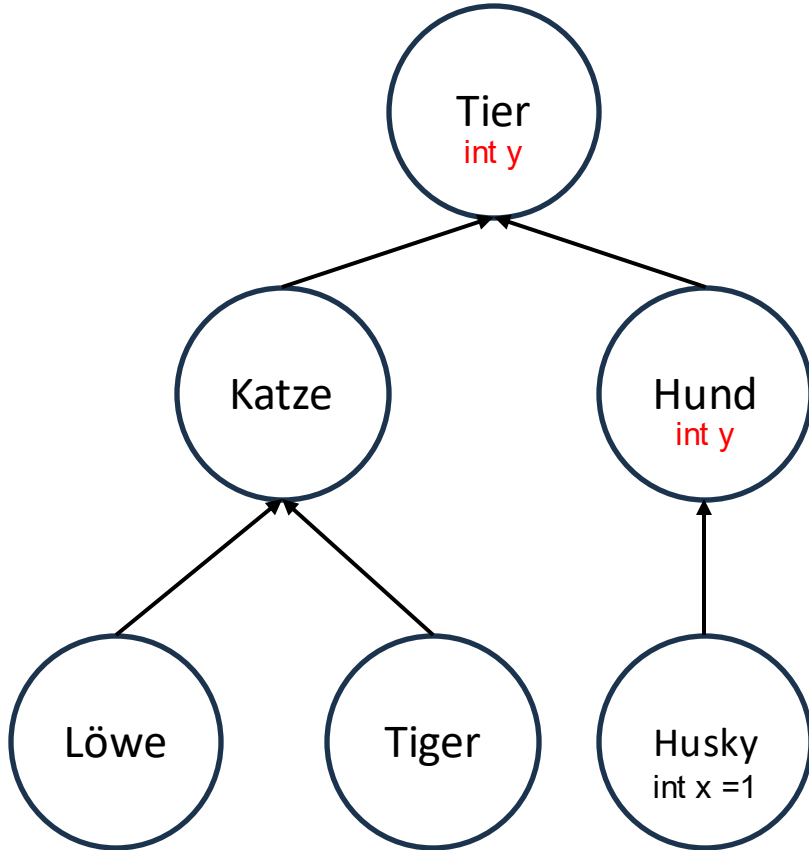
Die Pfeile sind eine “erbt von” - Beziehung

- 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()
```

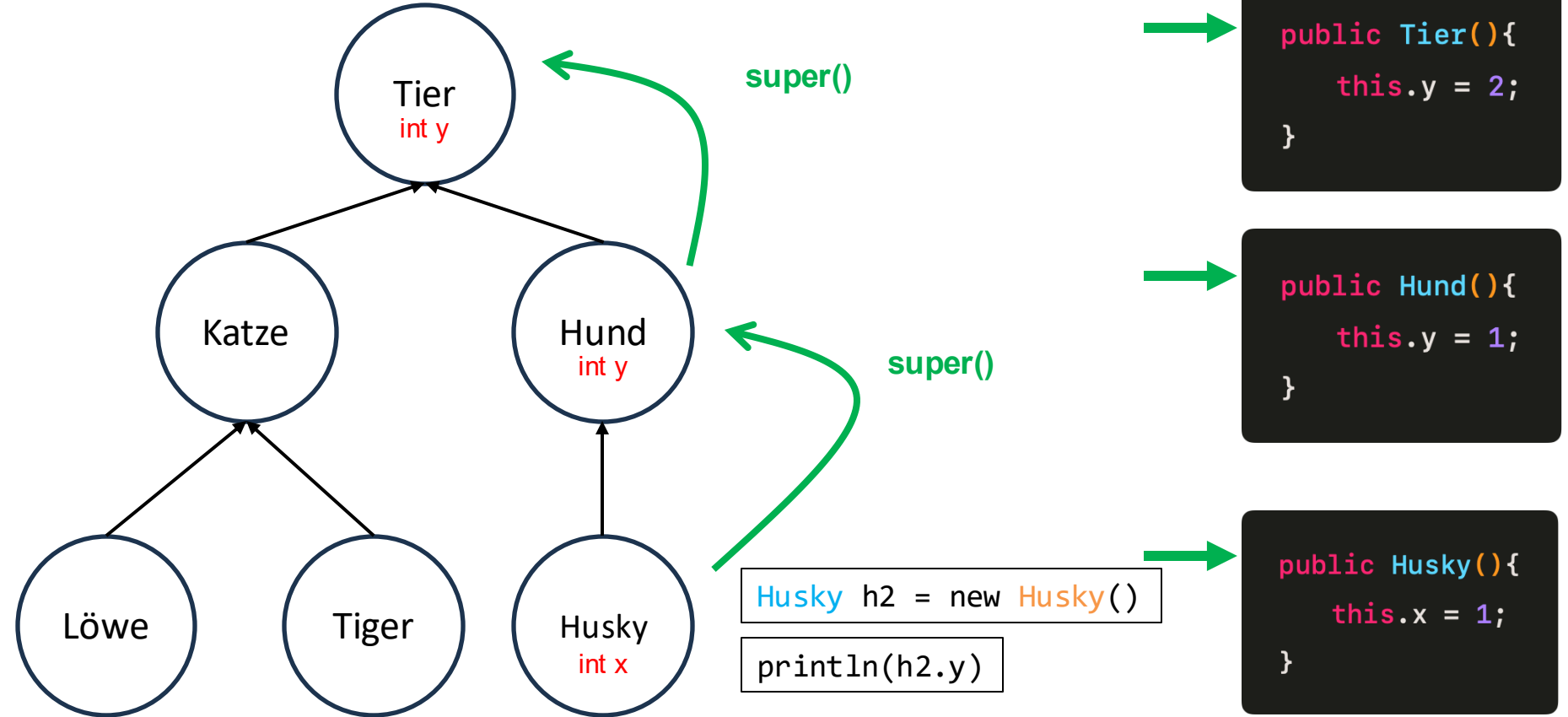
```
println(h2.y)
```



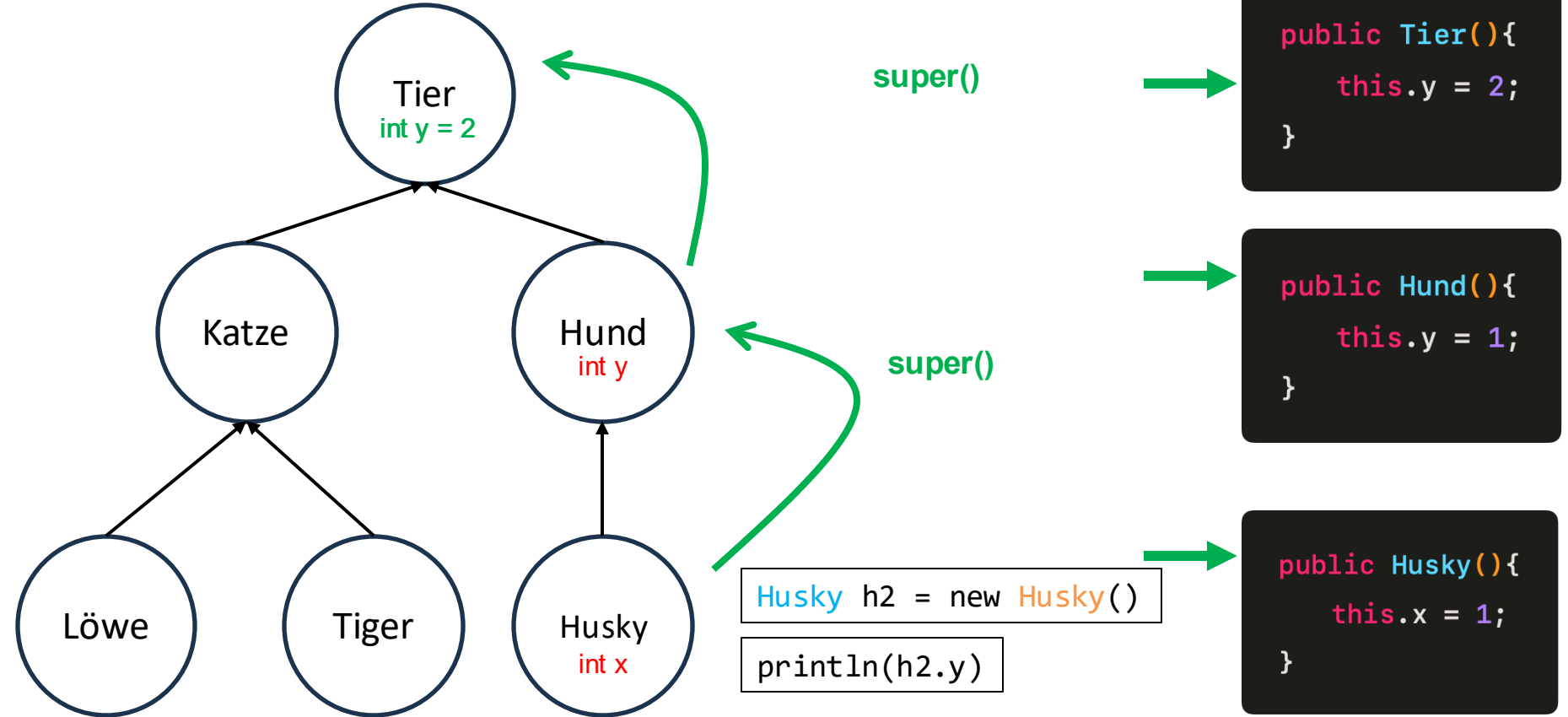
Konstrukturen

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

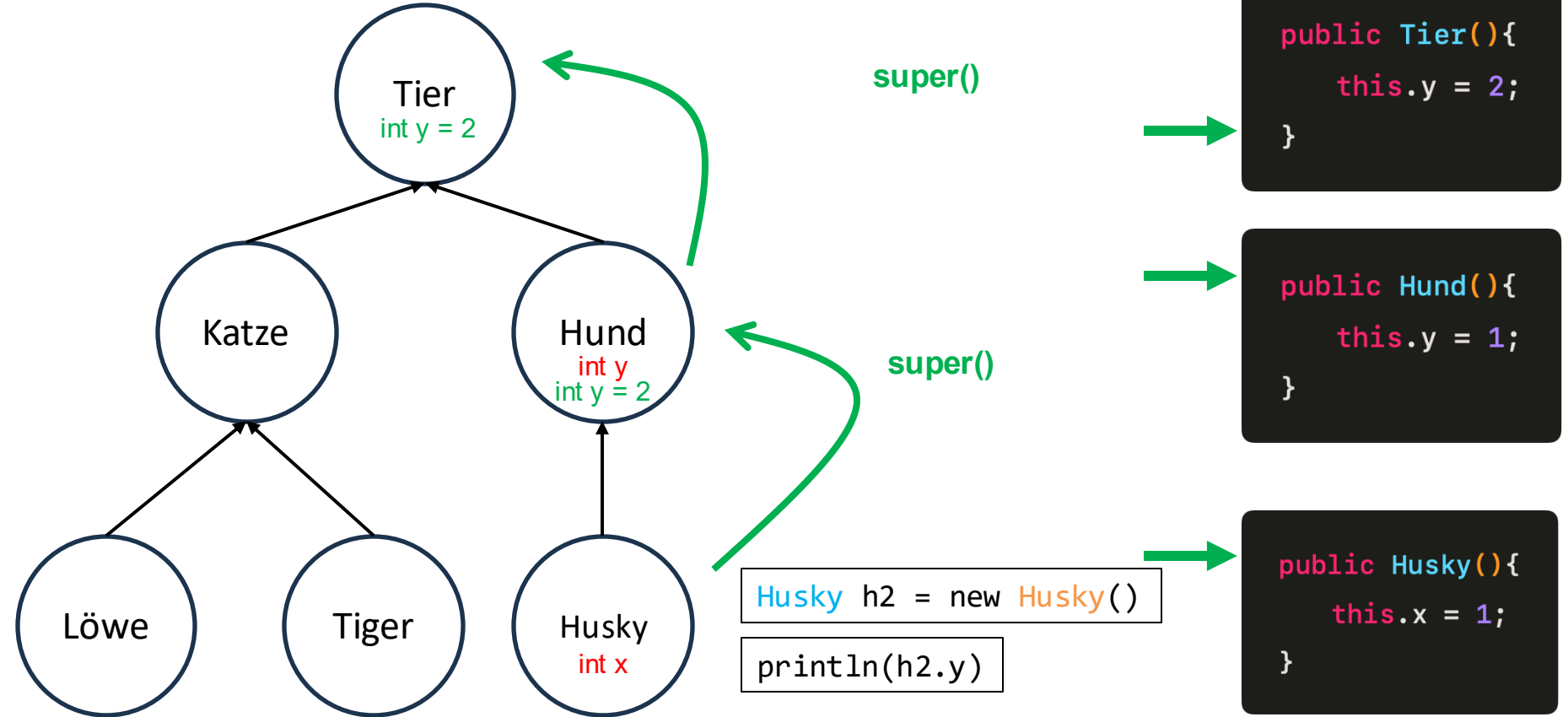
Konstrukturen - Beispiel



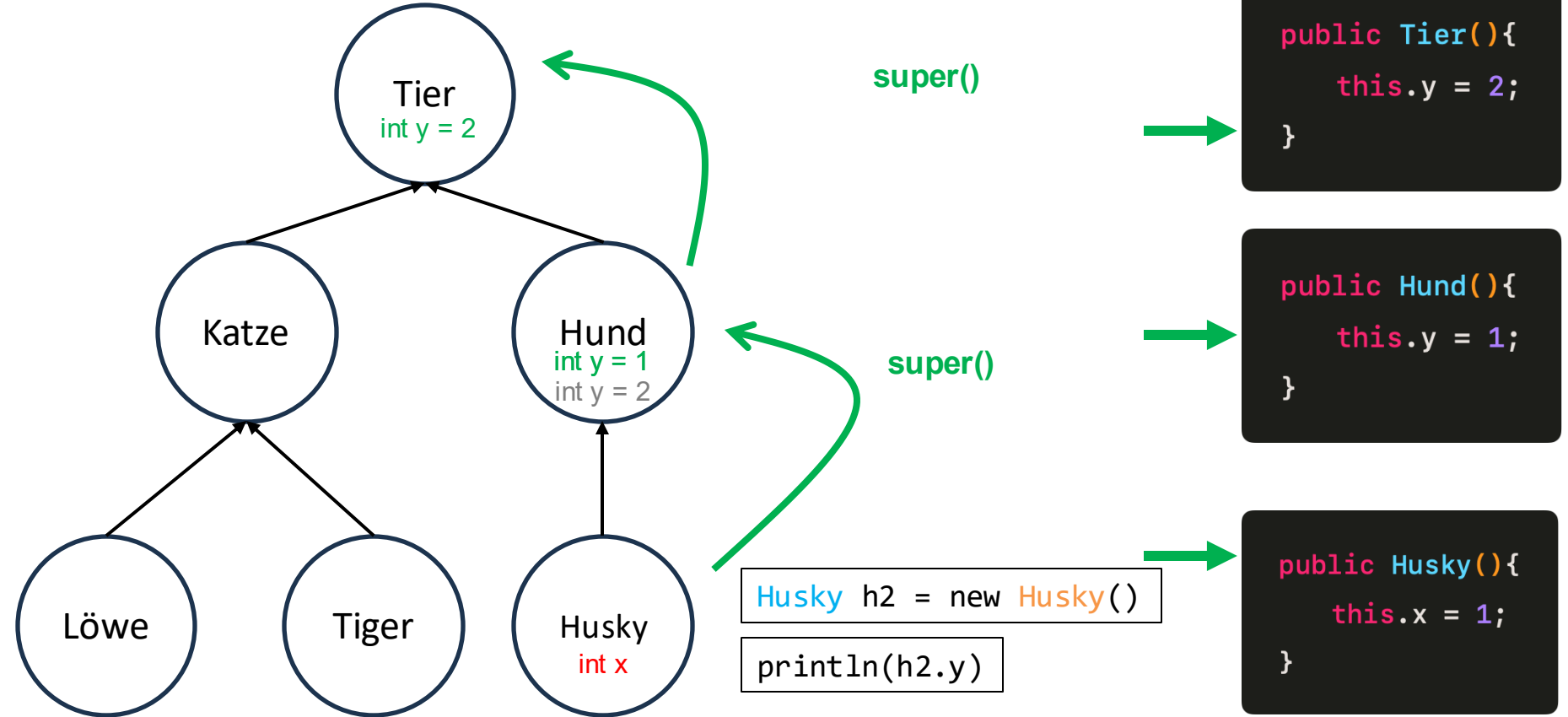
Konstrukturen - Beispiel



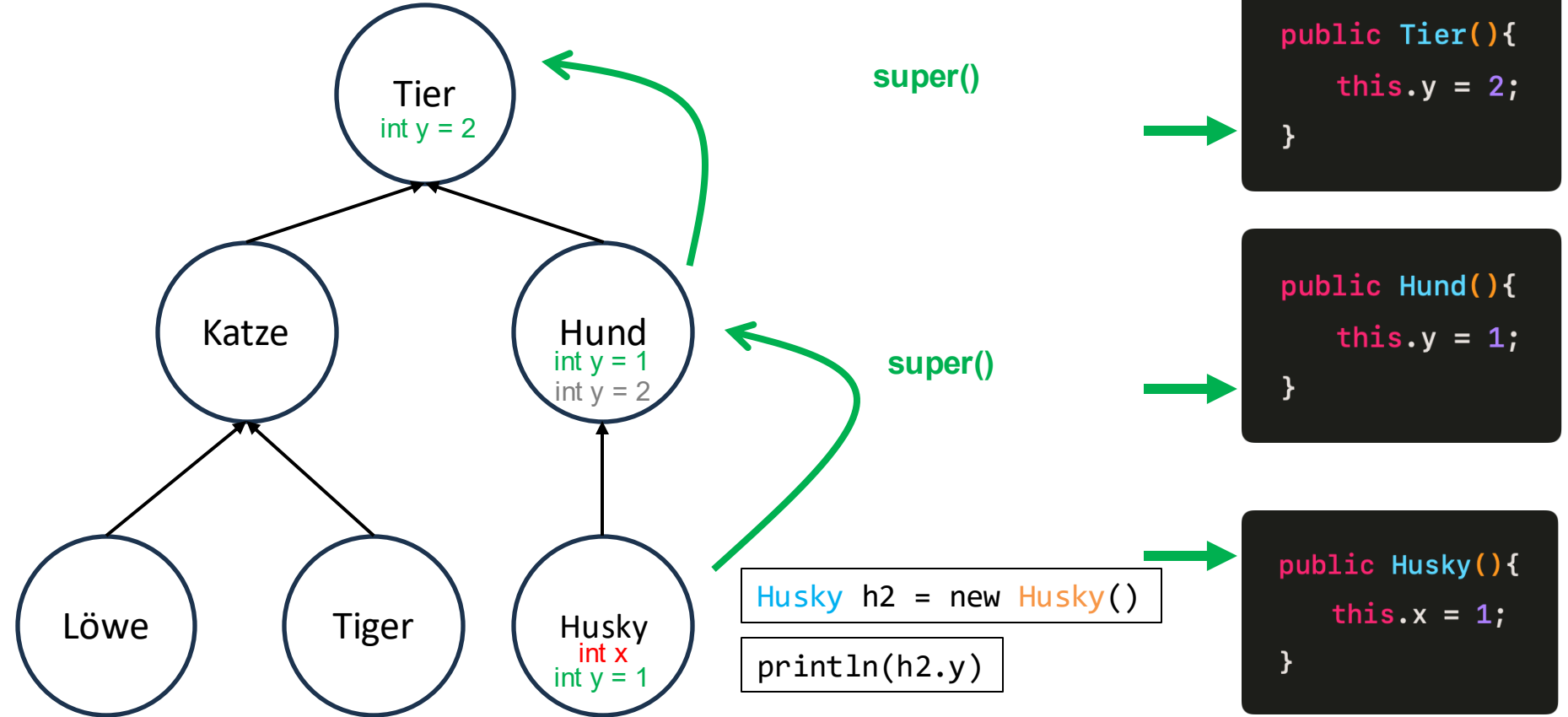
Konstrukturen - Beispiel



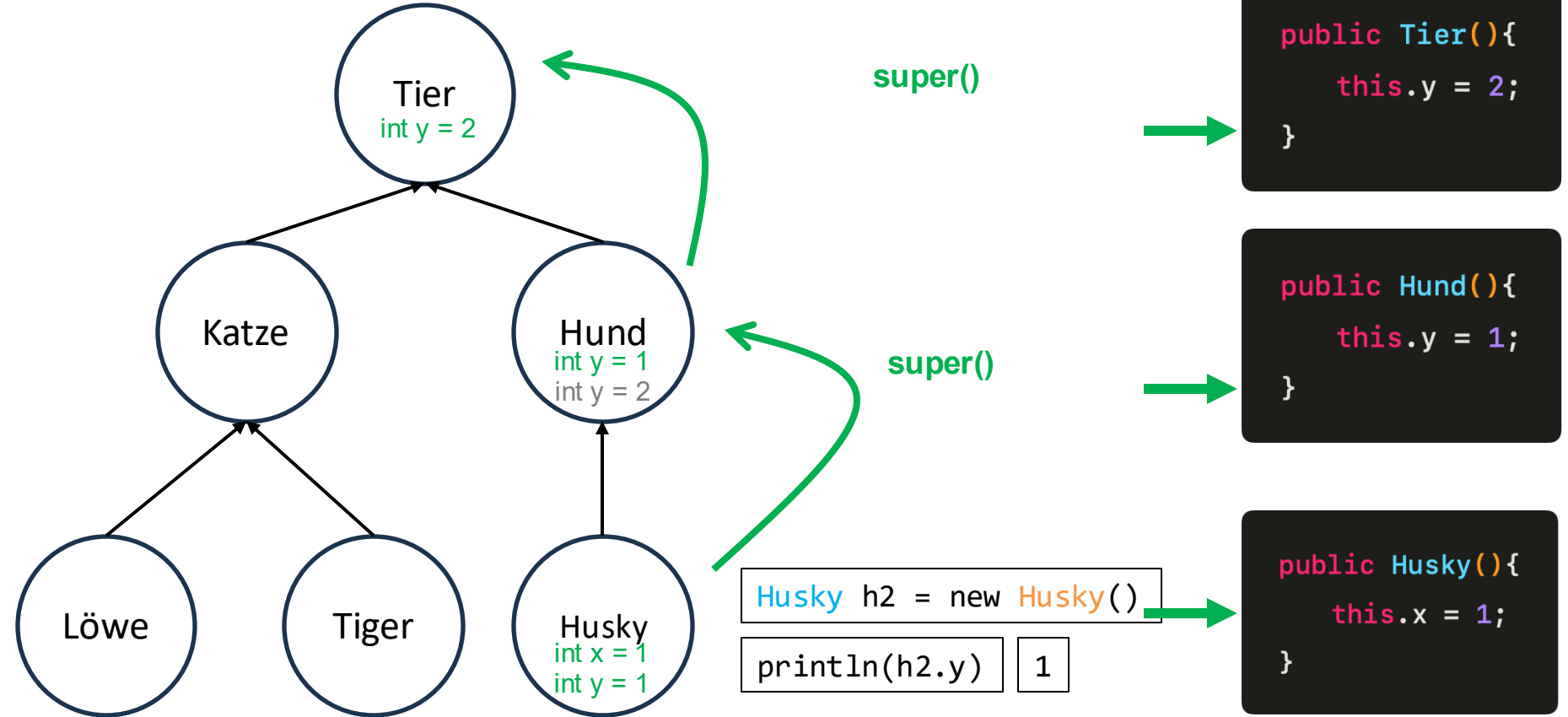
Konstrukturen - Beispiel



Konstrukturen - Beispiel



Konstrukturen - Beispiel



Konstrukturen

- Subklasse instanziiieren:
 - Erfordert die Instanziierung der Superklasse.
- Default-Konstruktor:
 - Ruft automatisch den Konstruktor der Superklasse auf.
- Konstrukturen und Vererbung:
 - Konstrukturen werden nicht vererbt.
- Sinn von Konstrukturen:
 - 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.



```
1 public static class Super {
2     int y;
3     public Super() {
4         this.y = 2;
5     }
6 }
7
8 public static class Mid extends Super{
9     int y;
10    public Mid(int y) {
11        this.y = 3;
12    }
13 }
14
15 public static class Sub extends Mid{
16     int x;
17     public Sub() {
18         super(3);
19         this.x = 2;
20     }
21 }
```

```
1 public static class Super {
2     int y;
3     public Super() {
4         this.y = 2;
5     }
6 }
7
8 public static class Mid extends Super{
9     int y;
10    public Mid(int y) {
11        this.y = 3;
12    }
13 }
14
15 public static class Sub extends Mid{
16     int x;
17    public Sub() {
18        super(3);
19        this.x = 2;
20    }
21 }
```

```
1 public static class Super {
2     int y;
3     public Super() {
4         this.y = 2;
5     }
6 }
7
8 public static class Mid extends Super{
9     int y;
10    public Mid(int y) {
11        this.y = 3;
12    }
13 }
14
15 public static class Sub extends Mid{
16     int x;
17    public Sub(int x, int y) {
18        super(y);
19        this.x = x;
20    }
21 }
```

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.



```
1 public interface Fahrzeug {
2     void start();
3     void stop();
4     void checkSystem();
5     void fahrmodusWechsel();
6     Fahrmodus aktuellerFahrmodus();
7
8     enum Fahrmodus{
9         P,D,R,N;
10    };
11 }
```



```
1 public interface Verbrenner {
2     int aktuellerGang();
3     void wechsleGang(int gang);
4 }
```



```
1 public interface Schluessel {
2     void neuerSchluessel(String id);
3     void verriegeln();
4     void entriegeln();
5     void fenster(boolean hoch);
6 }
```

```
public class Auto {  
}
```

```
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

```
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

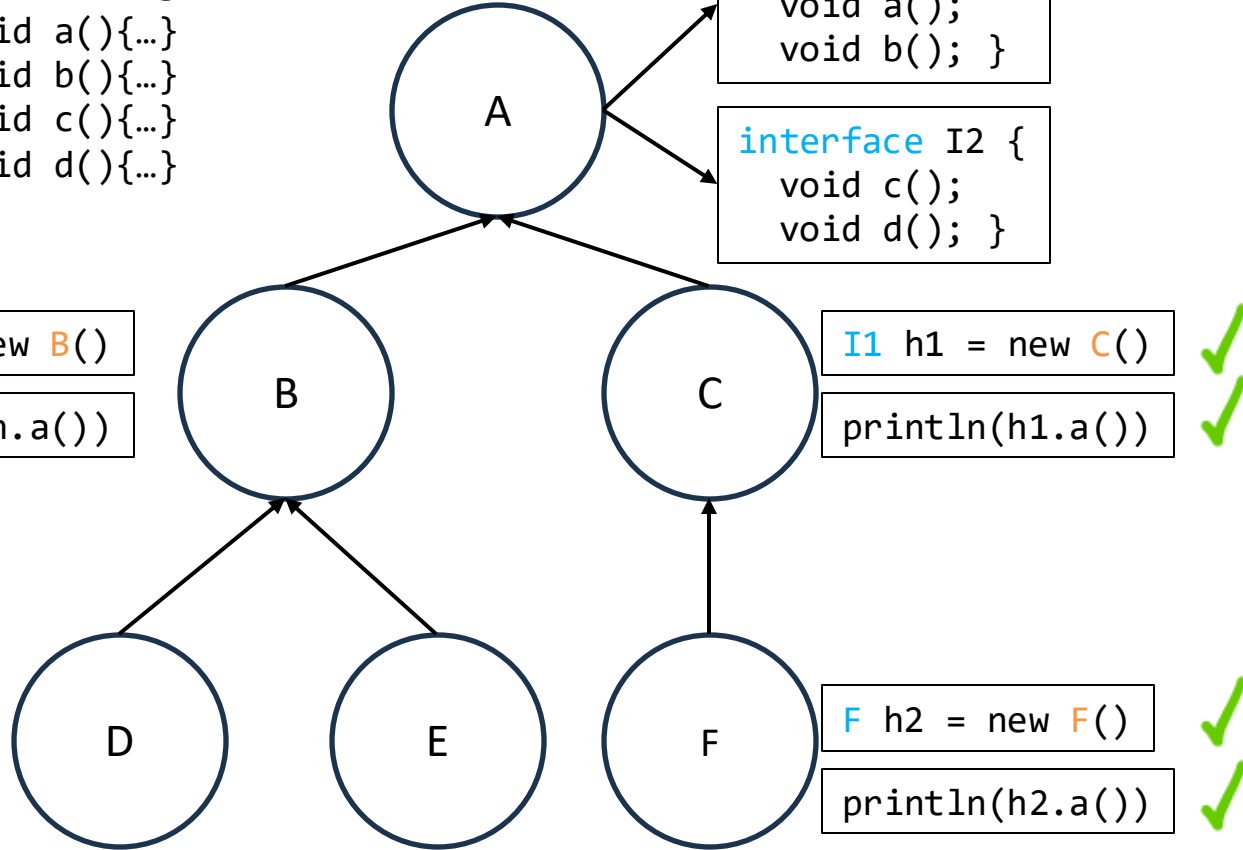
- **Pflicht zur Implementierung:**
 - Alle Methoden des Interfaces müssen in der implementierenden Klasse definiert werden.
- **"Vererbung" der Methodennamen:**
 - Nur die Signaturen der Methoden werden übernommen – die Implementierung erfolgt durch die Klasse.
- **Klare Regeln:**
 - Stellt sicher, dass 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(){...}
```

```
interface I1 {  
    void a();  
    void b(); }  
interface I2 {  
    void c();  
    void d(); }
```

```
interface I2 {  
    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!**

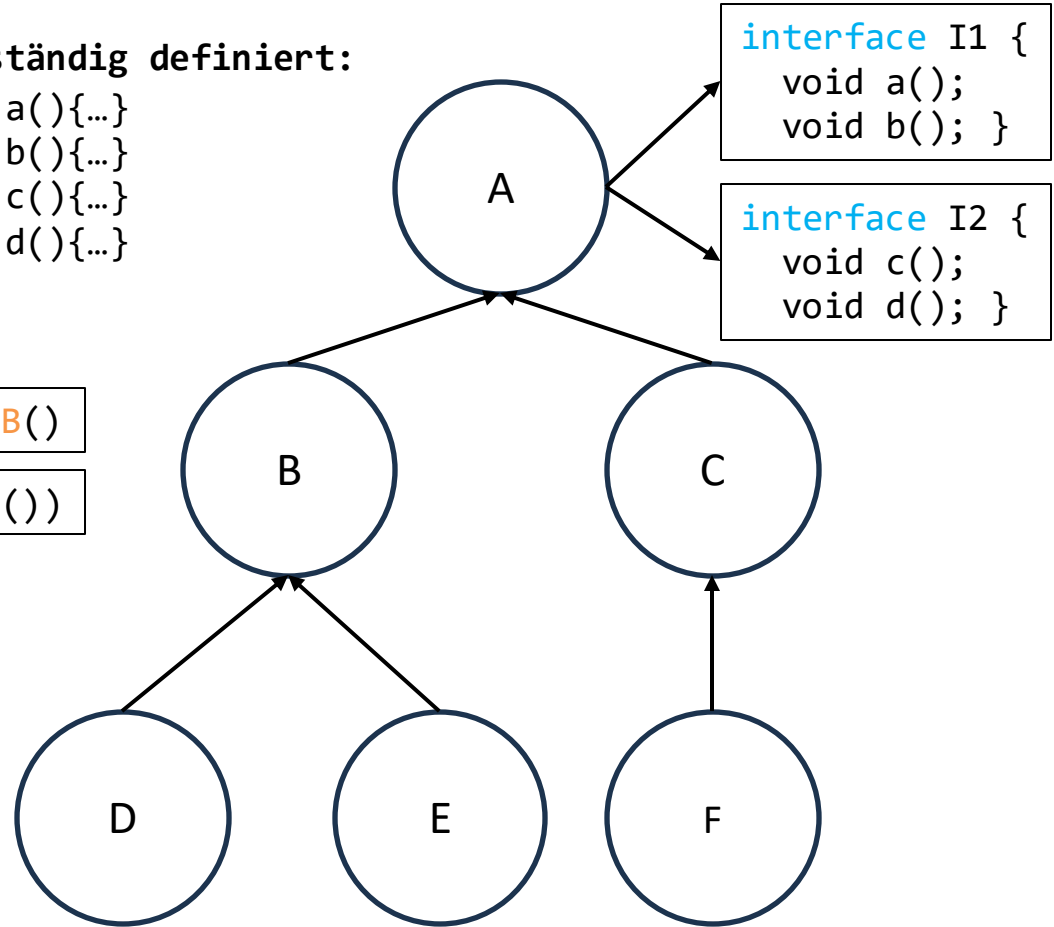
Vollständig definiert:

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



```
I2 h = new B()
```

```
println(h.a())
```



Kahoot

```

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

Kahoot