

**252-0027**

**Einführung in die Programmierung  
Übungen**

**Woche 11: Vererbung II**

**Henrik Pätzold  
Departement Informatik  
ETH Zürich**

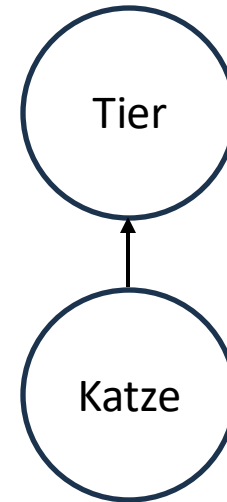
# Inheritance

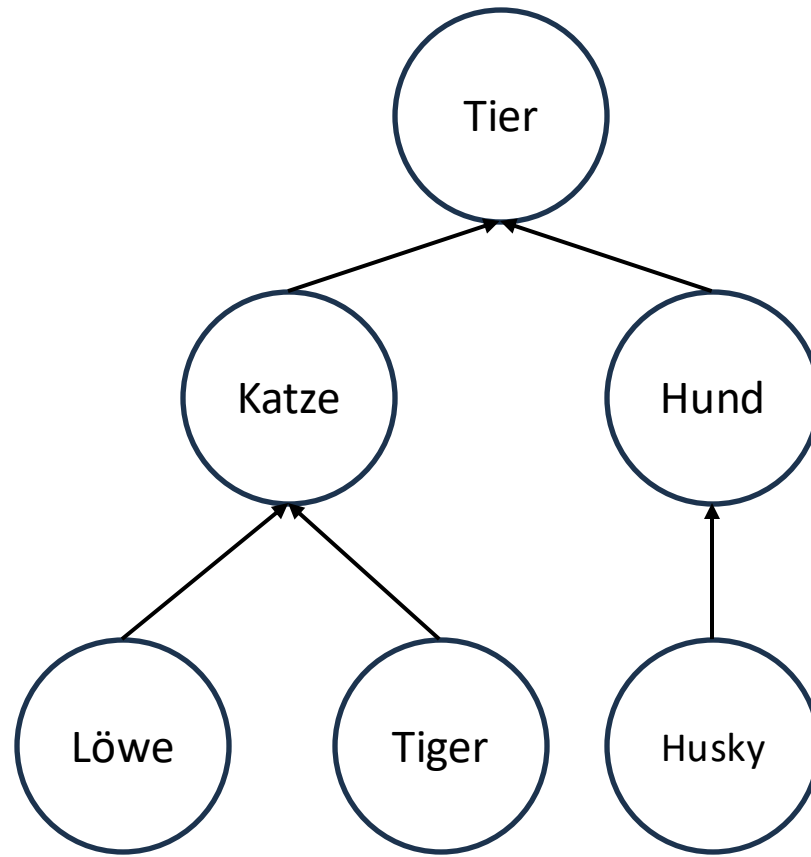
# Extends-Schlüsselwort

- **extends** spezifiziert, dass eine Klasse von einer anderen **erbt**
- Wir nennen die erbende Klasse im Folgenden Subklasse...
- und die vererbende Klasse Superklasse

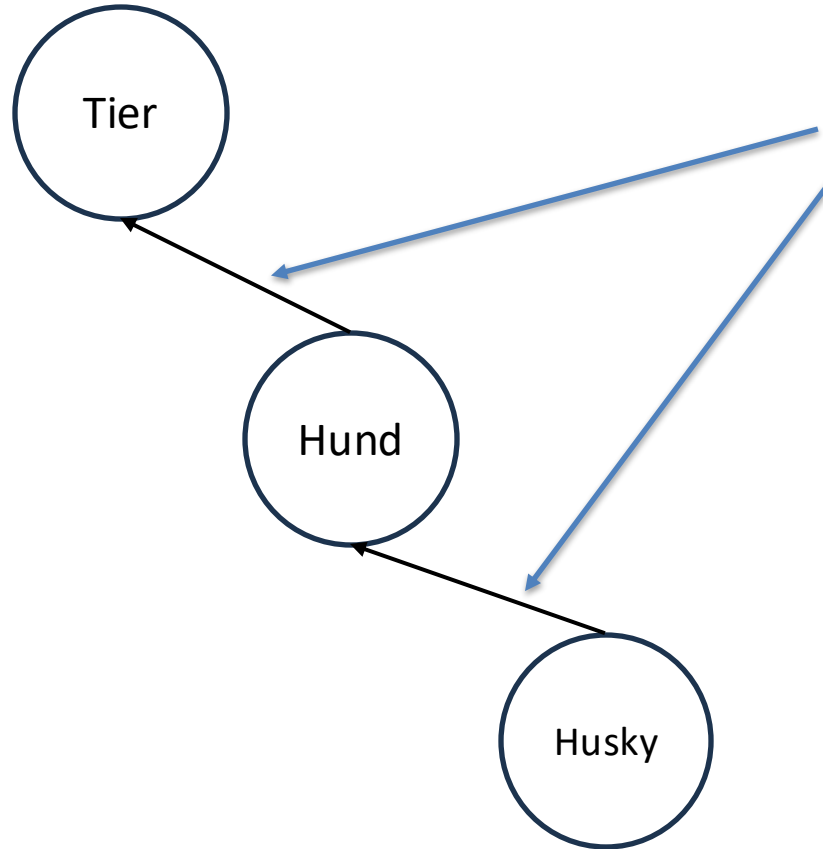


```
1 public class Katze extends Tier{}
```





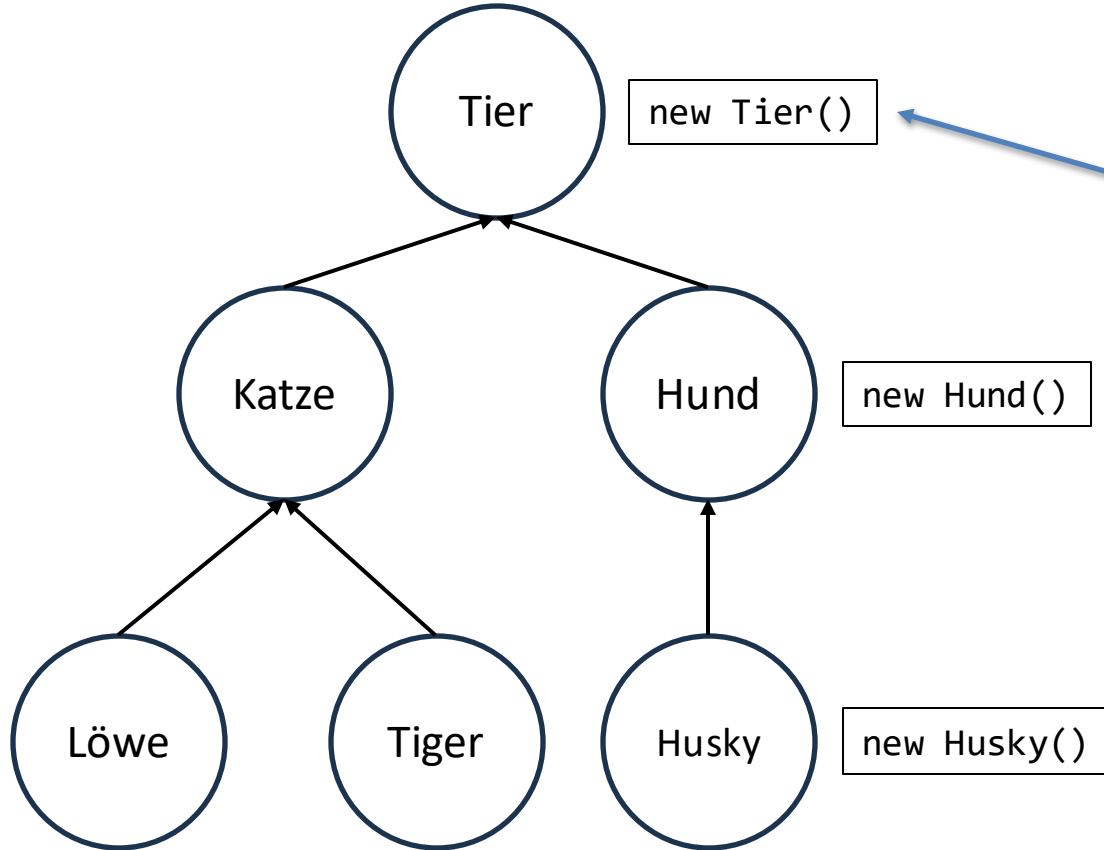
# Vererbungshierarchie



Die Pfeile sind eine  
“ist ein” - Beziehung

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

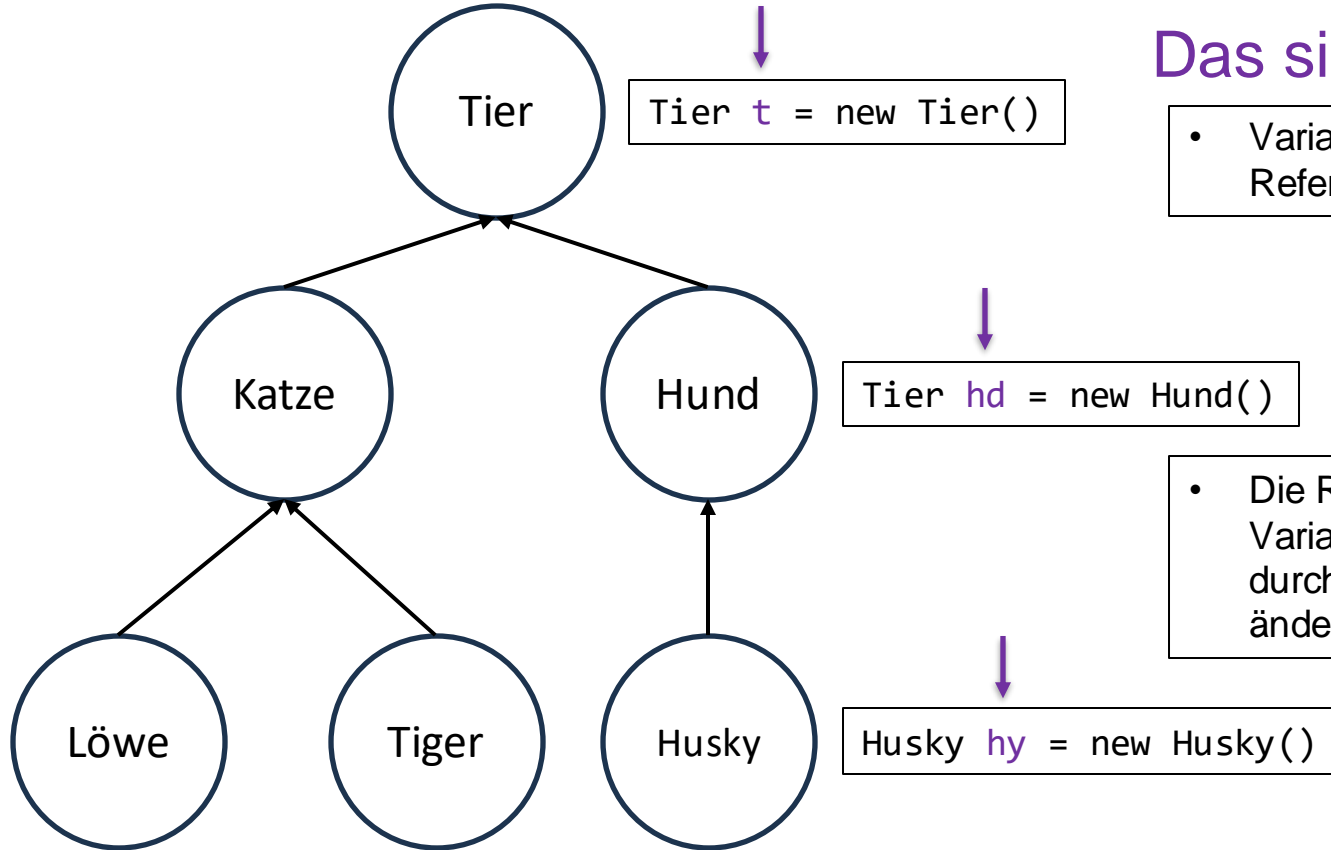
# Objekt vs Referenz



Das sind Objekte.

- Objekte ändern **nie** ihren Typ.
- Ein Husky-Objekt ist und bleibt ein Husky-Objekt.
- Ein Katzen-Objekt kann kein Löwen-Objekt werden.

# Objekt vs Referenz



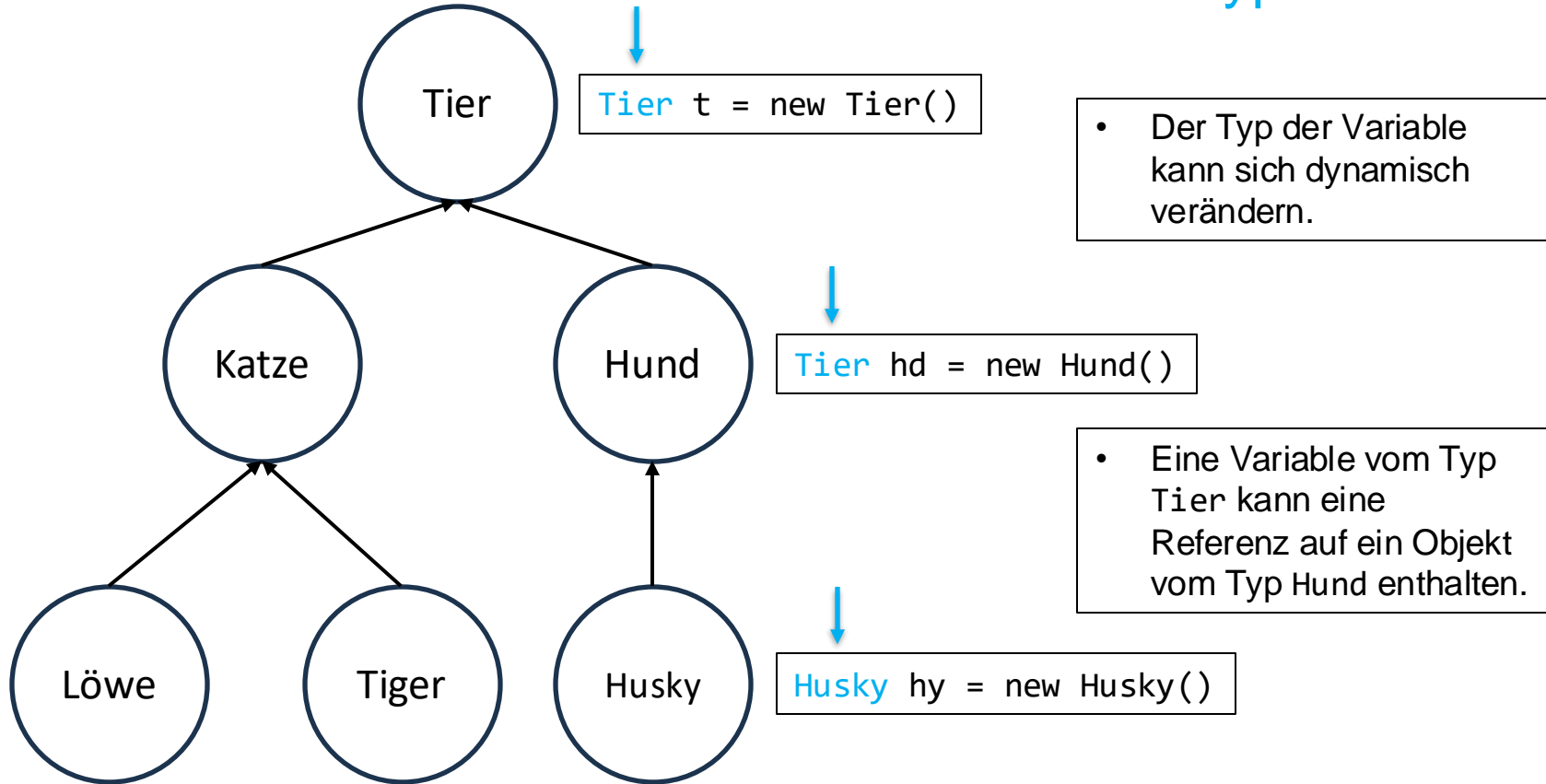
## Das sind Variablen.

- Variablen enthalten Referenzen oder Werte.

- Die Referenz in einer Variable kann sich durch Zuweisung mit = ändern.

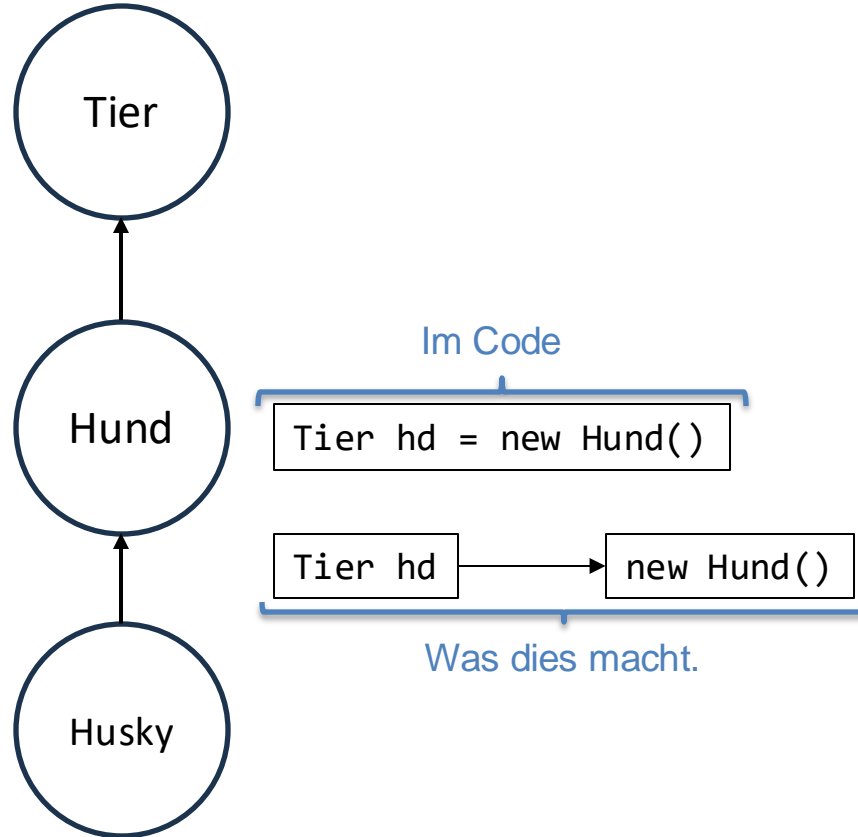
# Objekt vs Referenz

Das ist der Typ der Variable.

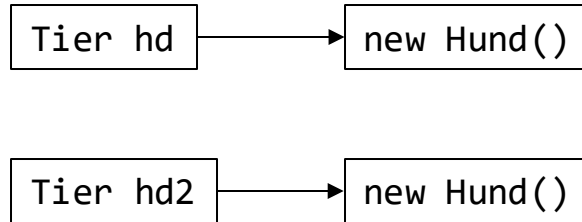
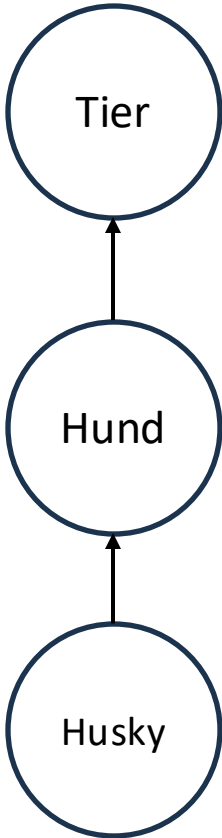




# Objekt vs Referenz

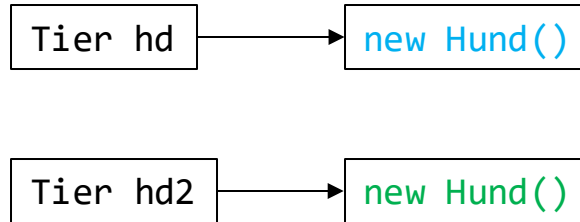
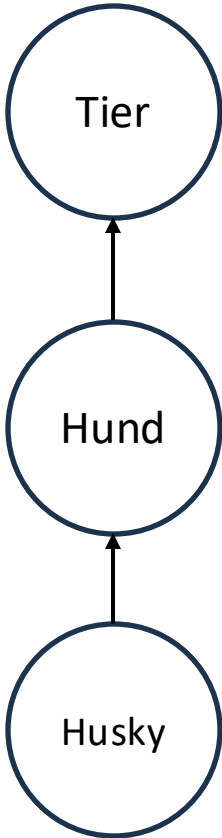


# Objekt vs Referenz



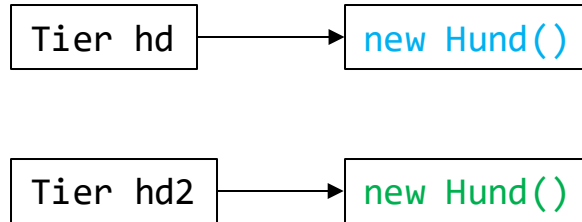
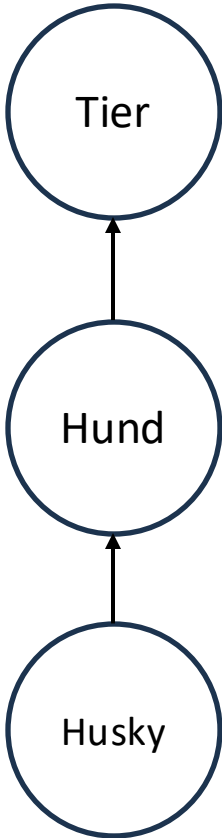
```
Tier hd = new Hund();  
Tier hd2 = new Hund();
```

# Objekt vs Referenz



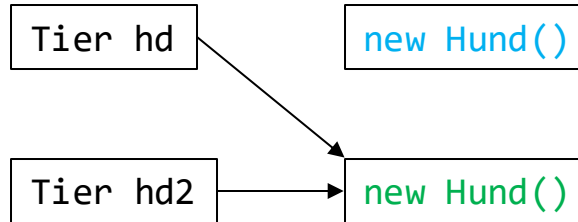
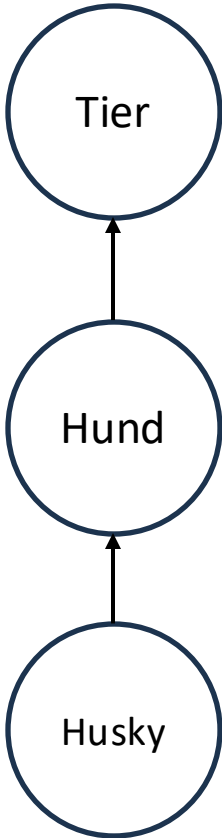
```
Tier hd = new Hund();  
Tier hd2 = new Hund();
```

# Objekt vs Referenz



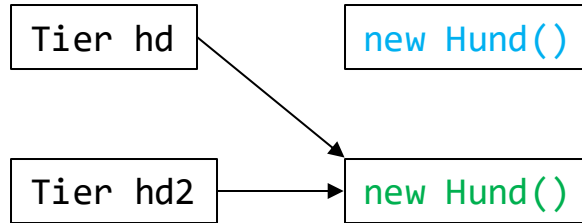
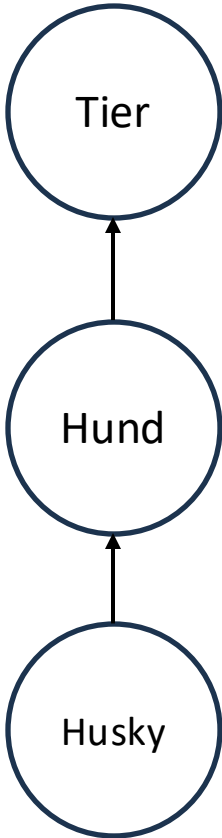
```
Tier hd = new Hund();  
  
Tier hd2 = new Hund();  
  
hd = hd2;
```

# Objekt vs Referenz



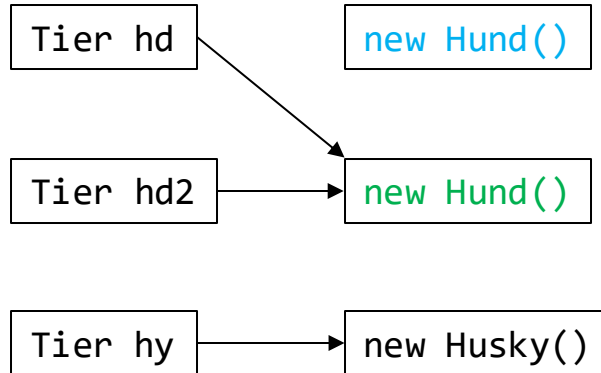
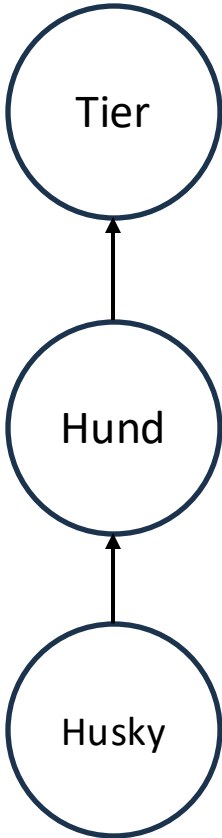
```
Tier hd = new Hund();  
  
Tier hd2 = new Hund();  
  
hd = hd2;
```

# Objekt vs Referenz



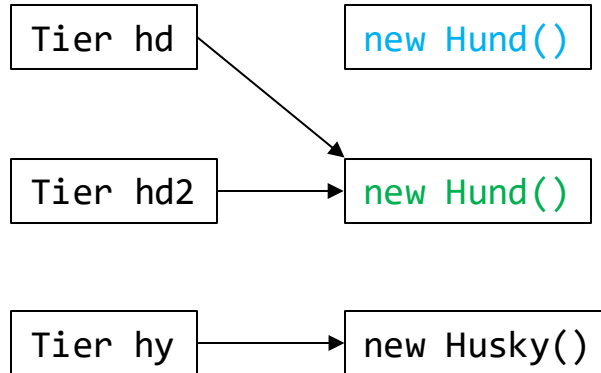
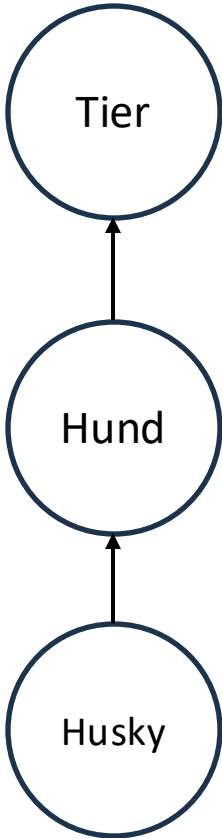
```
Tier hd = new Hund();  
Tier hd2 = new Hund();  
hd = hd2;  
Tier hy = new Husky();
```

# Objekt vs Referenz



```
Tier hd = new Hund();  
Tier hd2 = new Hund();  
hd = hd2;  
Tier hy = new Husky();
```

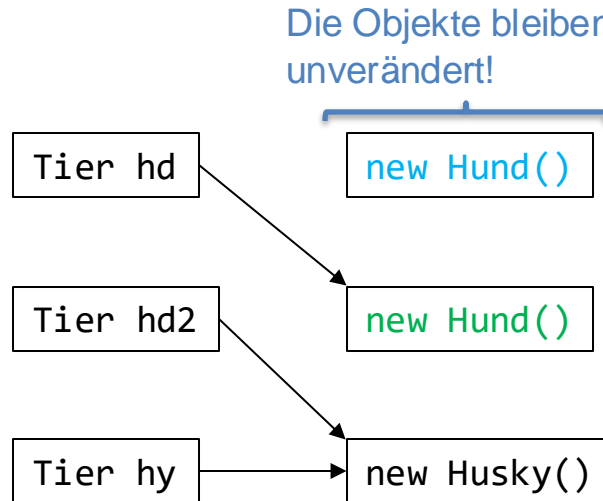
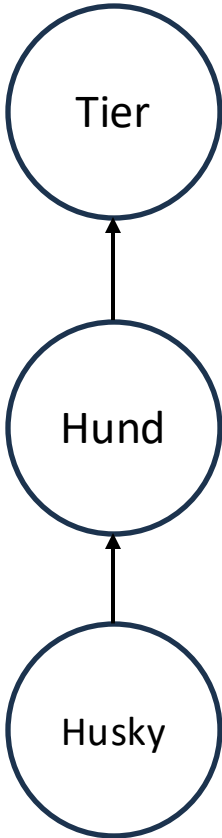
# Objekt vs Referenz



```
Tier hd = new Hund();  
Tier hd2 = new Hund();  
hd = hd2;  
Tier hy = new Husky();  
hd2 = hy;
```

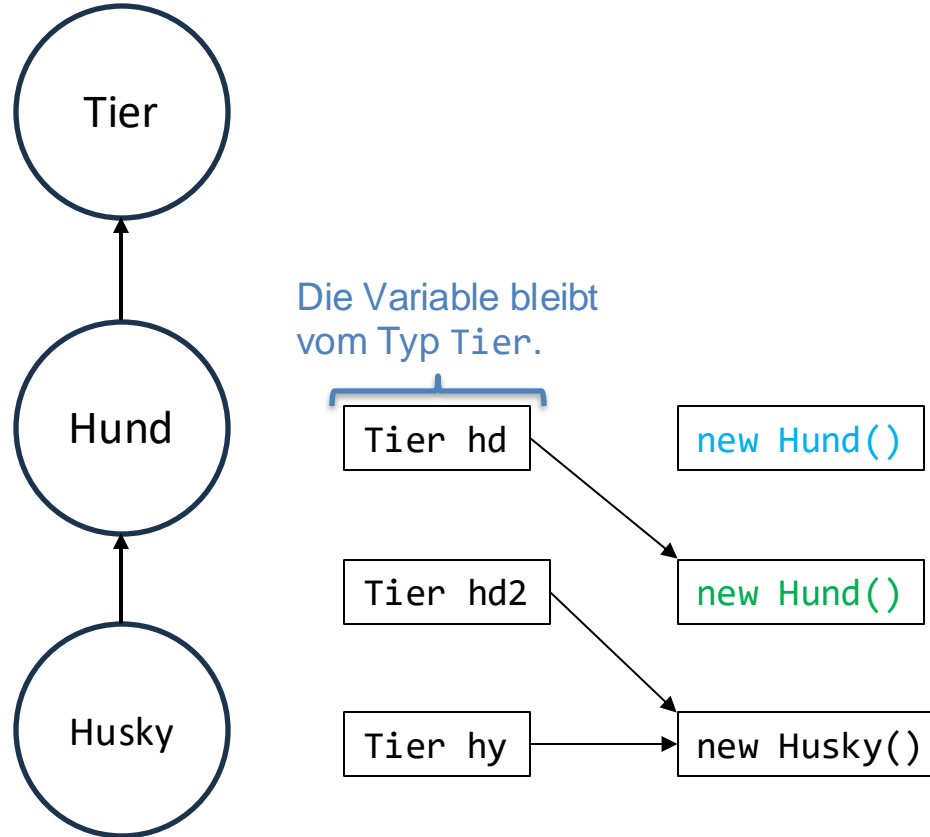


# Objekt vs Referenz



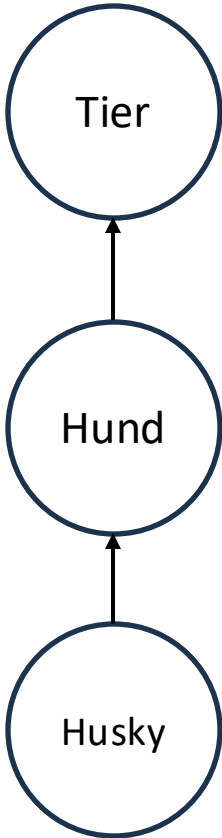
```
Tier hd = new Hund();  
Tier hd2 = new Hund();  
hd = hd2;  
Tier hy = new Husky();  
hd2 = hy;
```

# Objekt vs Referenz

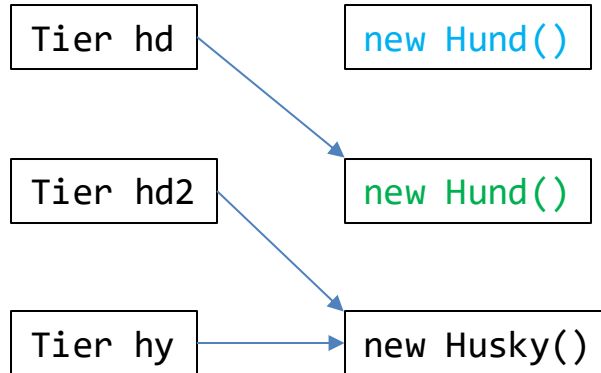


```
Tier hd = new Hund();  
  
Tier hd2 = new Hund();  
  
hd = hd2;  
  
Tier hy = new Husky();  
  
hd2 = hy;
```

# Objekt vs Referenz

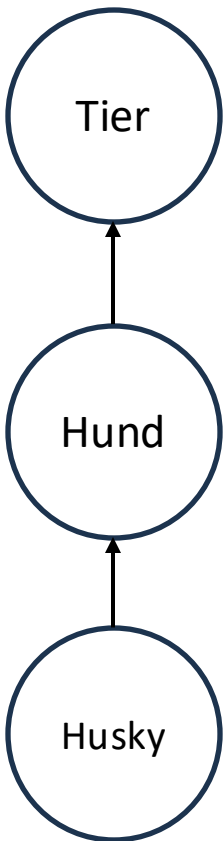


Die Referenzen in  
den Variablen  
verändern sich!



```
Tier hd = new Hund();  
  
Tier hd2 = new Hund();  
  
hd = hd2;  
  
Tier hy = new Husky();  
  
hd2 = hy;
```

# Objekt vs Referenz

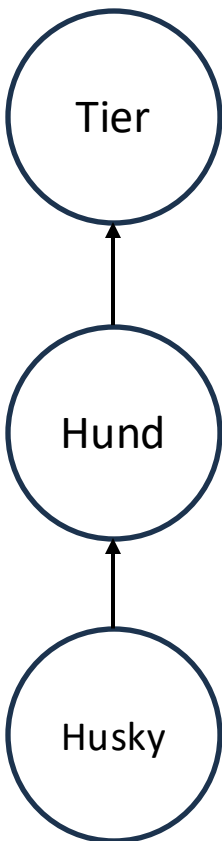


Wie unterscheiden wir zwischen:

- **Typ der Variable**
- **Typ des Objekts** auf welches die **Referenz** in der Variable **zeigt**?

```
Tier hd = new Husky()
```

# Objekt vs Referenz



Wie unterscheiden wir zwischen:

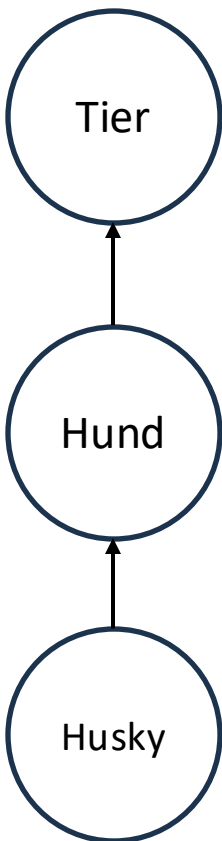
- **Typ** der **Variable**
- **Typ des Objekts** auf welches die **Referenz** in der Variable **zeigt**?

```
Tier hd = new Husky();
```

```
hd = new Hund();
```

```
hd = new Tier();
```

# Objekt vs Referenz

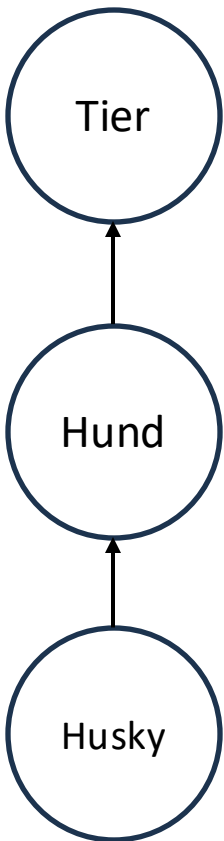


Wie unterscheiden wir zwischen:

- **Typ der Variable** (**Statischer Typ**)
- **Typ des Objekts** auf welches die **Referenz** in der Variable **zeigt**?

```
Tier hd = new Husky()
```

# Objekt vs Referenz

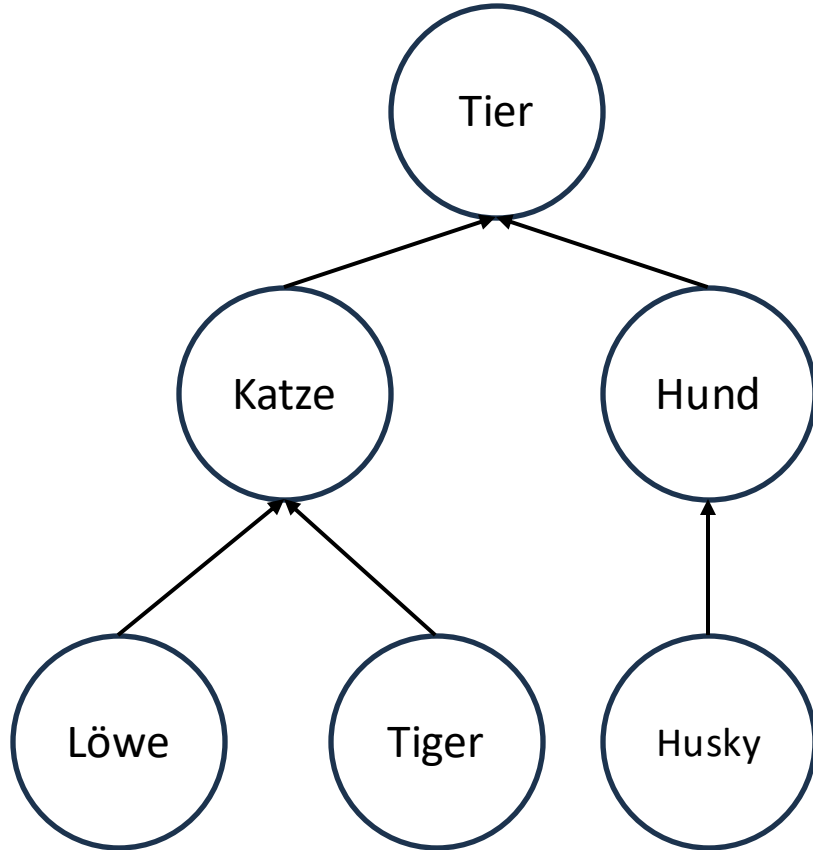


Wie unterscheiden wir zwischen:

- **Typ der Variable** (Statischer Typ)
- **Typ des Objekts** auf welches die **Referenz** in der Variable **zeigt?** (Dynamischer Typ)

```
Tier hd = new Husky()
```

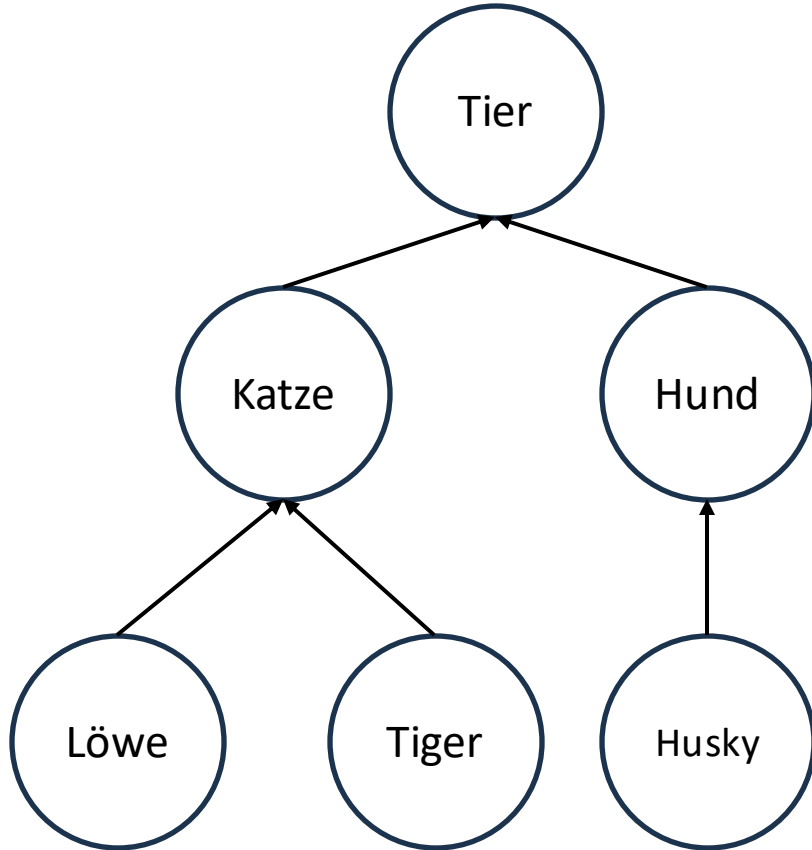
# Misconception: Objekte



Ein Objekt der Subklasse  
ist **immer** auch vom Typ  
der Superklasse.

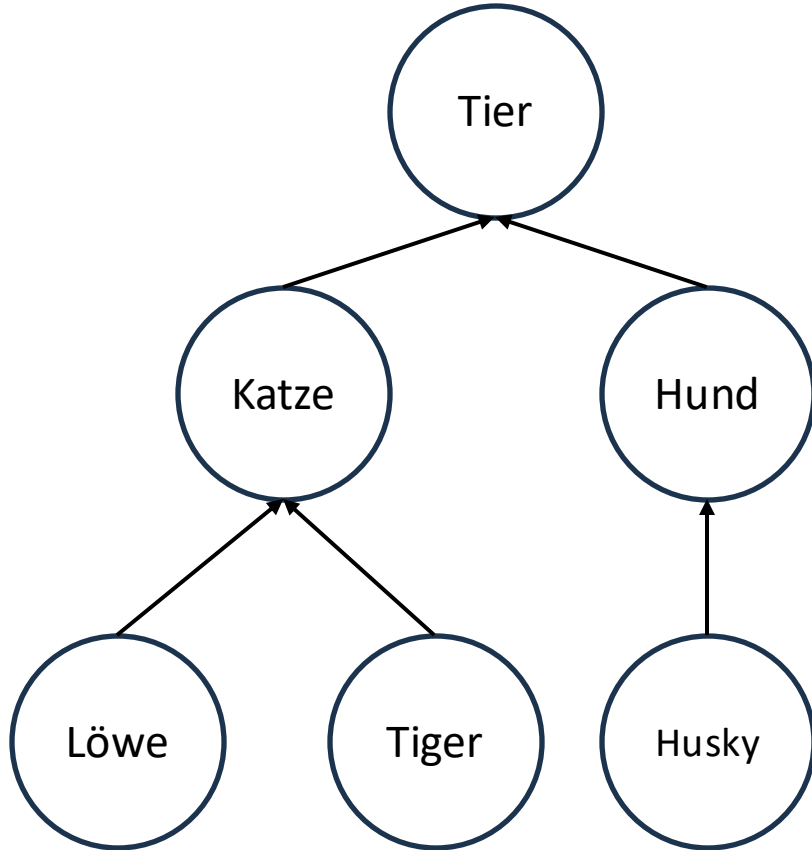


# Misconception: Objekte



Ein Objekt der Superklasse  
ist **nie** vom Typ der  
Subklasse.

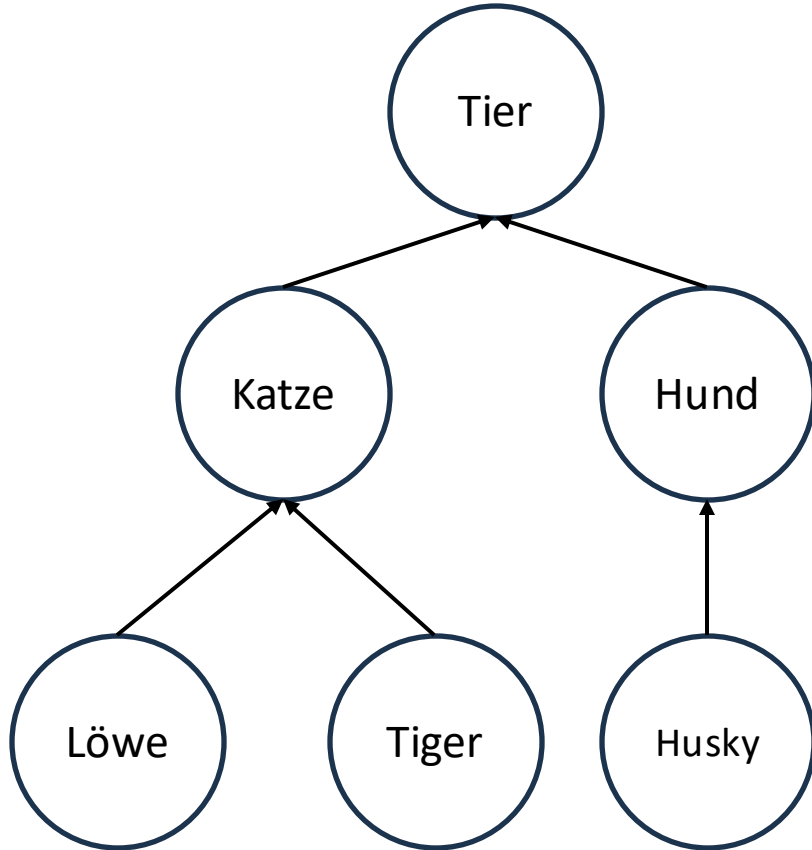
# Was wirklich passiert



Eine Variable vom Typ der Superklasse kann **immer** eine Referenz auf ein Objekt der Superklasse enthalten.

```
Tier hd = new Husky()
```

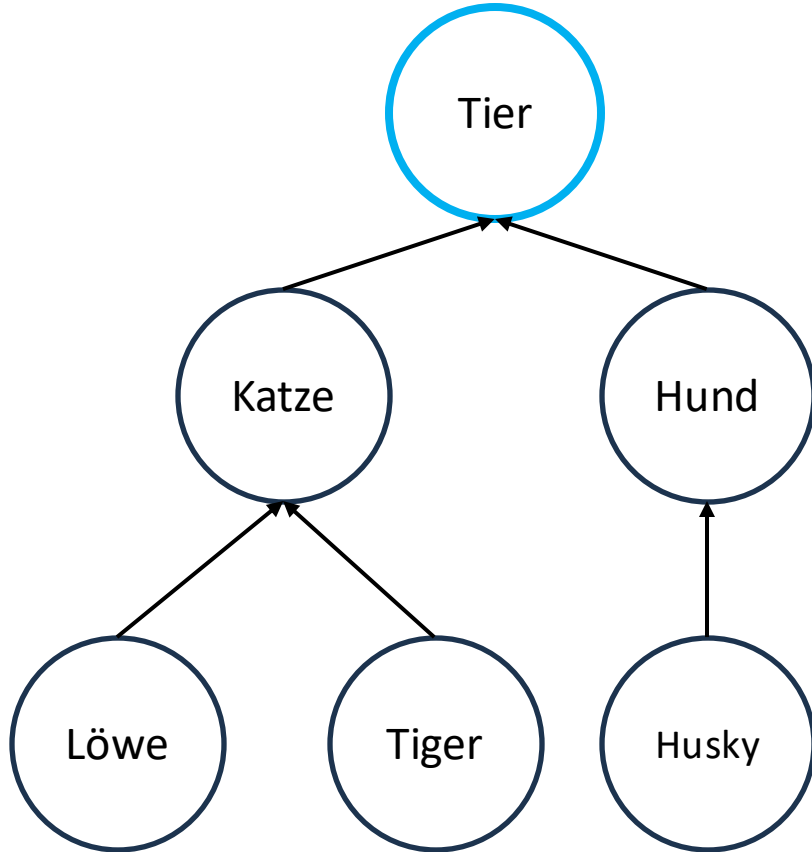
# Was wirklich passiert



Ein Variable vom Typ der Subklasse kann **nie** auf ein Objekt der Superklasse verweisen.

~~Husky hd = new Tier();~~

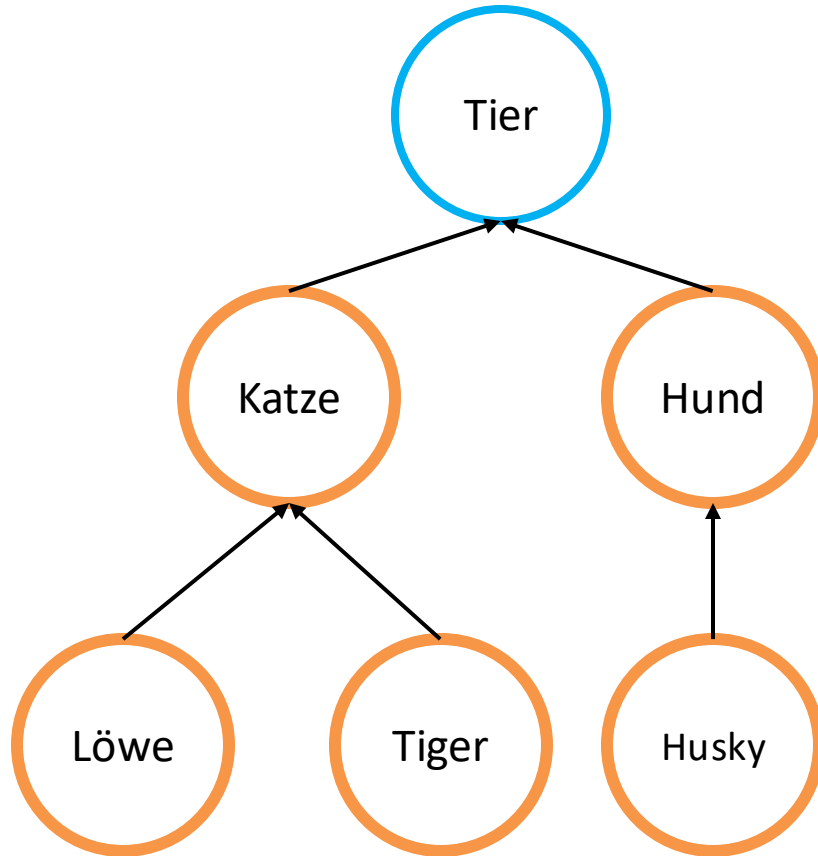
# Verdeutlicht – Was für Typen akzeptiert eine Variable?



Statischer Typ

Möglicher Dynamischer Typ

# Verdeutlicht – Was für Typen akzeptiert eine Variable?



Funktionieren diese Zuweisungen?

`Tier` hd = new `Husky`()



`Tier` k1 = new `Katze`()



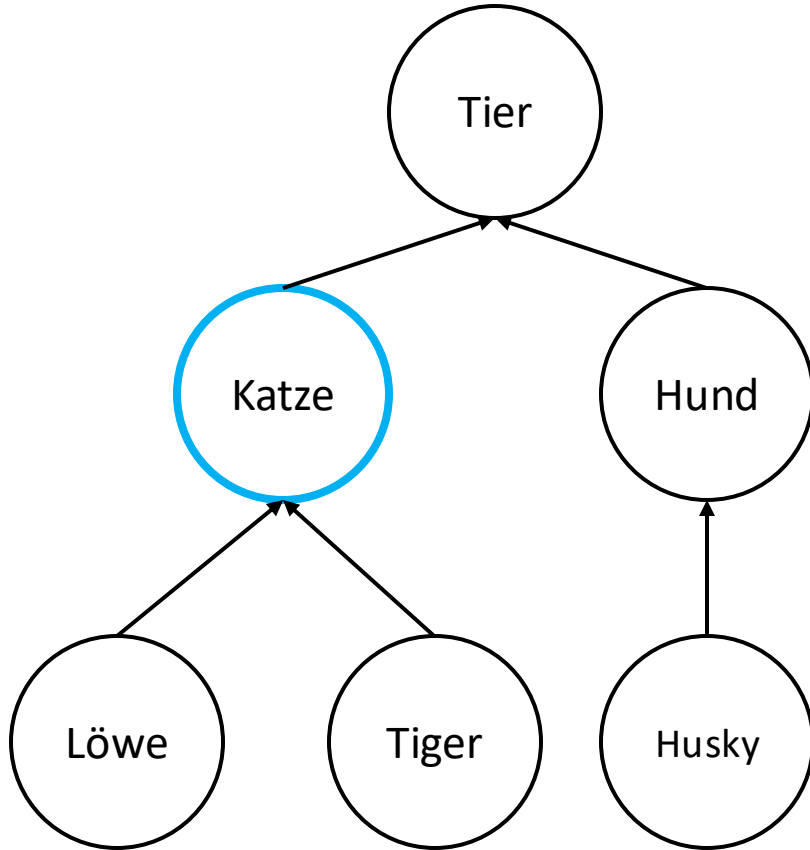
`Tier` t1 = new `Tiger`()



**Statischer Typ**

**Möglicher Dynamischer Typ**

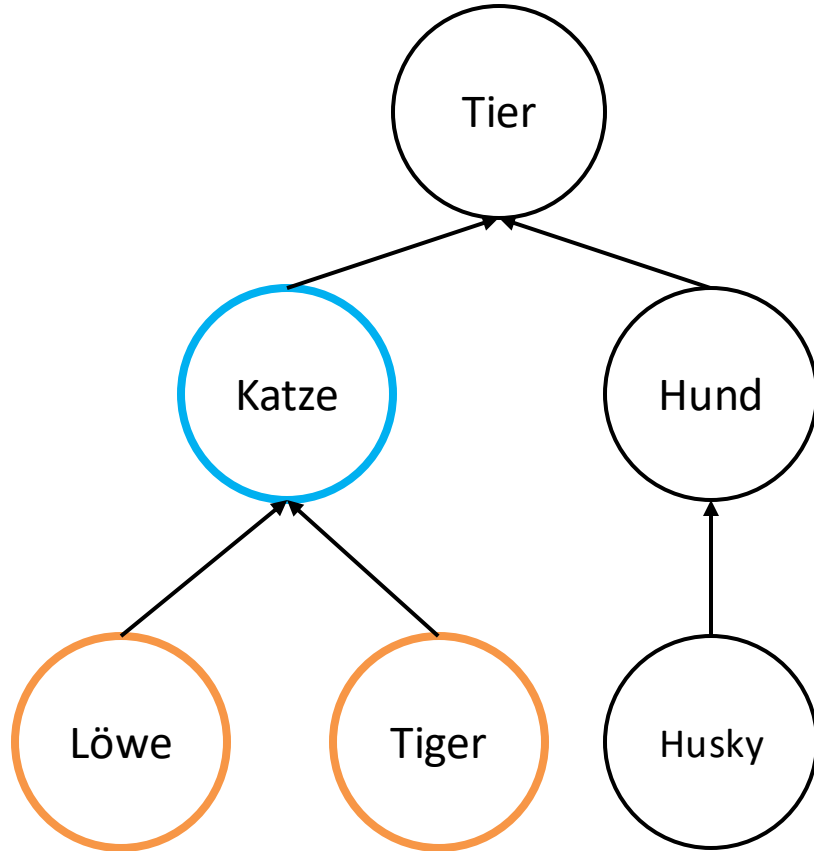
# Verdeutlicht – Was für Typen akzeptiert eine Variable?



Statischer Typ

Möglicher Dynamischer Typ

# Verdeutlicht – Was für Typen akzeptiert eine Variable?



Funktionieren diese Zuweisungen?

`Katze k1 = new Katze()`



`Tier t1 = new Tiger()`



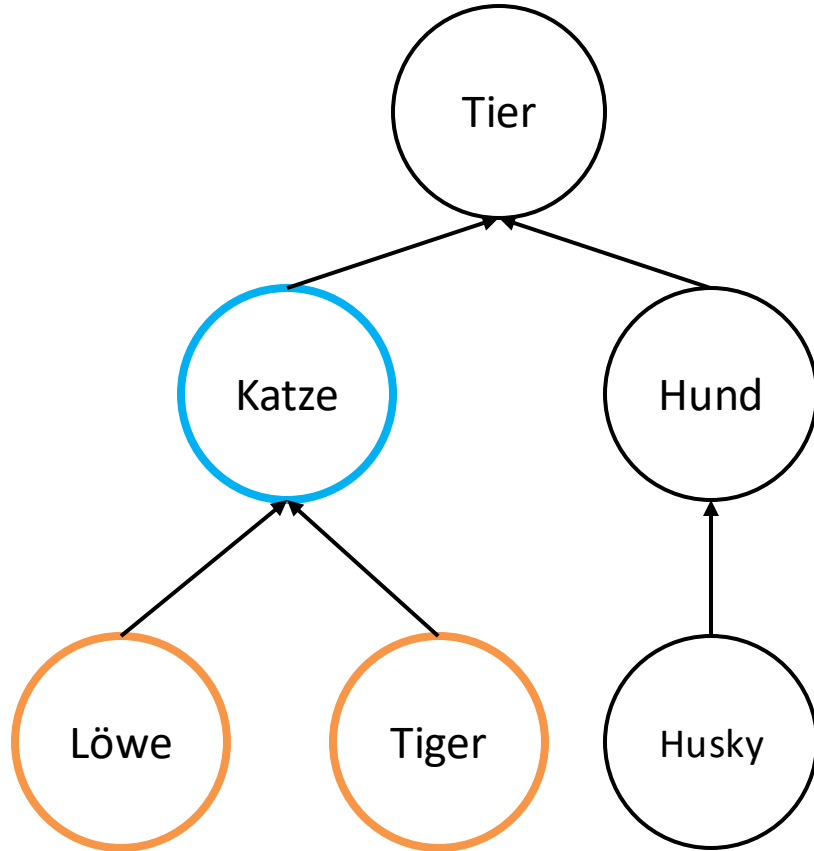
`Katze k3 = t1`



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?

Warum ist das problematisch?

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



```
Katze k3 = t1
```



Was sieht der Compiler? 

```
Katze = Tier
```

Die Zuweisung ist unsicher, da der Compiler **keine Garantie** hat, dass der dynamische Typ von **t1** mit dem Typ **Katze** kompatibel ist.  
**Intuitiv:** Nicht jedes Tier ist eine Katze.

# Verdeutlicht – Was für Typen akzeptiert eine Variable?

Warum ist das problematisch?

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



```
Katze k3 = t1
```



Was sieht der Compiler?



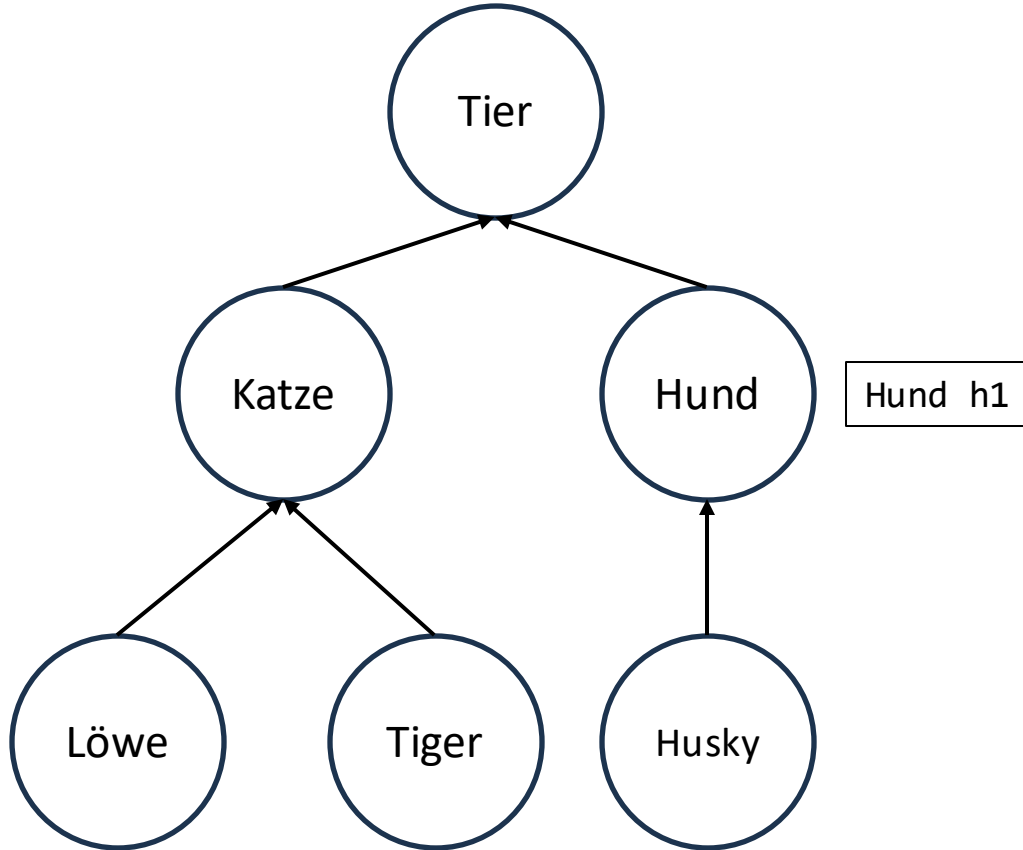
```
Katze = Tier
```

Aber wir probieren doch nur einen **Tiger** (subklasse der **Katze**) in **k3** zu speichern...

**Wir müssen dem Compiler garantieren können, dass er hier immer eine Subklasse von Katze bekommt, damit die Zuweisung legal ist.**

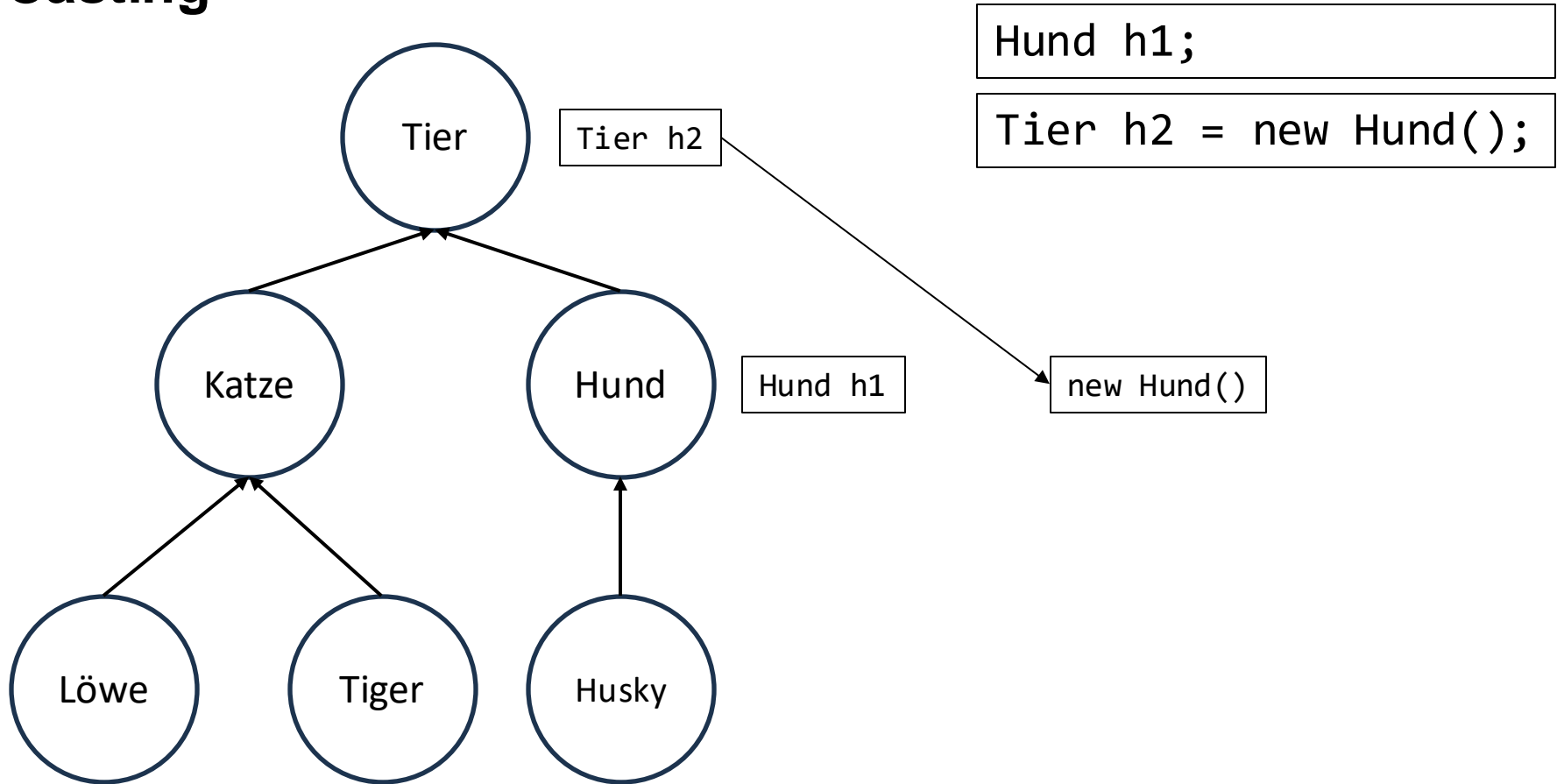
# Casting

# Casting

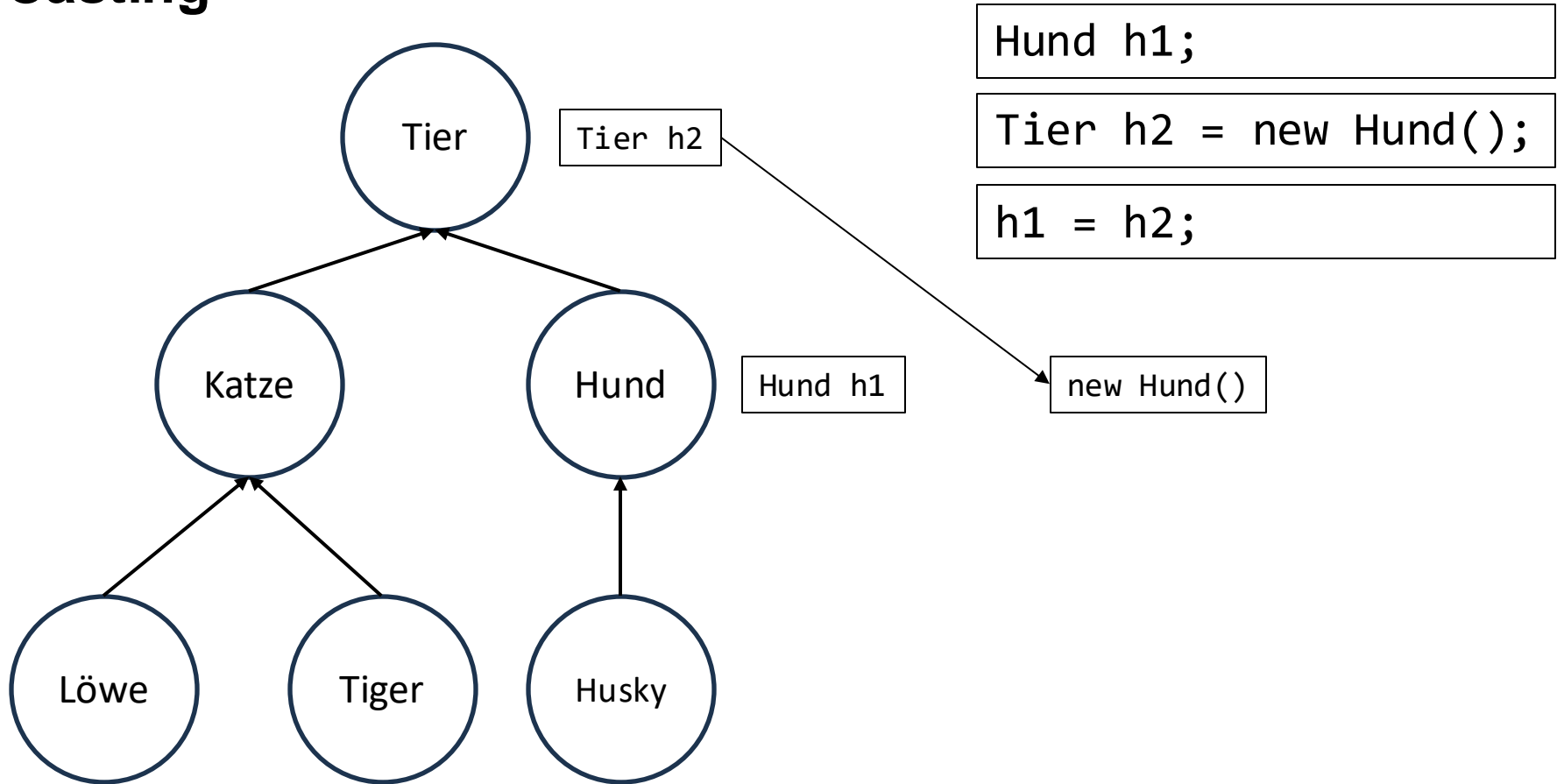


Hund h1;

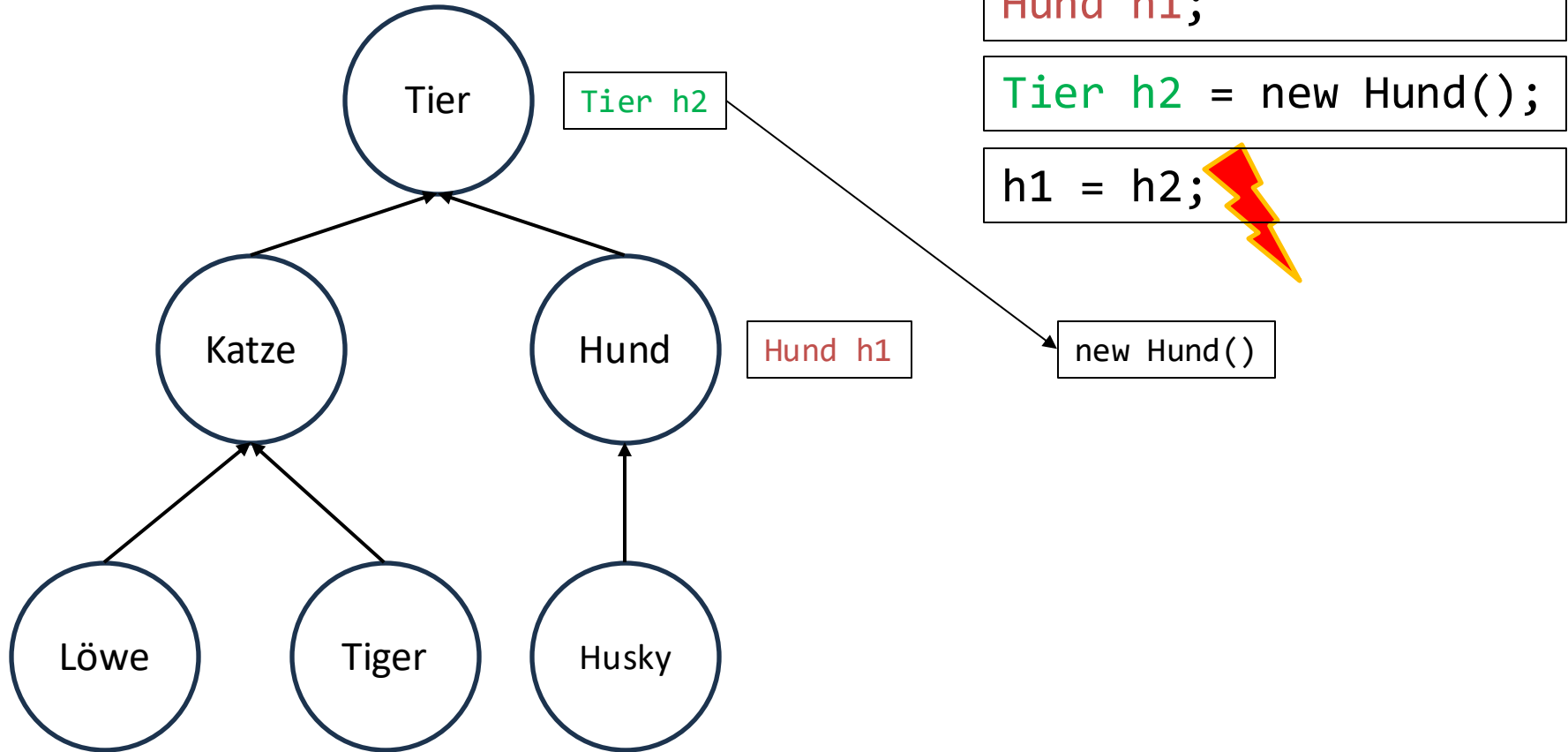
# Casting



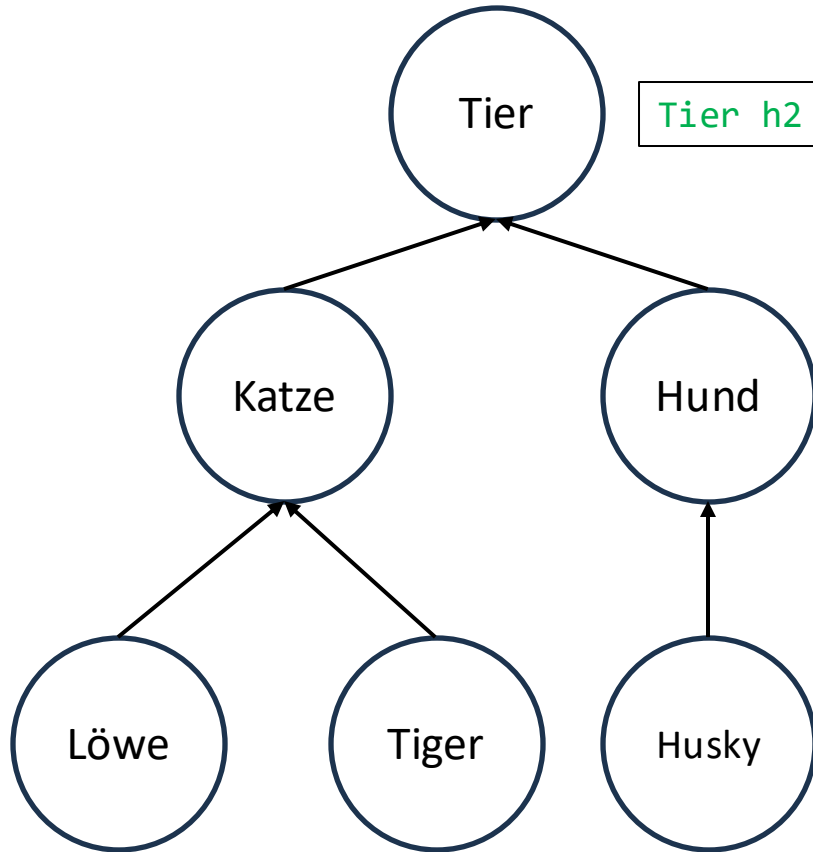
# Casting



# Casting



# Casting



Tier h2

Hund h1

```
Hund h1;
```

```
Tier h2 = new Hund();
```

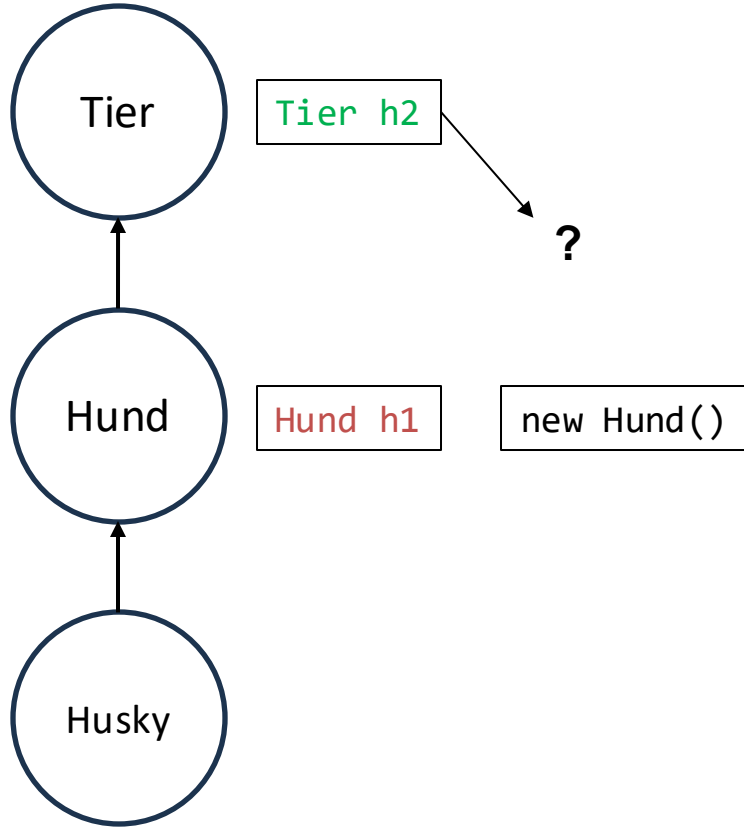
```
h1 = h2;
```


new Hund()

Exception in thread "main" java.lang.Error:  
Unresolved compilation problem:  
Type mismatch: cannot convert from  
Tier to Hund



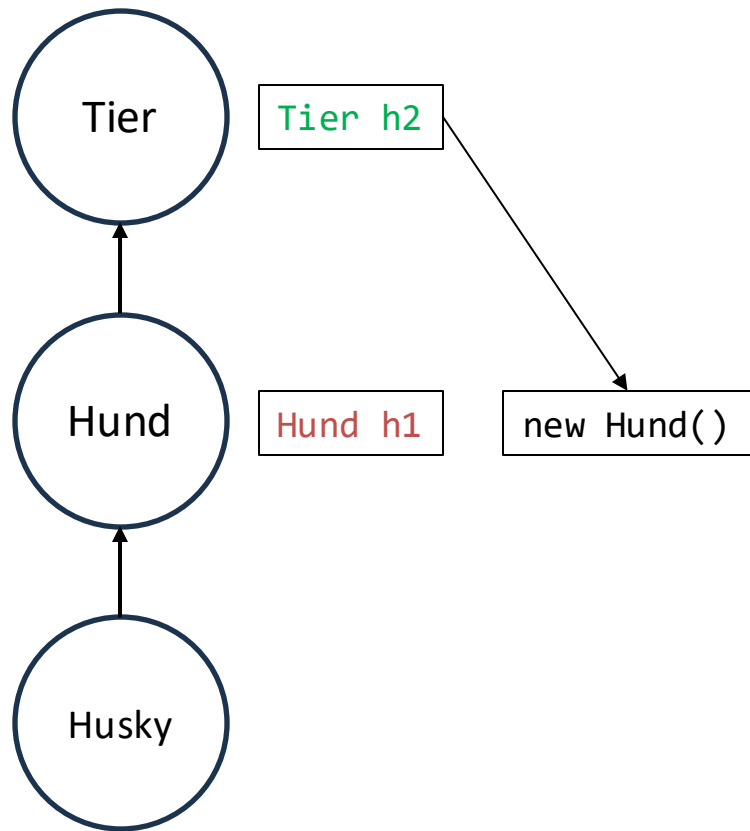
# Casting




```
void methode1(Tier h2) {  
    Hund h1;  
    h1 = h2;   
}
```

**Hier kennen wir den dynamischen Typ nicht...**

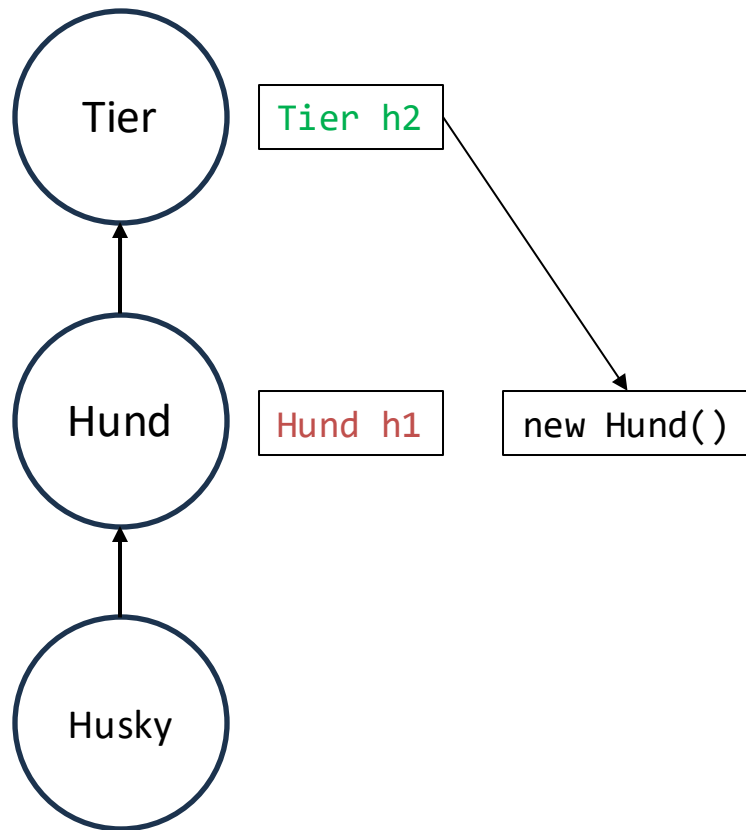
# Casting



```
void methode1(Tier h2) {  
  
    Hund h1;  
  
    h1 = h2;   
}
```

Wenn wir methode1 nur aufrufen, wenn h2 eine Referenz auf ein Objekt vom Typ Hund enthält, dann würde eigentlich `h1 = h2` immer gehen!

# Casting

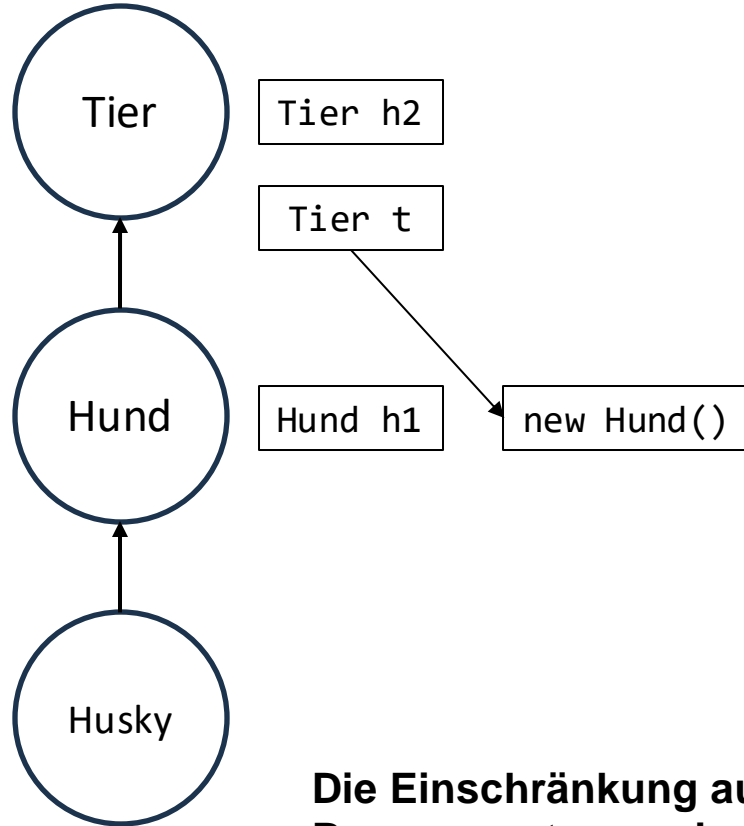


```
void methode1(Tier h2) {  
    Hund h1;  
    h1 = (Hund) h2;  
}
```

Wenn wir methode1 nur aufrufen, wenn h2 eine Referenz auf ein Objekt vom Typ Hund enthält, dann würde eigentlich `h1 = h2` immer gehen!

Ein Cast ist ein Versprechen an den Compiler, dass dies der Fall ist.

# Casting



```
void methode1(Tier h2) {  
    Hund h1;  
    h1 = (Hund) h2;  
}
```

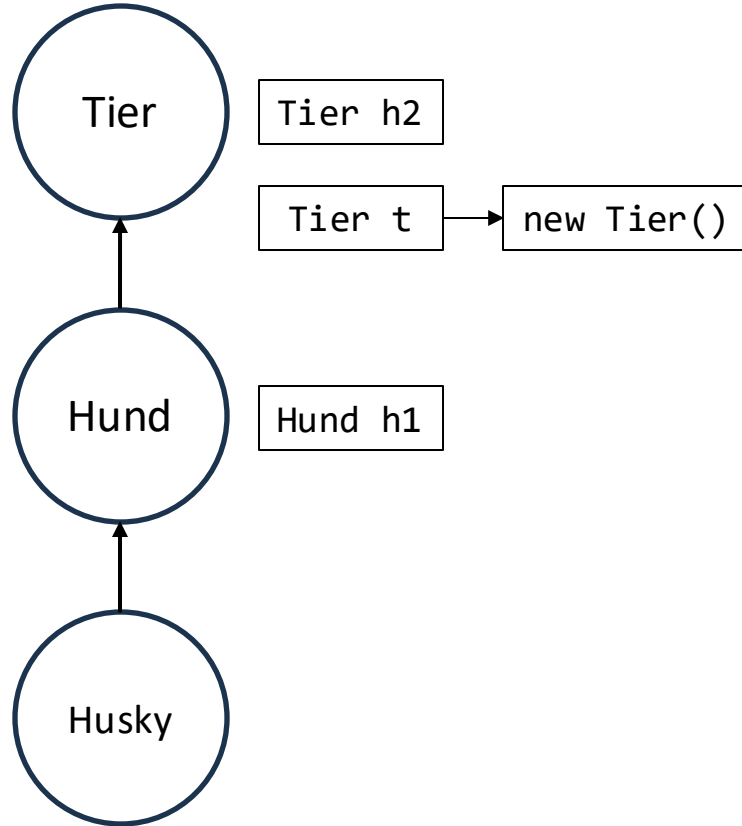
```
void methode2() {  
    Tier t = new Hund();  
    methode1(t);  
}
```

**Geht das?**



**Die Einschränkung auf einen Typen weiter unten im Baum nennt man einen Downcast.**

# Casting

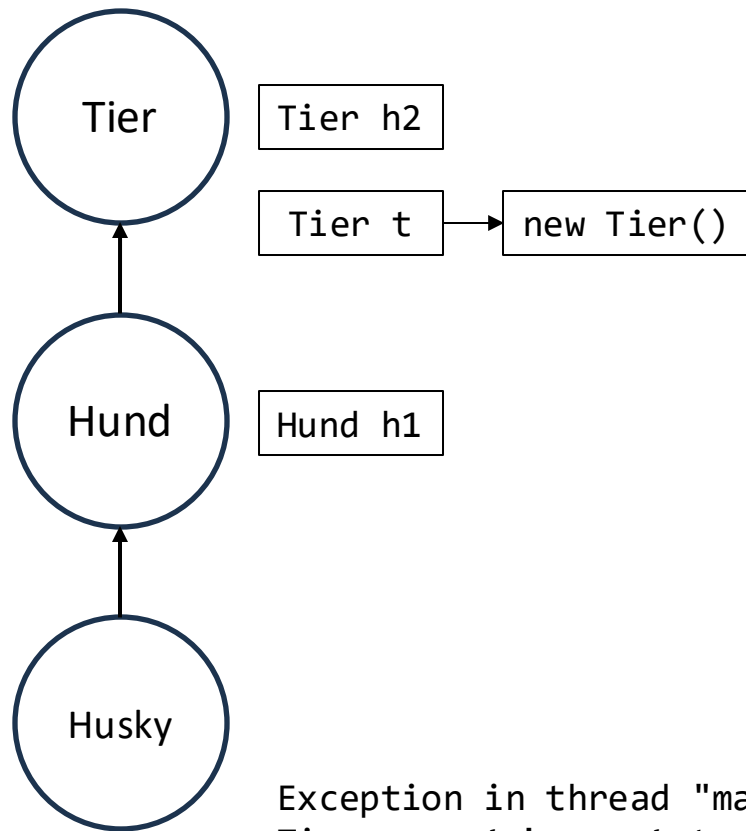


```
void methode1(Tier h2) {  
    Hund h1;  
    h1 = (Hund) h2;  
}
```

```
void methode2() {  
    Tier t = new Tier();  
    methode1(t);  
}
```

## Geht das?

# Casting



```
void methode1(Tier h2) {  
    Hund h1;  
    h1 = (Hund) h2;  
}
```

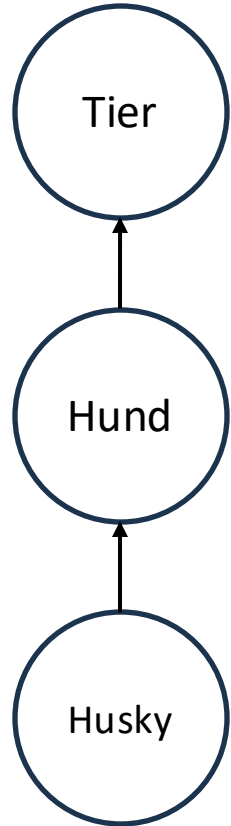


```
void methode2() {  
    Tier t = new Tier();  
    methode1(t);  
}
```

**Geht das? X**

Exception in thread "main" java.lang.ClassCastException: class Tier **cannot be cast** to class Hund

# Casting: Laufzeitfehler vs Compiler Fehler



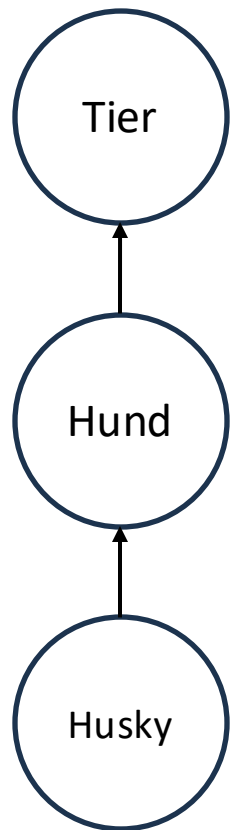
```
Hund h = new Tier();
```



```
Exception in thread "main" java.lang.Error: Unresolved  
compilation problem:  
    Type mismatch: cannot convert from Tier to Hund
```

**Compiler-Fehler:** Die Typen sind **nie** kompatibel.

# Casting: Laufzeitfehler vs Compiler Fehler



```
Tier t = new Tier();  
Hund h = (Hund) t;
```



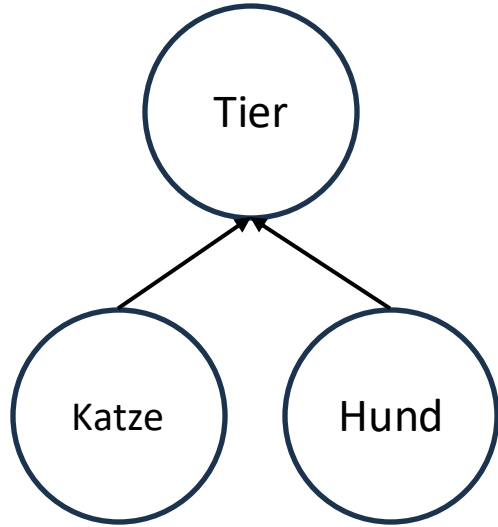
```
Exception in thread "main" java.lang.ClassCastException:  
class Tier cannot be cast to class Hund
```

**Laufzeitfehler:** Die Typen sind zwar nie kompatibel, aber das Versprechen (der Cast) an den Compiler lässt das Programm kompilieren.

- Beim Ausführen gibt es einen Laufzeitfehler.



# Casting: Laufzeitfehler vs Compiler Fehler

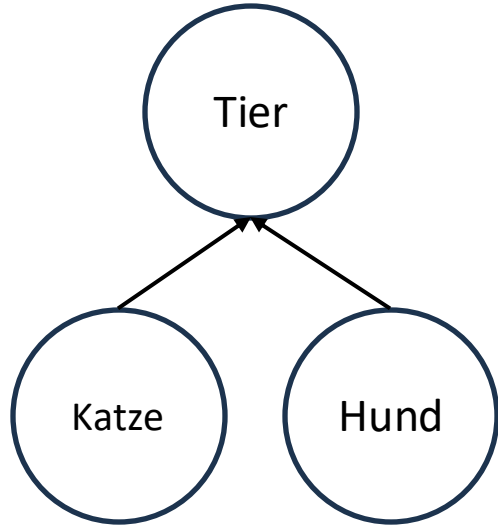


```
Tier k = new Katze();  
Tier h = new Hund();  
  
h = k;
```

**Geht das?**



# Casting: Laufzeitfehler vs Compiler Fehler



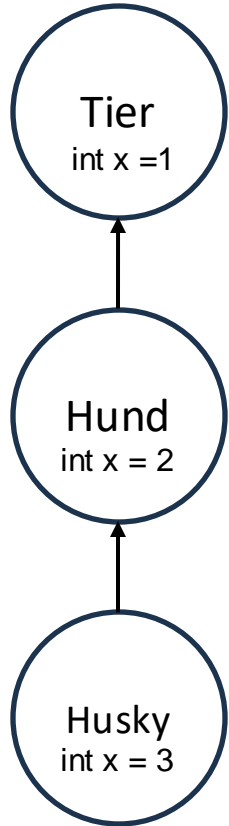
```
Tier k = new Katze();  
Hund h = new Hund();  
  
h = k;
```

**Geht das?** 

```
Exception in thread "main" java.lang.Error: Unresolved  
compilation problem:  
    Type mismatch: cannot convert from Tier to Hund
```

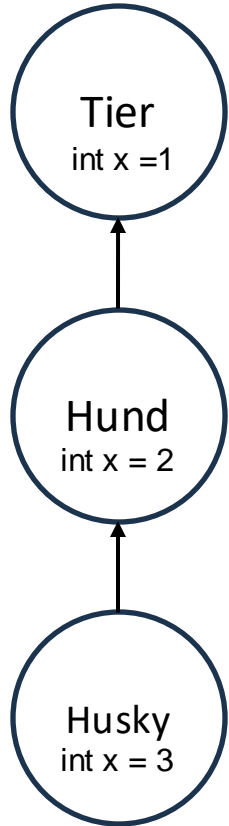
# Attributwahl

# Attribute:



**Regel:** Attribute werden anhand vom **statischen Typ** ausgewählt.

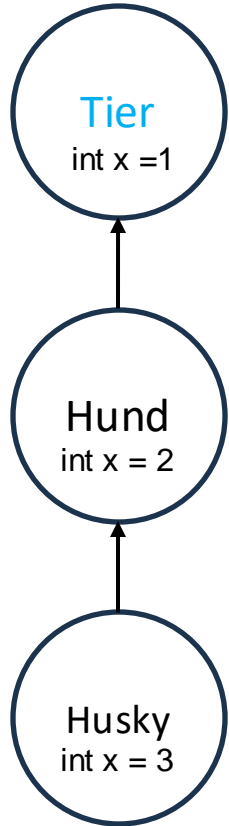
# Attribute:



**Regel:** Attribute werden an Hand vom **statischen Typ** ausgewählt.

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

# Attribute:

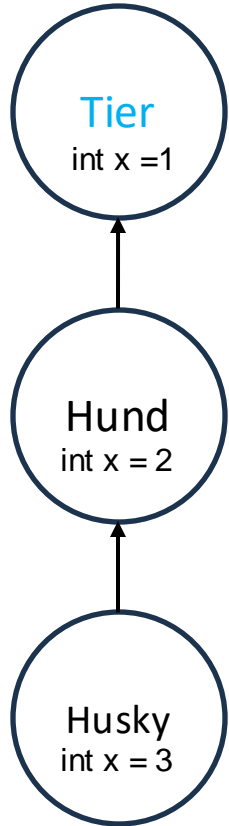


**Regel:** Attribute werden an Hand vom **statischen Typ** ausgewählt.

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

Resultat: 1

# Attribute:

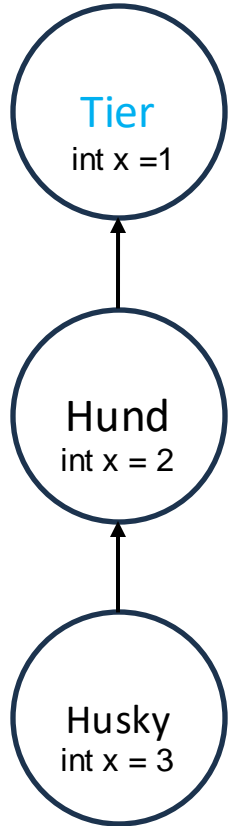


**Regel:** Attribute werden an Hand vom **statischen Typ** ausgewählt.

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

Resultat: 1

# Attribute:



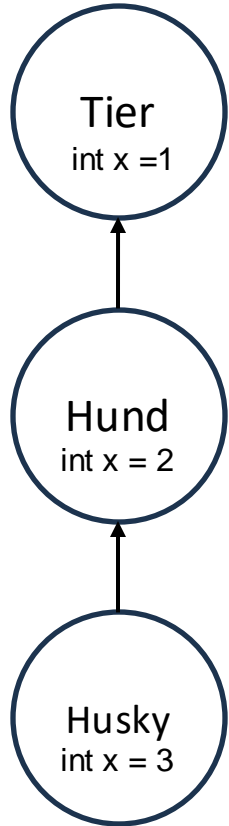
**Regel:** Attribute werden an Hand vom **statischen Typ** ausgewählt.

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

Resultat: 1



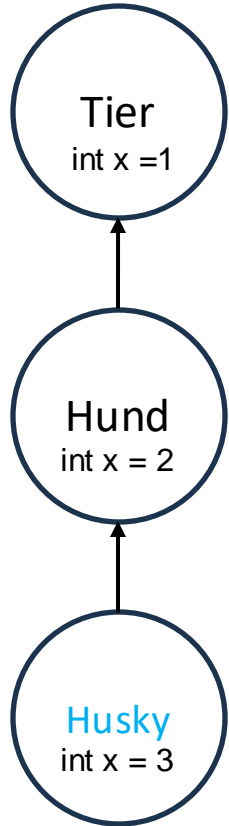
# Attribute:



**Regel:** Attribute werden an Hand vom **statischen Typ** ausgewählt.

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

# Attribute:

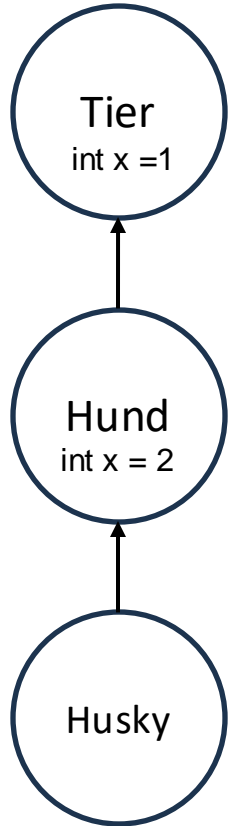


**Regel:** Attribute werden an Hand vom **statischen Typ** ausgewählt.

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

Resultat: 3

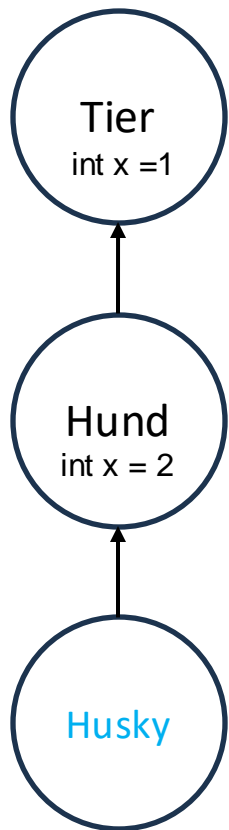
# Attribute:



**Regel:** Attribute werden an Hand vom **statischen Typ** ausgewählt.

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

# Attribute:



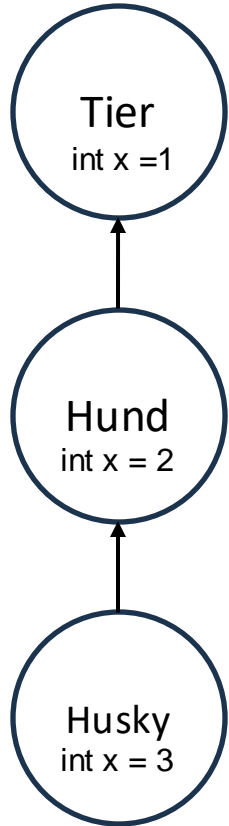
**Regel:** Attribute werden an Hand vom **statischen Typ** ausgewählt.

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

Resultat: 2

Einer Klasse stehen grundsätzlich alle **nicht private** Variablen der Superklasse zur Verfügung. Wird die Variable explizit deklariert, wird die vererbte Variable “verdeckt” und ist nicht mehr zugänglich.

# Attribute:

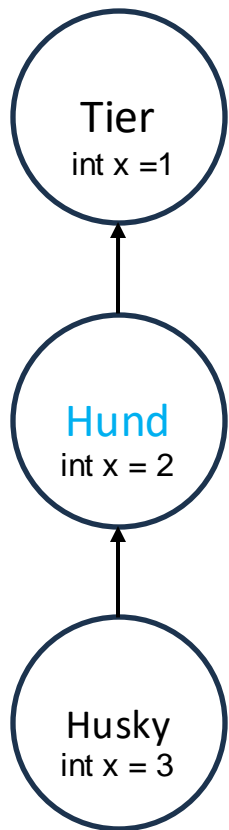


**Regel:** Attribute werden an Hand vom **statischen Typ** ausgewählt.

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

Resultat: 2

# Attribute:



**Regel:** Attribute werden an Hand vom **statischen Typ** ausgewählt.

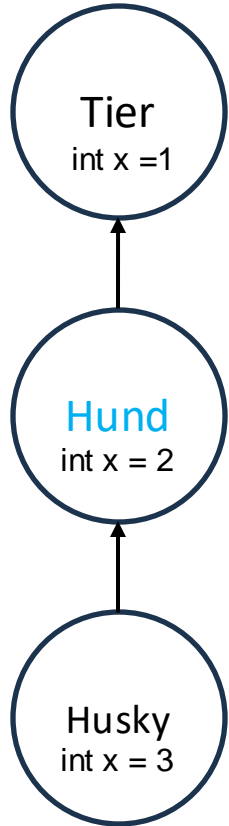
```
Husky t = new Husky();
```

```
System.out.println(((Hund)t).x);
```

Resultat: 2

Durch den Cast sehen wir: Das neu definieren von **x** hat **keine** Auswirkung auf **x** der Superklasse.

# Attribute:



```
Husky t = new Husky();
```

```
System.out.println(((Hund)t).x);
```



Hier wird implizit eine neue, temporäre Variable mit statischem Typ Hund erstellt.

```
Husky t = new Husky();
```

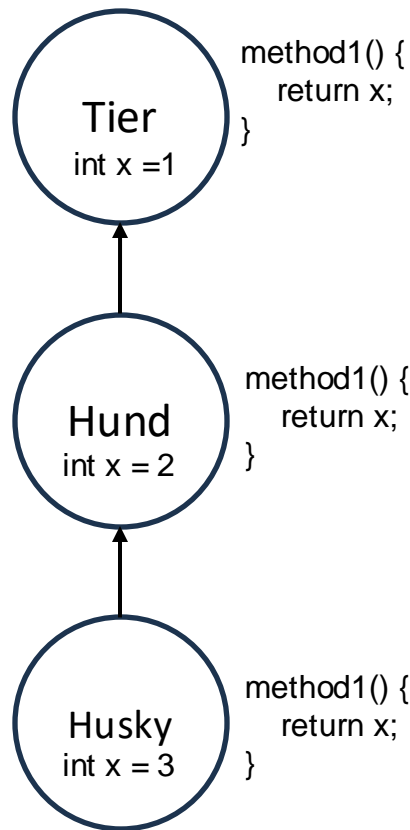
```
Hund casted_t = (Hund) t;
```

```
System.out.println(casted_t.x);
```

# Methodenwahl



## Methoden:



**Regel:** Methoden\* werden anhand vom **dynamischen Typ** ausgewählt.

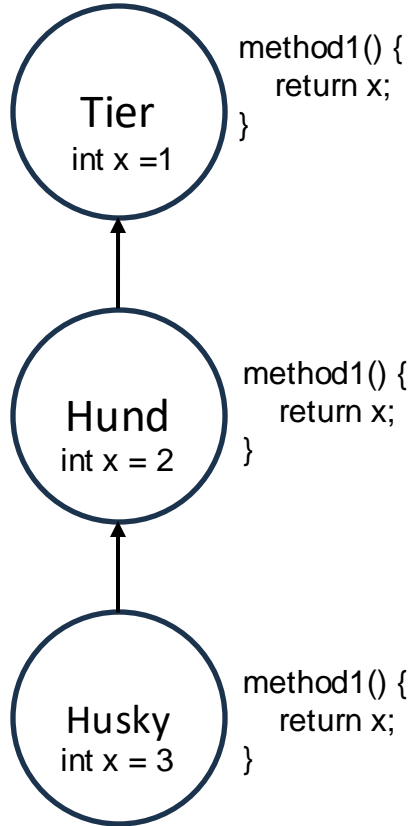
\* ausser private, static und final Methoden (hier statischer Typ)

```
Tier t = new Husky();
```

```
System.out.println(t.method1());
```

Resultat: 3

# Methoden:



**Regel:** Methoden\* werden an Hand vom **dynamischen Typ** ausgewählt.

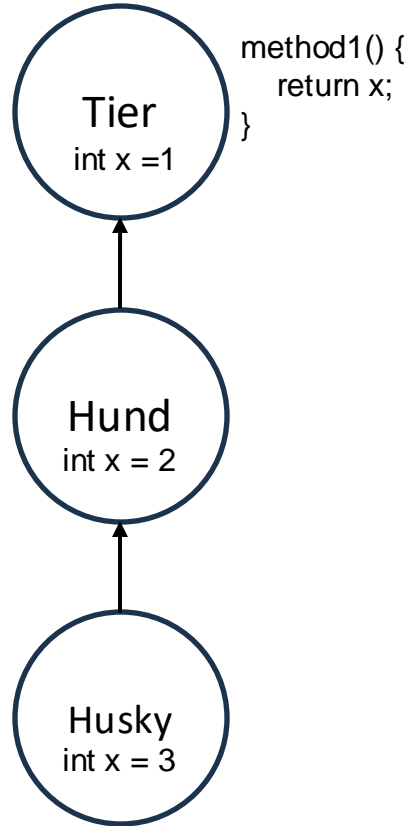
\* ausser private, static und final Methoden (hier statischer Typ)

```
Hund t = new Hund();
```

```
System.out.println(t.method1());
```

Resultat: 2

# Methoden:



**Regel:** Methoden\* werden an Hand vom **dynamischen Typ** ausgewählt.

\* ausser private, static und final Methoden (hier statischer Typ)

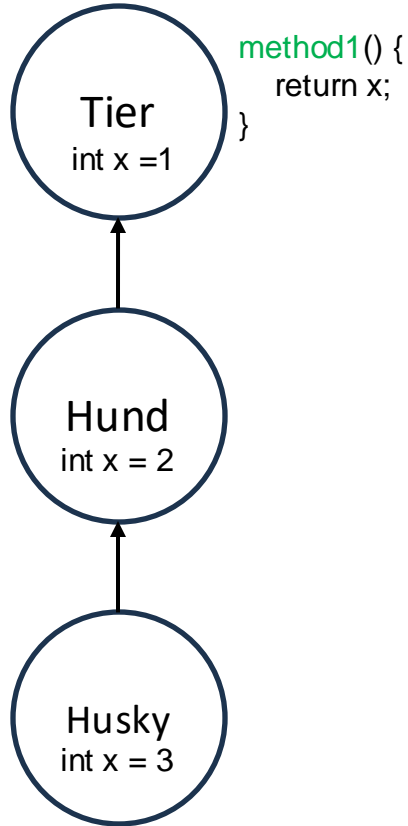
```
Tier t = new Husky();
```

```
System.out.println(t.method1());
```

Resultat: 1 ?

method1 existiert in der Husky Klasse nicht. Deshalb gehen wir durch alle Superklassen durch, bis wir eine solche Methode finden.

# Methoden:



**Regel:** Methoden\* werden an Hand vom **dynamischen Typ** ausgewählt.

\* ausser private, static und final Methoden (hier statischer Typ)

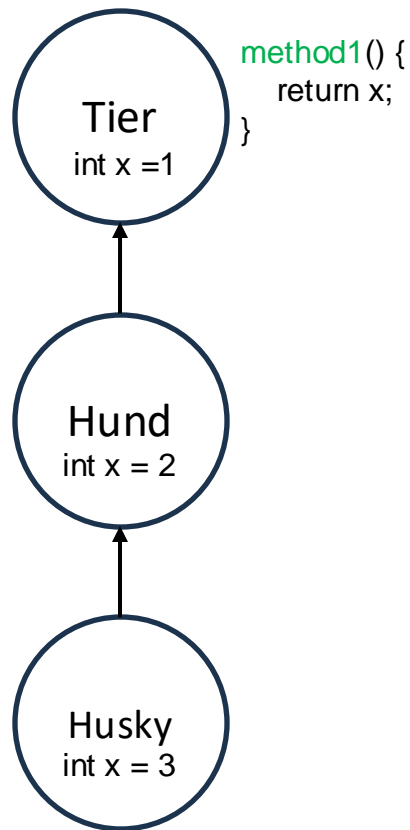
```
Tier t = new Husky();
```

```
System.out.println(t.method1());
```

Resultat: 1

method1 existiert in der Husky Klasse nicht. Deshalb gehen wir durch alle Superklassen durch, bis wir eine solche Methode finden.

# Methoden:



**Regel:** Methoden\* werden an Hand vom **dynamischen Typ** ausgewählt.

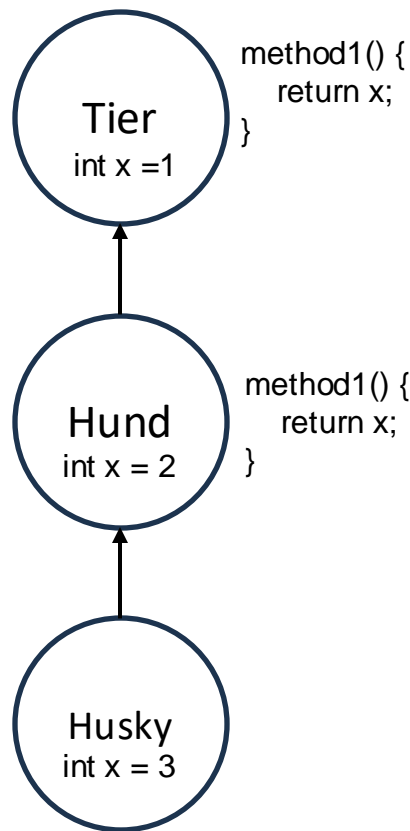
\* ausser private, static und final Methoden (hier statischer Typ)

`method1` existiert in der **Husky** Klasse nicht. Deshalb gehen wir durch alle Superklassen durch, bis wir eine solche Methode finden.

Das Attribut `x` wird weiterhin statisch ausgewählt.

- Beim Kompilieren wird bestimmt, dass falls `method1` in der Klasse **Tier** aufgerufen wird, dass wir **immer** das Attribut `x` aus der Klasse **Tier** wählen.

# Methoden:



**Regel:** Methoden\* werden an Hand vom **dynamischen Typ** ausgewählt.

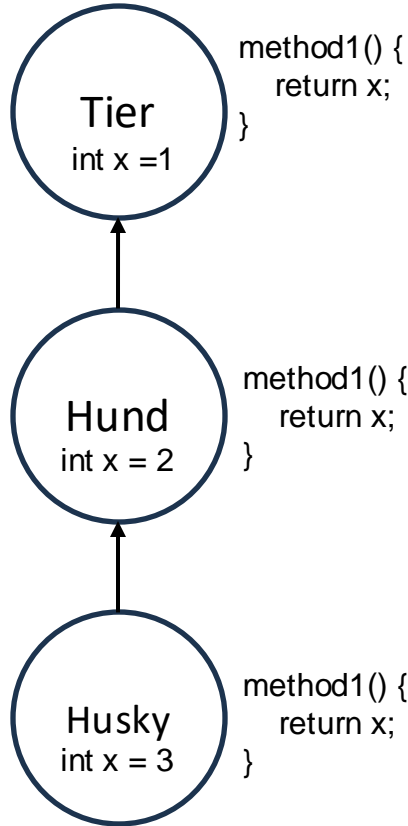
\* ausser private, static und final Methoden (hier statischer Typ)

```
Tier t = new Husky();
```

```
System.out.println(t.method1());
```

Resultat: 2

# Methoden:



**Regel:** Methoden\* werden an Hand vom **dynamischen Typ** ausgewählt.

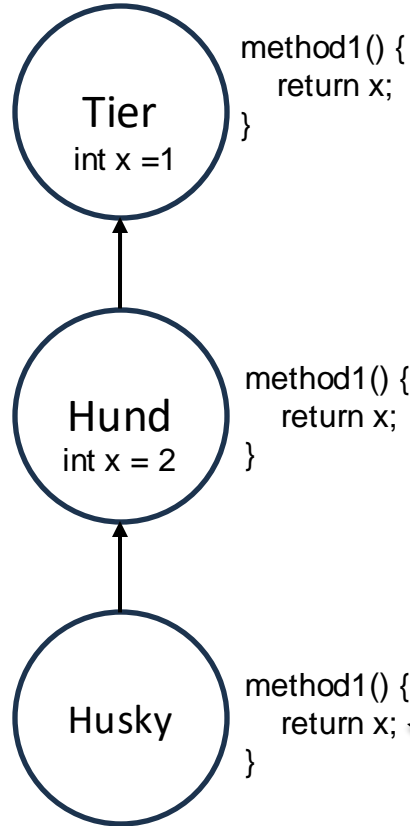
\* ausser private, static und final Methoden (hier statischer Typ)

```
Tier t = new Husky();
```

```
System.out.println(t.method1());
```

Resultat: 3

# Methoden:



**Regel:** Methoden\* werden an Hand vom **dynamischen Typ** ausgewählt.

\* ausser private, static und final Methoden (hier statischer Typ)

```
Tier t = new Husky();
```

```
System.out.println(t.method1());
```

Resultat: 2

Husky hat selbst kein Attribut x. Beim Kompilieren wird bestimmt, dass falls `method1` in der Klasse `Husky` aufgerufen wird, dass immer das Attribut `x` aus der Superklasse gewählt wird.



**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

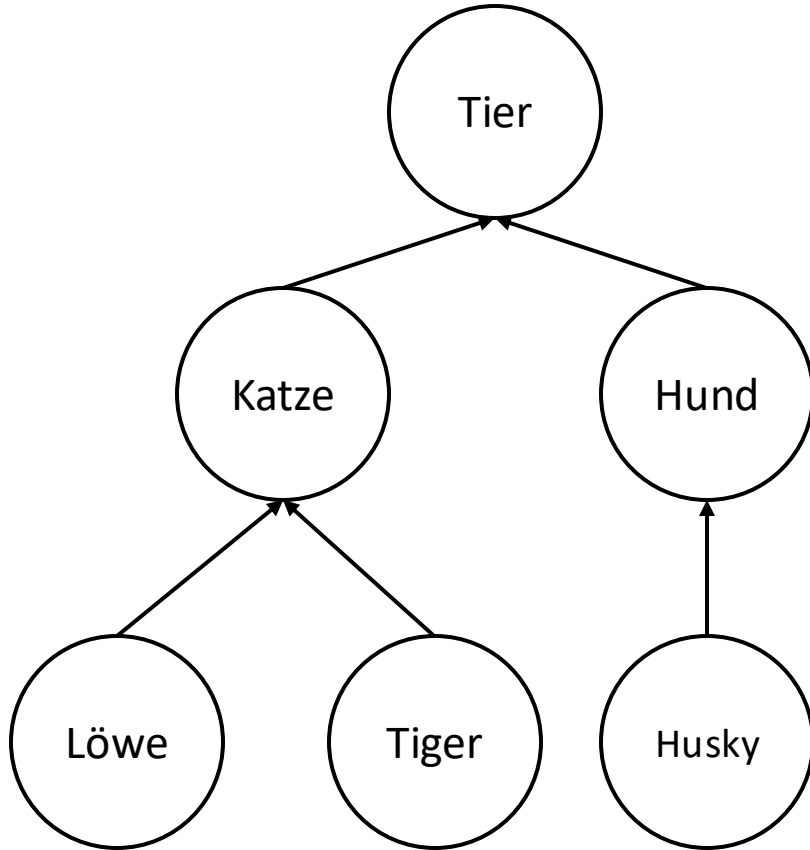
# super-Keyword

- `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

```

class Aaa {
    Integer s = 1;

    public void fct1() {
        System.out.println("Aaa " + s);
    }
}

class Mmm extends Aaa {
    String s = "Mmm";

    public void fct1() {
        System.out.println("Mmm fct1 " + s);
    }

    public void fct2() {
        System.out.println("Mmm fct2 " + s);
    }
}

class Nnn extends Aaa {
    String s = "Nnn";

    public void fct1() {
        System.out.println("Nnn fct1 " + s);
        fct3();
    }

    public void fct3() {
        System.out.println("Nnn fct3 " + s);
    }
}

```

```

class Bbb extends Nnn {
    String s = "Bbb";

    public void fct2() {
        System.out.println("Bbb fct2 " + s);
    }
}

class Ccc extends Nnn {
    String s = "Ccc";

    public void fct2() {
        super.fct1();
        System.out.println("Ccc fct2 " + s);
    }

    public void fct3() {
        System.out.println("Ccc fct3 " + s);
    }

    public String toString() {
        return s;
    }
}

class Ddd extends Ccc {
    String s = "Ddd";

    public void fct2() {
        System.out.println("Ddd fct2 " + s);
    }
}

```

```
Object ox = new Aaa();  
Object oy = (Aaa) ox;  
oy.fct1();
```

## Compile-Fehler

---

```
Aaa mx = new Mmm();  
mx.fct1();
```

## Mmm fct1 Mmm

---

```
Bbb bx = new Bbb();  
bx.fct2();
```

## Bbb fct2 Bbb

---

```
Ccc cz = new Ddd();  
cz.fct3();
```

**Ccc fct3 Ccc**

---

```
Ccc ca = new Ddd();  
((Aaa) ca).fct1();
```

**Nnn fct1 Nnn**  
**Ccc fct3 Ccc**

---

```
Ccc cx = new Ccc();  
Ddd dy = (Ddd) cx;  
dy.fct3();
```

**Exception**

---



```
Ddd dx = new Ddd();  
System.out.println(dx);
```

**Ccc**

---

**Kahoot**