252-0027

Einführung in die Programmierung

Übungen

Vererbung I

Henrik Pätzold
Departement Informatik
ETH Zürich

Heutiger Plan

- Theorie zu Invarianten (Schon wieder :/)
- Nachbesprechung TheoSim1 (mit Fokus auf Invarianten)
- Vererbung I

Invarianten (schon wieder)

Wie überprüfen wir eine Invariante

- Precondition ⇒ Invariante
- Gilt {Schleifenbedingung ∧ Invariante}
  SchleifenCode
  {Invariante}
- ¬Schleifenbedingung ∧ Invariante ⇒ Postcondition

```
public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = 0;
    res = b;

    // Loop Invariante:
    while (x < a) {
        res = res - 1;
        x = x + 1;
    }
    // Postcondition:  res == b - a
    return res;
}
```

```
public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = 0;
    res = b;

    // Loop Invariante: res == b - x
    while (x < a) {
        res = res - 1;
        x = x + 1;
    }
    // Postcondition:  res == b - a
    return res;
}
```

Precondition => Invariante

```
public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = 0;
    res = b;

    // Loop Invariante: res == b - x
    while (x < a) {
        res = res - 1;
        x = x + 1;
    }
    // Postcondition:  res == b - a
    return res;
}
```

```
public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = 0;
    res = b;

    // Loop Invariante: res == b - x
```

s

Gilt für a >= 0 immer res == b-x nach **s**?

Ja!

Wie überprüfen wir eine Invariante

- Precondition ⇒ Invariante 
- Gilt {Schleifenbedingung ∧ Invariante}
  SchleifenCode
  {Invariante}
- ¬Schleifenbedingung ∧ Invariante ⇒ Postcondition

```
public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = 0;
    res = b;

    // Loop Invariante: res == b - x
    while (x < a) {
        res = res - 1;
        x = x + 1;
    }
    // Postcondition:  res == b - a
    return res;
}
```

```
while (x < a) {
    res = res - 1;
    x = x + 1;
}
```

{Schleifenbedingung && Invariante}

```
    res = res - 1;
    x = x + 1;
```

{Invariante}

{x < a && res == b - x}

```
    res = res - 1;
    x = x + 1;
```

{res == b - x}

{ **}**

```
    res = res - 1;

    x = x + 1;
```

{res == b - x}

{ **}**

```
    res = res - 1;
```
{res == b - (x + 1)}
```
    x = x + 1;
```
{res == b - x}

{ }

{res - 1 == b - x - 1}

```
res = res - 1;
```

{res == b - (x + 1)}

```
x = x + 1;
```

{res == b - x}

{ }

{res == b - x}

{res - 1 == b - x - 1}

```
res = res - 1;
```

{res == b - (x + 1)}

```
x = x + 1;
```

{res == b - x}

```
while (x < a) {
 {                                    }
```

{res == b - x}

{res - 1 == b - x - 1}

```
res = res - 1;
```

{res == b - (x + 1)}

```
x = x + 1;
```

{res == b - x}

```
}
```

```
while (x < a) {
 {x < a                                    }
      {res == b - x}
      {res - 1 == b - x - 1}
      res = res - 1;
      {res == b - (x + 1)}
      x = x + 1;
 {res == b - x}
}
```

```
while (x < a) {
 {x < a  && res == b - x }
      {res == b - x}
      {res - 1 == b - x - 1}
      res = res - 1;
      {res == b - (x + 1)}
      x = x + 1;
 {res == b - x}
}
```

```
while (x < a) {
```
{x < a && res == b - x } **= {Schleifenbedingung && Invariante}**

{res == b - x}

{res - 1 == b - x - 1}

```
res = res - 1;
```

{res == b - (x + 1)}

```
x = x + 1;
```

{res == b - x} **= {Invariante}**

```
}
```

Wie überprüfen wir eine Invariante

- Precondition ⇒ Invariante 
- Gilt {Schleifenbedingung ∧ Invariante}
  SchleifenCode
  {Invariante} 
- ¬Schleifenbedingung ∧ Invariante ⇒ Postcondition

$$\neg \text{Schleifenbedingung} \land \text{Invariante} \Rightarrow \text{Postcondition}$$

$$\neg(x<a) \land res == b - x$$

$$\neg(x<a) \land res == b - x$$
$$\equiv x \geq a \land res == b - x$$

$$x \geq a \land res == b - x \Rightarrow res == b - a$$

Hält nicht für x > a

Wie überprüfen wir eine Invariante

- Precondition ⇒ Invariante 
- Gilt {Schleifenbedingung ∧ Invariante}
  SchleifenCode
  {Invariante} 
- ¬Schleifenbedingung ∧ Invariante ⇒ Postcondition 

$$x \geq a \land res == b - x \Rightarrow res == b - a$$

Hält nicht für $x > a$

Damit die Implikation hält, muss $x == a$ gelten

Wir erweitern die Invariante:

res == b - x

Wir erweitern die Invariante:

$$res == b - x \land x \leq a$$

$$\neg \text{Schleifenbedingung} \land \text{Invariante} \Rightarrow \text{Postcondition}$$

$$\neg(x < a) \land res == b - x \land x \leq a \Rightarrow res == b - a$$

$$x \geq a \land res == b - x \land x \leq a \Rightarrow res == b - a$$

$$x == a \land res == b - x \Rightarrow res == b - a$$

`res == b - a ⇒ res == b - a`

Hält trivialerweise

Wie überprüfen wir eine Invariante

- Precondition ⇒ Invariante 
- Gilt {Schleifenbedingung ∧ Invariante}
  SchleifenCode
  {Invariante} 
- ¬Schleifenbedingung ∧ Invariante ⇒ Postcondition 

res == b - x ∧ x ≤ a ist eine gültige Invariante!

a)
```
P: { x >= 0 }
int i = 0;
int s = 0;
while(i <= x) {
  s = s + i;
  i = i + 1;
}
Q: { s == x*(x+1)/2 }
```

Wählen Sie die passende Option:

- ☐ `{i <= x && s == (i-1)*i/2}`
- ☐ `{i <= x && s == i*(i+1)/2}`
- ☐ `{i <= x + 1 && s == (i-1)*i/2}`
- ☐ Keine der drei Optionen ist eine gültige Invariante

a)
```
P: { x >= 0 }
int i = 0;
int s = 0;
while(i <= x) {
  s = s + i;
  i = i + 1;
}
Q: { s == x*(x+1)/2 }
```

Wählen Sie die passende Option:

- ☐ {i <= x && s == (i-1)*i/2}
- ☐ {i <= x && s == i*(i+1)/2}
- ☐ {i <= x + 1 && s == (i-1)*i/2}
- ☐ Keine der drei Optionen ist eine gültige Invariante

- Die ersten beiden verletzen Bedingung 2:
  - Für i == x haben wir

```
{i <= x && i <= x && s == i*(i+1)/2}
  s = s + i;
  i = i + 1;
{i > x && s == i*(i+1)/2}
```

a)
```
P: { x >= 0 }
int i = 0;
int s = 0;
while(i <= x) {
  s = s + i;
  i = i + 1;
}
Q: { s == x*(x+1)/2 }
```

- Bedingung 1:
  - Für beliebige x >= 0 haben wir vor der ersten Iteration immer s = i = 0
  - Damit gilt i <= x + 1 && s == (i-1)*i/2

Wählen Sie die passende Option:

- ☐ {i <= x && s == (i-1)*i/2}
- ☐ {i <= x && s == i*(i+1)/2}
- ☐ {i <= x + 1 && s == (i-1)*i/2}
- ☐ Keine der drei Optionen ist eine gültige Invariante

a)
```
P: { x >= 0 }
int i = 0;
int s = 0;
while(i <= x) {
  s = s + i;
  i = i + 1;
}
Q: { s == x*(x+1)/2 }
```

Wählen Sie die passende Option:

- ☐ {i <= x && s == (i-1)*i/2}
- ☐ {i <= x && s == i*(i+1)/2}
- ☐ {i <= x + 1 && s == (i-1)*i/2}
- ☐ Keine der drei Optionen ist eine gültige Invariante

- Bedingung 1:
  - Für beliebige x >= 0 haben wir vor der ersten Iteration immer s = i = 0
  - Damit gilt i <= x + 1 && s == (i-1)*i/2
- Bedingung 2:

```
while(i <= x) {
```

{i <= x + 1 && s == (i-1) * i/2}

{i <= x + 1 && s == (i^2 - i)/2}

{i <= x + 1 && s == (i^2 + i)/2 - i}

{i <= x + 1 && s + i == (i^2 + i)/2}

{i <= x + 1 && s + i == i*(i+1)/2}

{i+1 <= x + 1 && s + i == ((i+1)-1)*(i+1)/2}

```
s = s + i;
```

{i+1 <= x + 1 && s == ((i+1)-1)*(i+1)/2}

```
i = i + 1;
```

{i <= x + 1 && s == (i-1)*i/2}

```
}
```

a)
```
P: { x >= 0 }
int i = 0;
int s = 0;
while(i <= x) {
  s = s + i;
  i = i + 1;
}
Q: { s == x*(x+1)/2 }
```

Wählen Sie die passende Option:

- ☐ {i <= x && s == (i-1)*i/2}
- ☐ {i <= x && s == i*(i+1)/2}
- ☐ {i <= x + 1 && s == (i-1)*i/2}
- ☐ Keine der drei Optionen ist eine gültige Invariante

- Bedingung 1:
  - Für beliebige x >= 0 haben wir vor der ersten Iteration immer s = i = 0
  - Damit gilt i <= x + 1 && s == (i-1)*i/2
- Bedingung 2: hält

$$\neg(i<=x) \land i<=x+1 \land s==(i-1)*i/2$$

$$i>x \wedge i<=x+1 \wedge s==(i-1)*i/2$$

$$i==x+1 \land s==(i-1)*i/2 \Rightarrow s == x*(x+1)/2$$

$$s==x*(x+1)/2 \Rightarrow s == x*(x+1)/2$$

a)
```
P: { x >= 0 }
int i = 0;
int s = 0;
while(i <= x) {
  s = s + i;
  i = i + 1;
}
Q: { s == x*(x+1)/2 }
```

Wählen Sie die passende Option:

- ☐ {i <= x && s == (i-1)*i/2}
- ☐ {i <= x && s == i*(i+1)/2}
- ☑ {i <= x + 1 && s == (i-1)*i/2}
- ☐ Keine der drei Optionen ist eine gültige Invariante

- Bedingung 1:
  - Für beliebige x >= 0 haben wir vor der ersten Iteration immer s = i = 0
  - Damit gilt i <= x + 1 && s == (i-1)*i/2
- Bedingung 2: hält
- Bedingung 3: hält

b)
```
P: { n >= 0 }
int y = 0;
int z = 0;
while(y*y < n) {
  y = y + 1;
  if (y*y <= n) {
    z = z + 1;
  }
}
Q: {z == Math.floor(Math.sqrt(n))}
```

Wählen Sie die passende Option:

- ☐ `{y*y <= n && z*z <= n}`
- ☐ `{y*y <= n && (y == z || y == z+1)}`
- ☐ `{z*z <= n && (y == z || y == z+1)}`
- ☐ Keine der drei Optionen ist eine gültige Invariante

b)
```
P: { n >= 0 }
int y = 0;
int z = 0;
while(y*y < n) {
  y = y + 1;
  if (y*y <= n) {
    z = z + 1;
  }
}
Q: {z == Math.floor(Math.sqrt(n))}
```

Wählen Sie die passende Option:

- ☐ `{y*y <= n && z*z <= n}`
- ☐ `{y*y <= n && (y == z || y == z+1)}`
- ☐ `{z*z <= n && (y == z || y == z+1)}`
- ☐ Keine der drei Optionen ist eine gültige Invariante

Wählen wir n == 3, so gilt nach Ausführung der 3. Loop-Iteration für y = 2 y*y == 4 (Dies verletzt Bedingung 2). Dies schließt die ersten beiden Optionen aus.

```
!(y*y<n) && z*z <= n && (y == z || y == z+1)
≡ y*y>=n && z*z <= n && (y == z || y == z+1)
≡ y*y>=n && z*z <= n && (y == z || y == z+1)
≡ (y*y>=n && z*z <= n && y == z) || (y*y>=n && z*z <= && n y == z+1)
≡ (z*z>=n && z*z <= n) || (y*y>=n && z*z <= n && y == z+1)
≡ (z*z==n) || ((z+1)*(z+1)>=n && z*z <= n && y == z+1)
≡ (z*z==n) || ((z+1)*(z+1)>=n && z*z <= n)
≡  (z*z==n) || ((z+1)*(z+1)>=n && z*z <= n)
```

```
(z*z==n) || ((z+1)*(z+1)>=n && z*z <= n)
```

$$\Rightarrow$$ `z == Math.floor(Math.sqrt(n))`

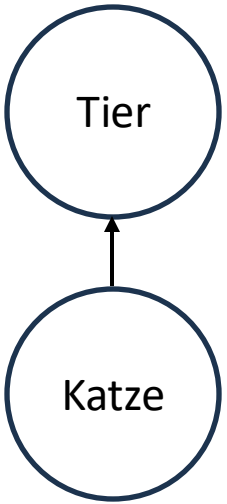
Gilt nicht für n=25, z = 4, y = 5

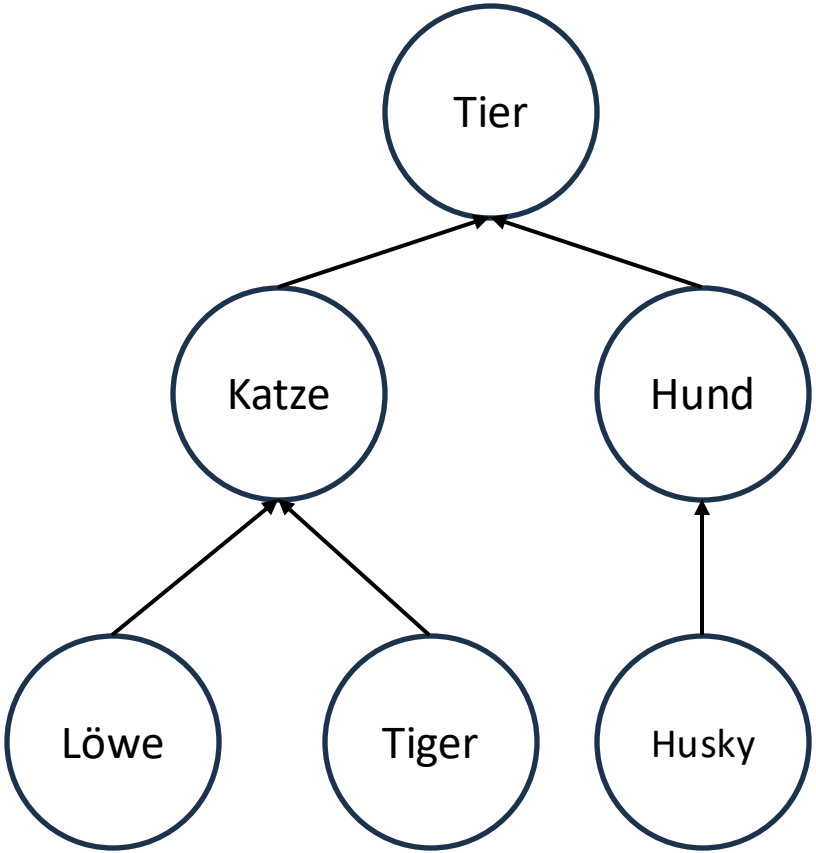
Vererbung

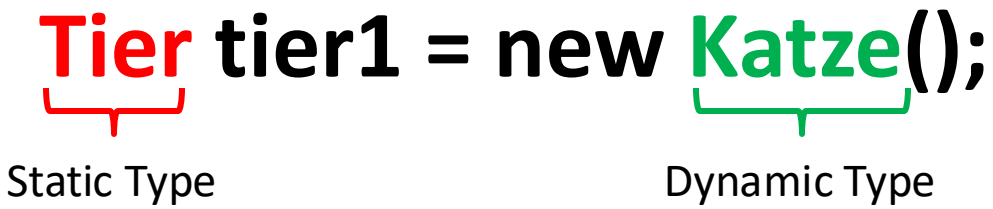
Extends-Schlüsselwort

- **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{}
```




Tier
Katze
Hund
Löwe
Tiger
Husky

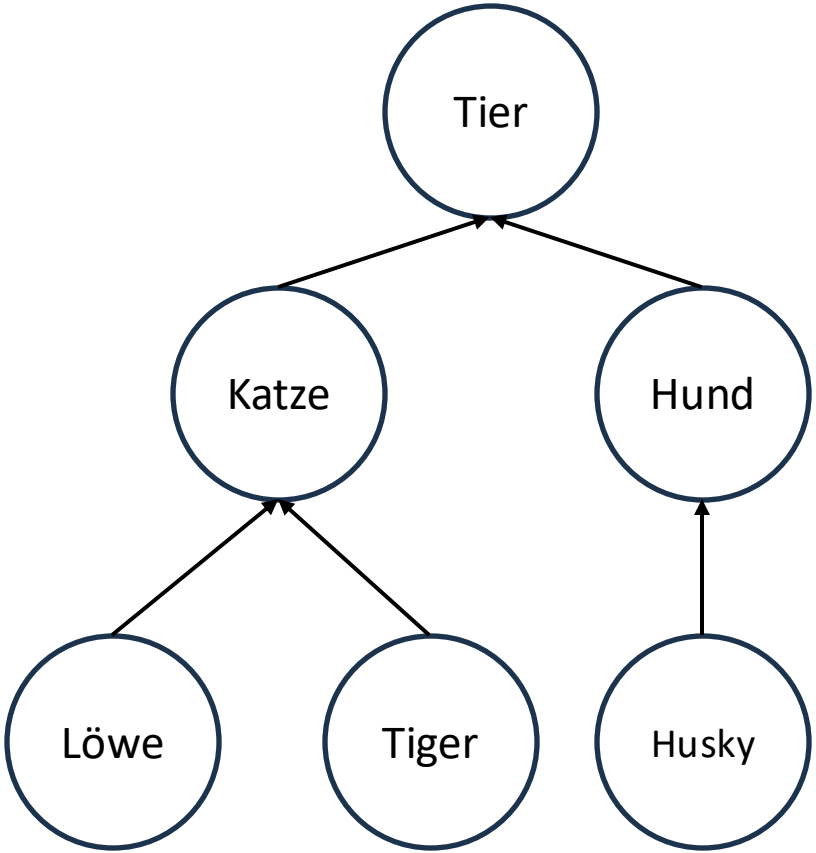

Tier tier1 = new Katze();
Static Type
Dynamic Type

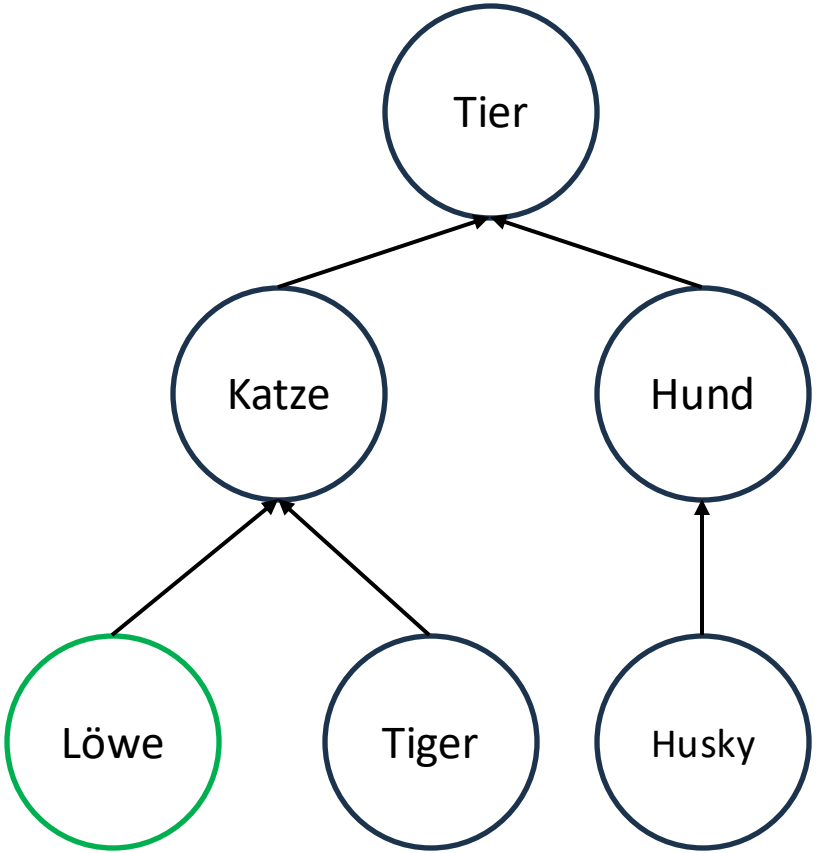
Static-Type

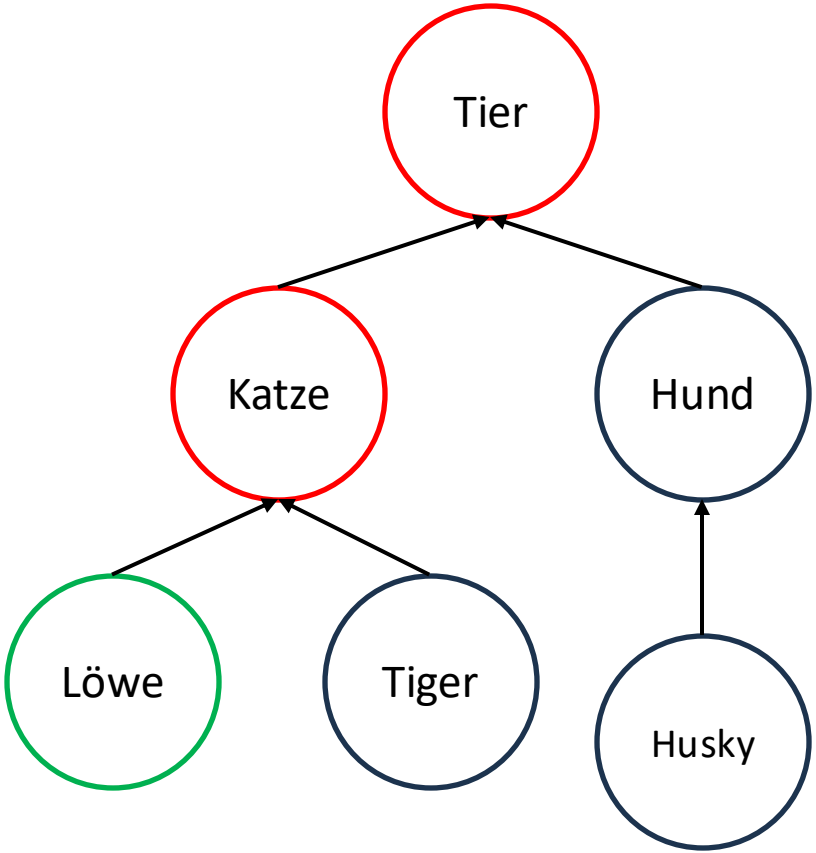
- Wird zur Deklarationszeit angegeben und durch den Compiler überprüft
- Der Compiler verwendet den statischen Typ, um sicherzustellen, dass nur Methoden und Attribute des deklarierten Typs aufgerufen werden können.
- Bleibt während „Lebensdauer“ der Variable gleich

Dynamic-Type

- ist der Typ des Objekts, auf das die Variable zur Laufzeit verweist.
- Der dynamische Typ kann sich ändern, indem die Referenz auf ein anderes Objekt gesetzt wird.
- Der dynamic Type muss ein Subtyp oder der gleiche Typ wie der static Typ sein (merken)


Tier
Katze
Hund
Löwe
Tiger
Husky


Tier
Katze
Hund
Löwe
Tiger
Husky

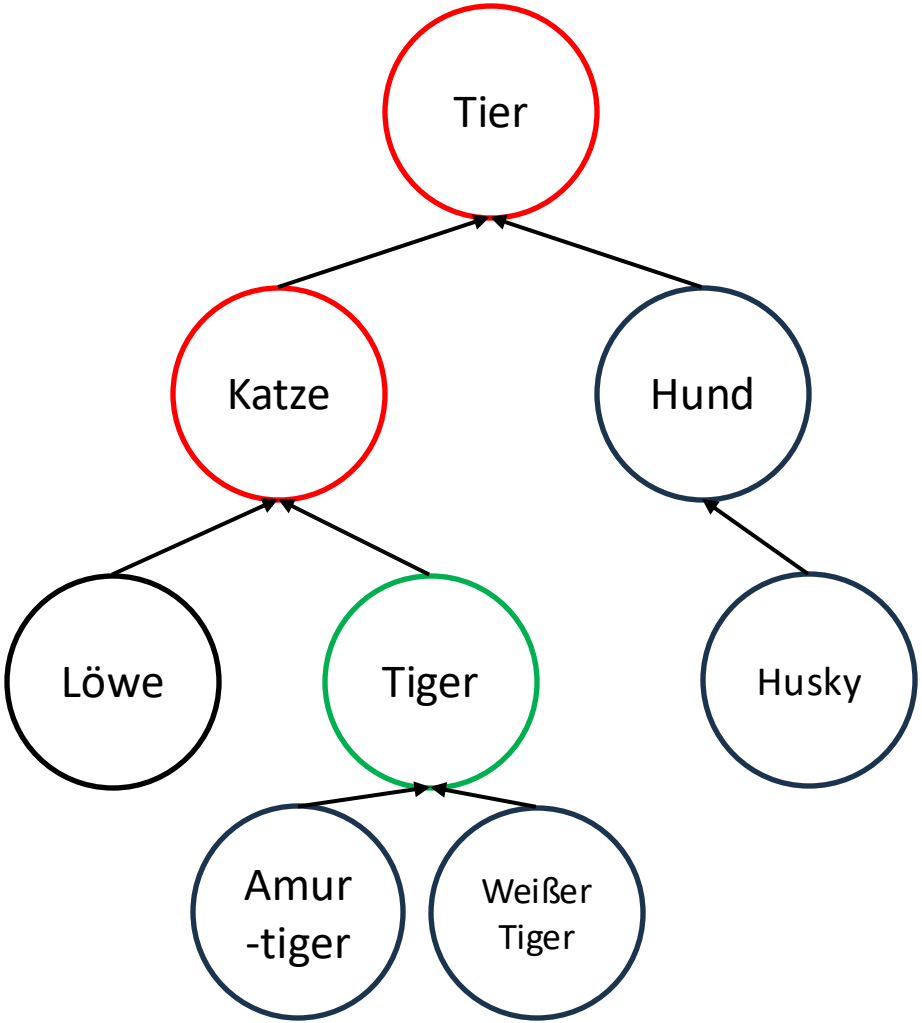


Möglicher Static type

Dynamic type

Casting bei Vererbung

- Wir unterscheiden zwischen **upcasts** und **downcasts**
- **upcasts** sind implizit – wir können so weit nach oben casten wie wir wollen, ohne den cast spezifisch angeben zu müssen **(Slide)**
- **downcasts** sind explizit – wir müssen den cast spezifisch angeben und können auf alle Superklassen (inklusive) der eigenen **downcasten (Slide)**
  - Diese gelten als unsicherer und werden zur Laufzeit geprüft - **ClassCastException**
- **quercasts** sind **immer** Illegal **(Slide)**
  - Der Compiler erkennt das sofort - **Compiler Fehler**
- Alle drei sind sehr beliebt an Theorieprüfungen



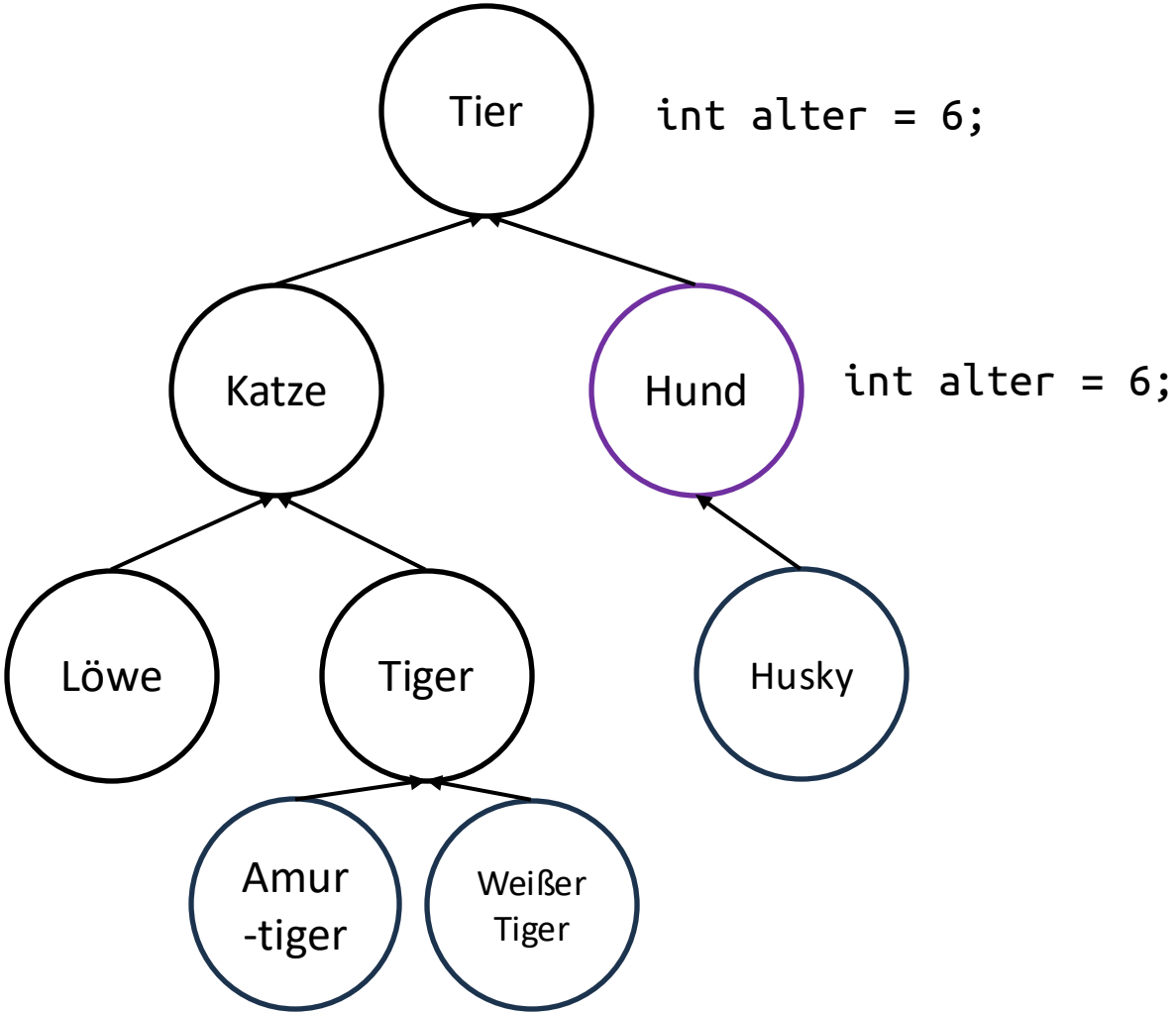
Möglicher Static type

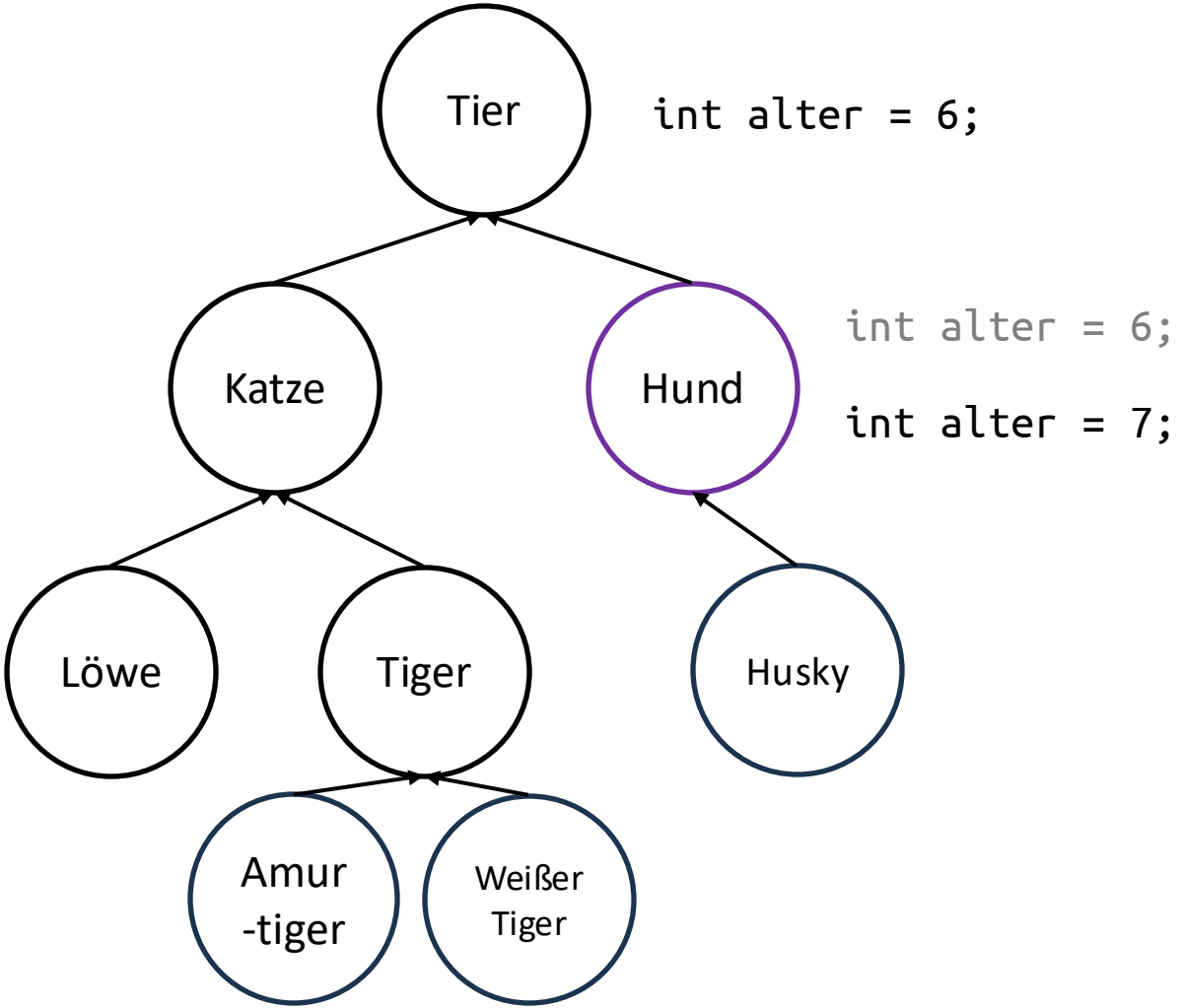
Dynamic type

Showcase

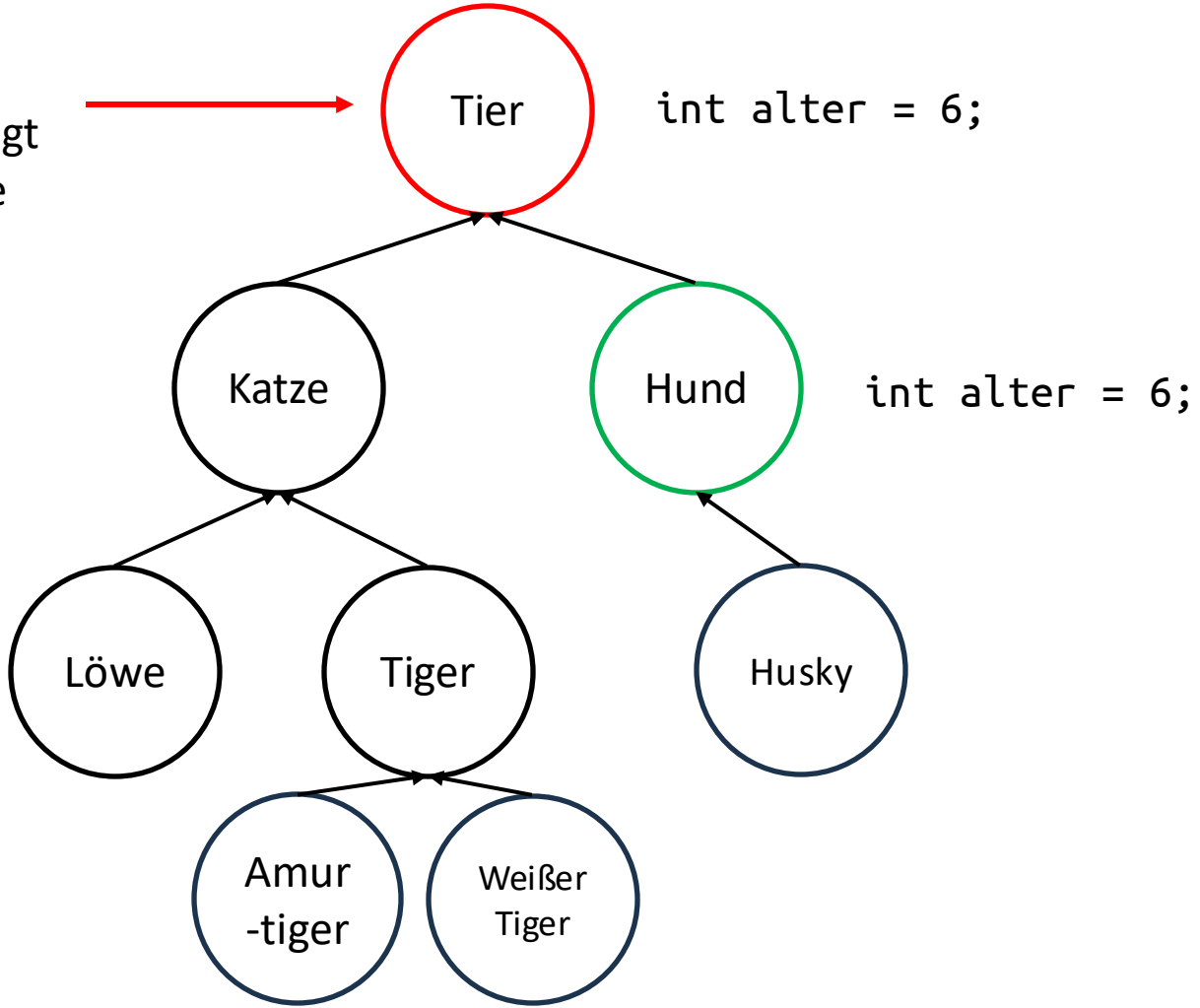
Vererbung - Attribute

- Wenn eine Subklasse von einer Superklasse erbt, stehen ihr alle nicht-privaten Attribute der Superklasse zur Verfügung.
- können direkt verwendet oder von Subklassen initialisiert werden.
- Wenn in einer Subklasse ein Attribut mit dem gleichen Namen wie in der Superklasse deklariert wird, wird das Attribut der Superklasse versteckt.
- Zugriff auf das versteckte Attribut ist weiterhin möglich, jedoch nur über die Superklassen-Referenz.
- Bei Zugriff auf das Attribut einer variable entscheidet der **static Type**, auf welche Klasse zugegriffen wird **(Wichtig)**


Tier
int alter = 6;
Katze
Hund
int alter = 6;
Löwe
Tiger
Husky
Amur
-tiger
Weißer
Tiger


Tier
int alter = 6;
Katze
Hund
int alter = 6;
int alter = 7;
Löwe
Tiger
Husky
Amur
-tiger
Weißer
Tiger

Zugriff (egal, ob lesen oder schreiben) erfolgt beim static type

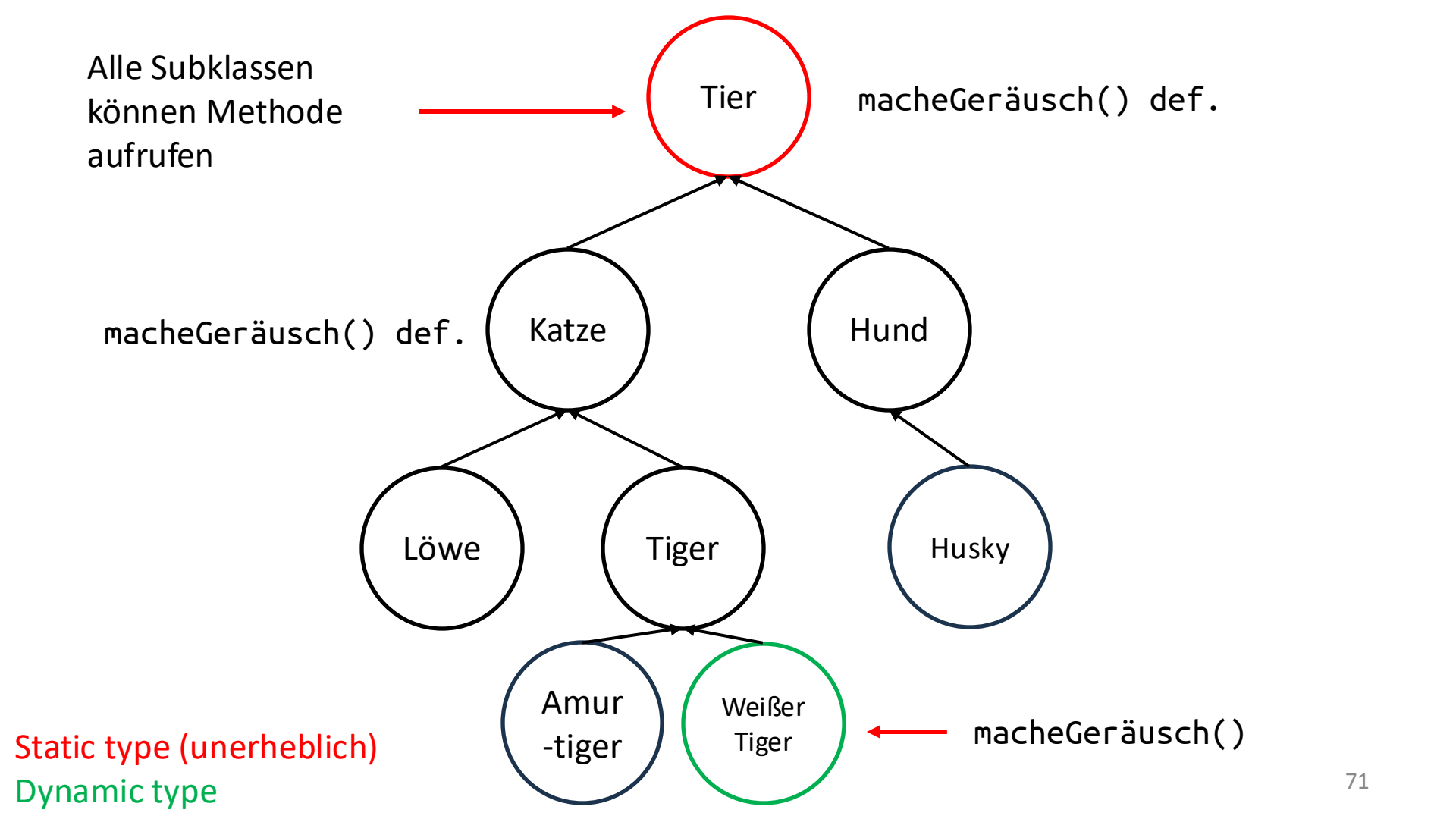


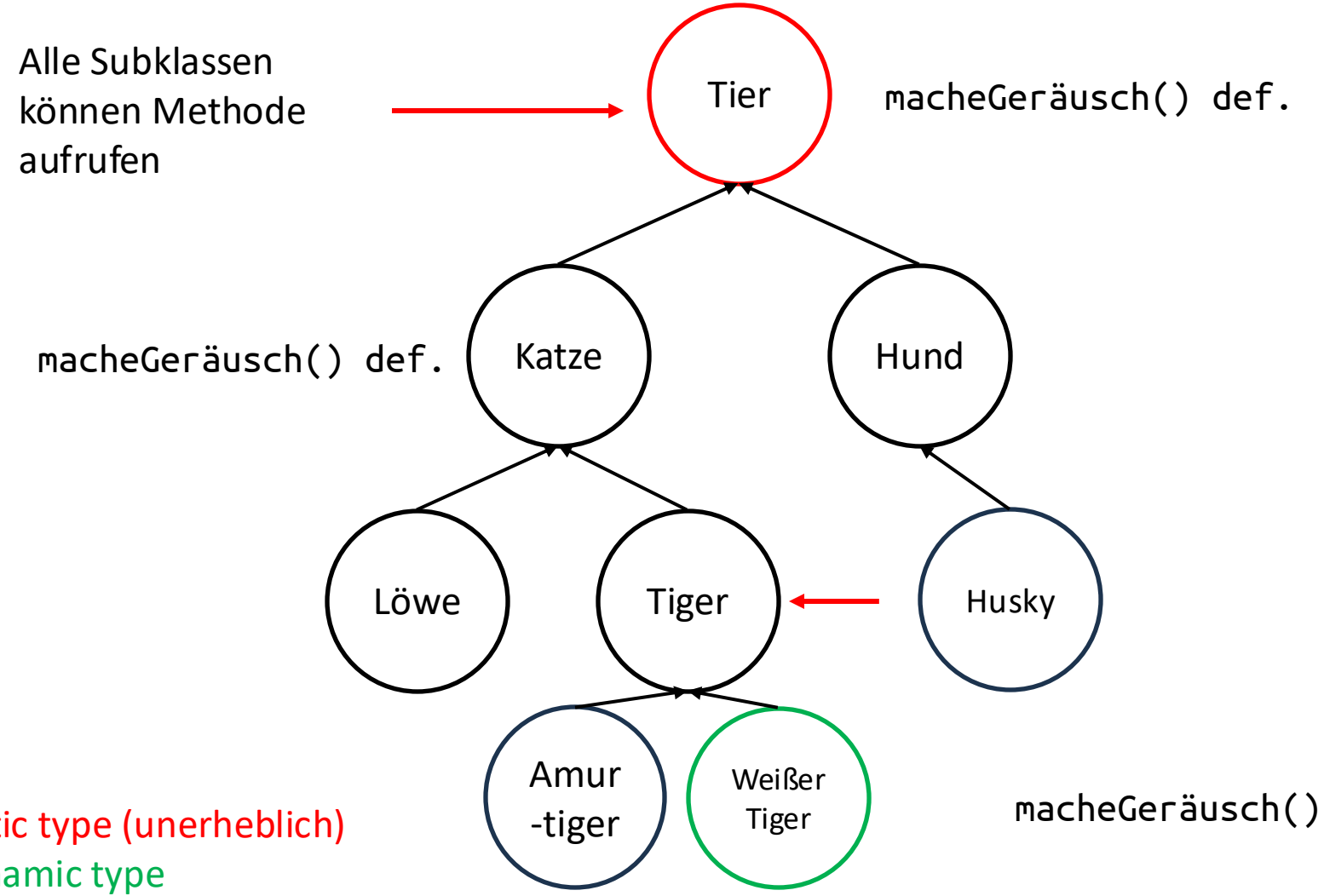
Static type
Dynamic type

Showcase

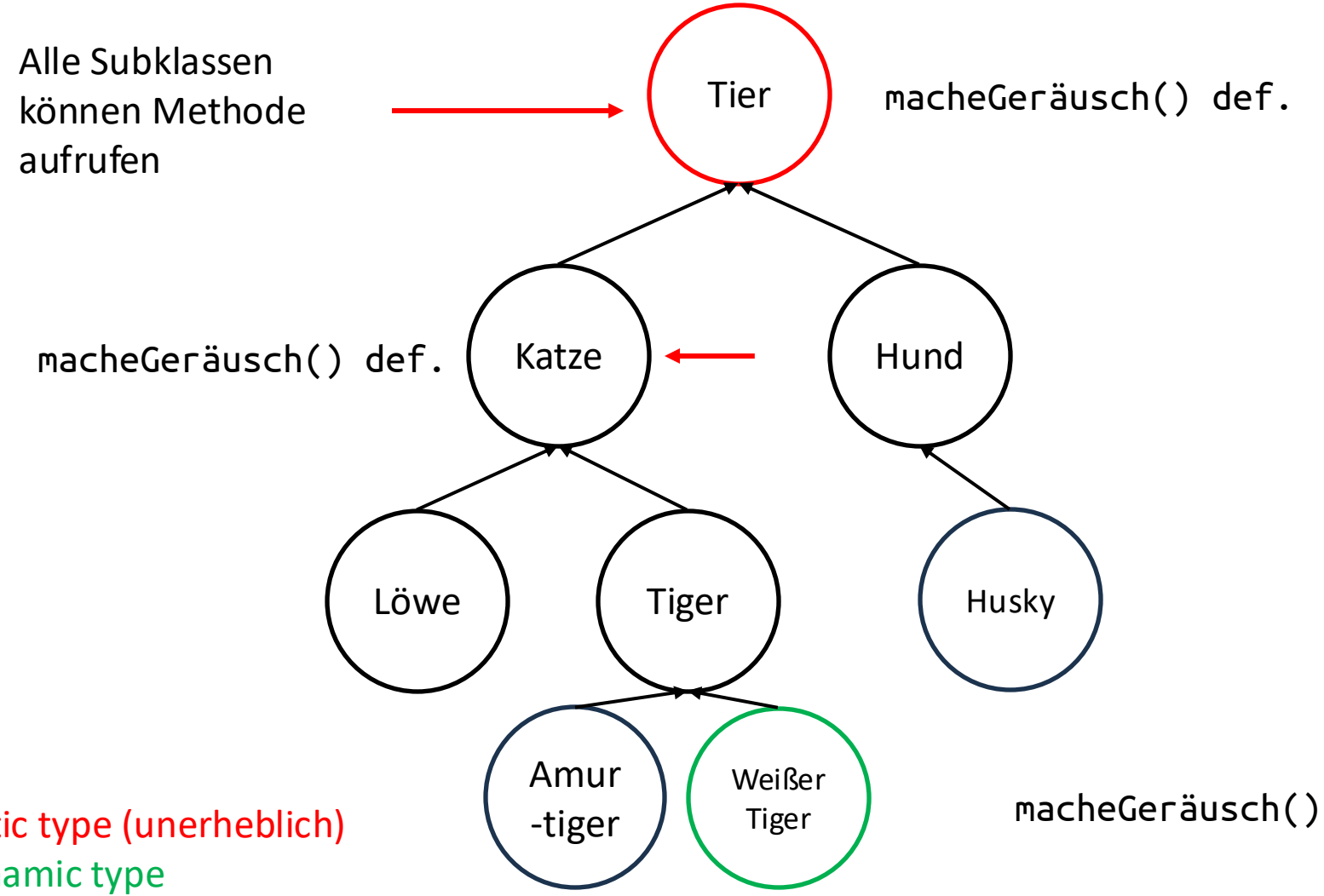
Vererbung - Methoden

- Methoden werden vererbt:
  - Subklassen erben alle nicht privaten Methoden ihrer Superklassen.
  - Beim Aufrufen entscheidet der dynamische Typ des Objekts zur Laufzeit, welche Methode aufgerufen wird, beginnend bei der konkreten Klasse und aufsteigend in der Hierarchie.
- Dynamic Binding: **(Wichtig)**
  - Wenn keine überschreibende Methode vorhanden ist, wird die Implementierung in der nächstgelegenen Superklasse ausgeführt.
  - Wird eine Methode nicht in der Subklasse gefunden, sucht Java automatisch in der nächstgelegenen Superklasse nach der Methode.


Alle Subklassen können Methode aufrufen
Tier
macheGeräusch() def.
macheGeräusch() def.
Katze
Hund
Löwe
Tiger
Husky
Amur -tiger
Weißer Tiger
macheGeräusch()
Static type (unerheblich)
Dynamic type



Static type (unerheblich)
Dynamic type


Alle Subklassen können Methode aufrufen
Tier
macheGeräusch() def.
macheGeräusch() def.
Katze
Hund
Löwe
Tiger
Husky
Amur -tiger
Weißer Tiger
macheGeräusch()
Static type (unerheblich)
Dynamic type

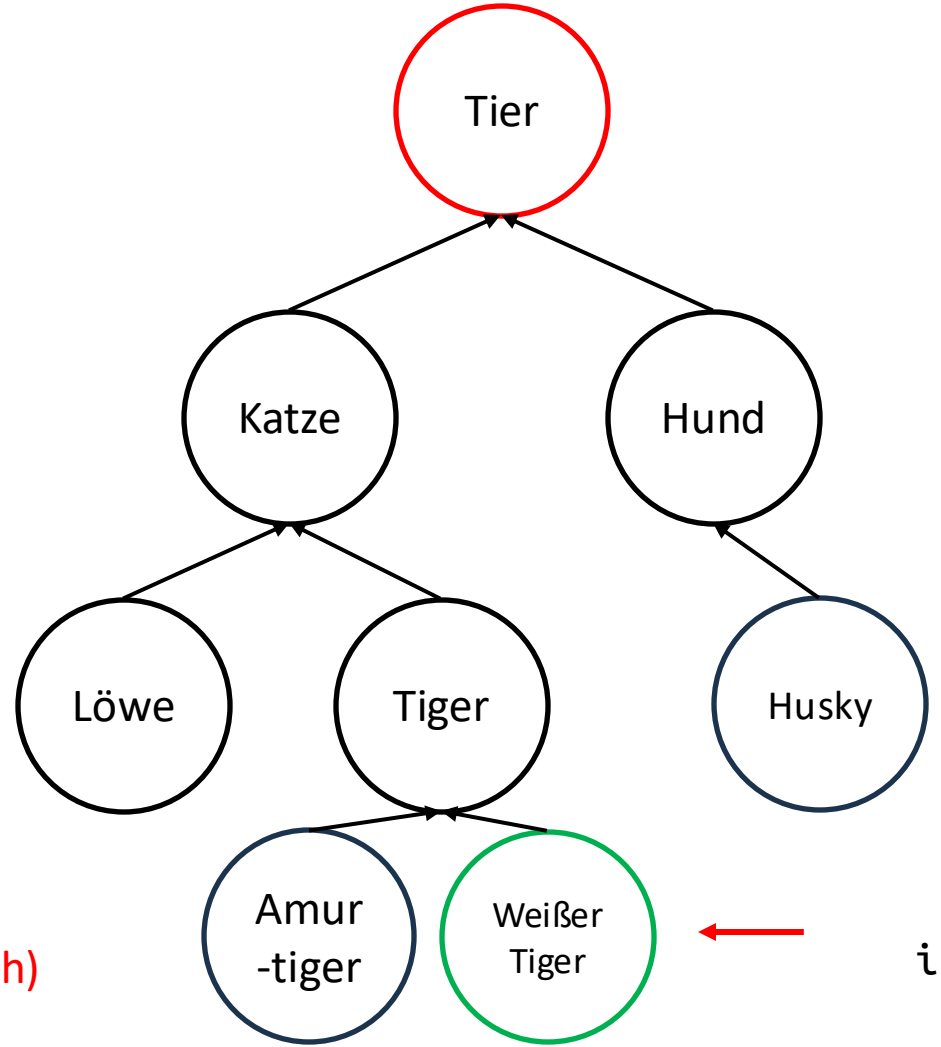
Showcase

Vererbung - Methoden

- Methoden werden vererbt:
  - Subklassen erben alle nicht privaten Methoden ihrer Superklassen.
  - Beim Aufrufen entscheidet der dynamische Typ des Objekts zur Laufzeit, welche Methode aufgerufen wird, beginnend bei der konkreten Klasse und aufsteigend in der Hierarchie.
- Dynamic Binding: **(Wichtig)**
  - Wenn keine überschreibende Methode vorhanden ist, wird die Implementierung in der nächstgelegenen Superklasse ausgeführt.
  - Wird eine Methode nicht in der Subklasse gefunden, sucht Java automatisch in der nächstgelegenen Superklasse nach der Methode.
  - Es werden beim Aufruf einer Methode die Attribute der aufrufenden Klasse verwendet

Konstruktoren

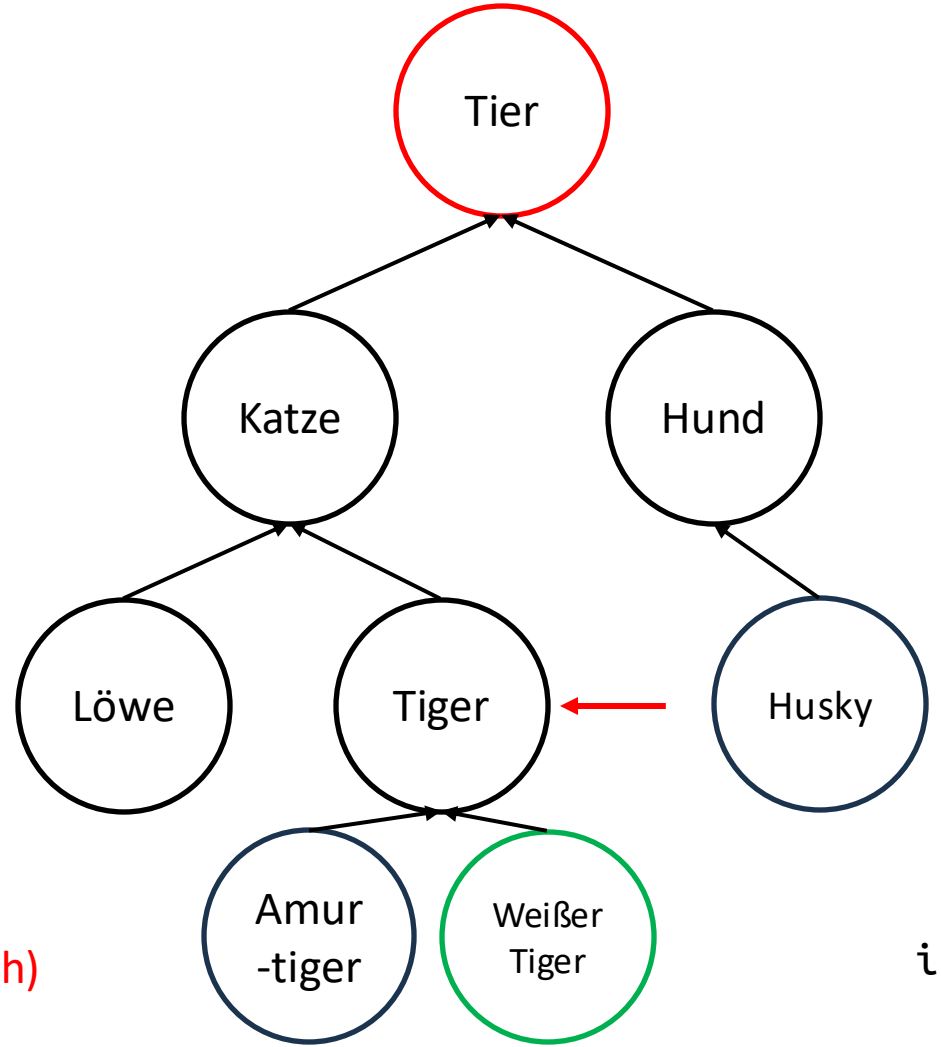
- Impliziter Aufruf des parameterlosen Konstruktors der Superklasse:
  - Wenn in der Subklasse kein expliziter super(...)-Aufruf erfolgt, wird automatisch der parameterlose Konstruktor der Superklasse aufgerufen.
- Expliziter Aufruf mit super(...):
  - Wenn die Superklasse keinen parameterlosen Konstruktor hat, muss die Subklasse super(...) mit passenden Parametern aufgerufen werden.
- Warum rufen wir den Konstruktor der Superklasse auf?
  - Attribute und Logik der Superklassen korrekt initialisiert, bevor Subklasse eigene Initialisierungen durchführt
  - Verhindert unvollständig initialisierte Objekte



Static type (unerheblich)

Dynamic type

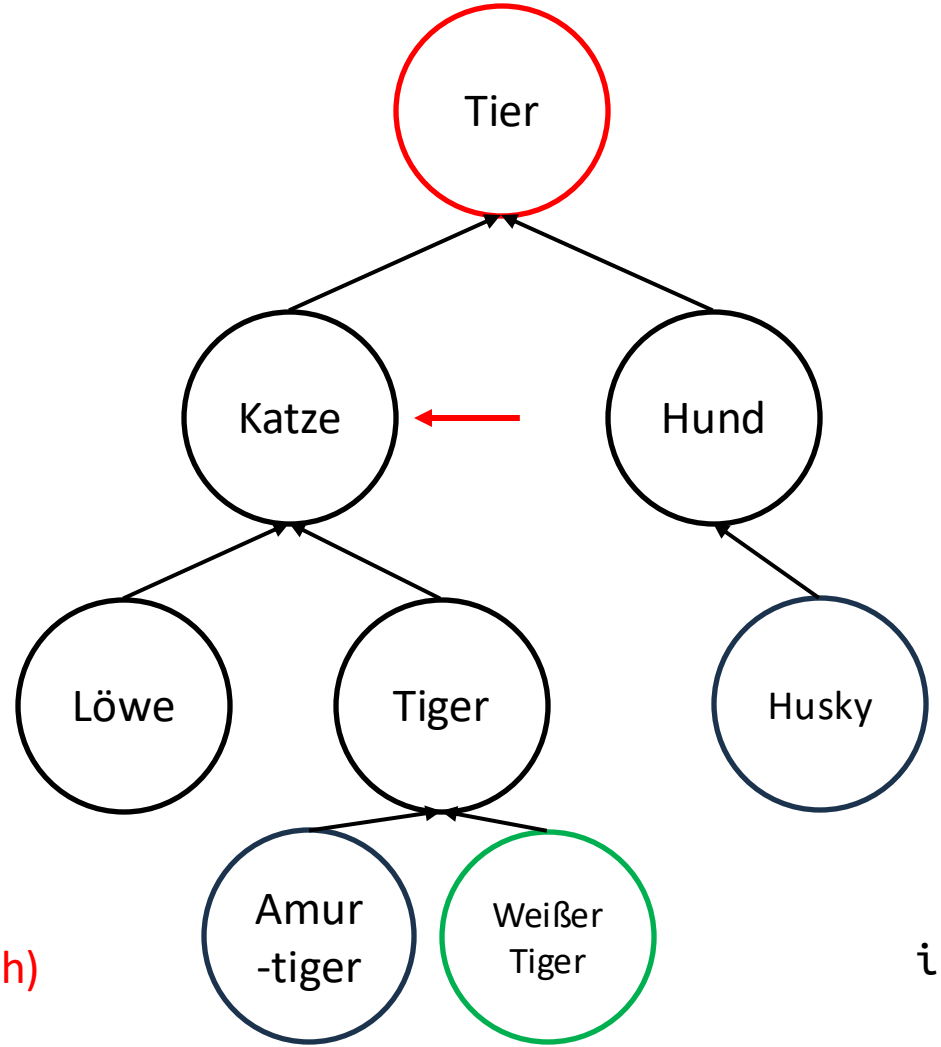
```
init.
```



Static type (unerheblich)

Dynamic type

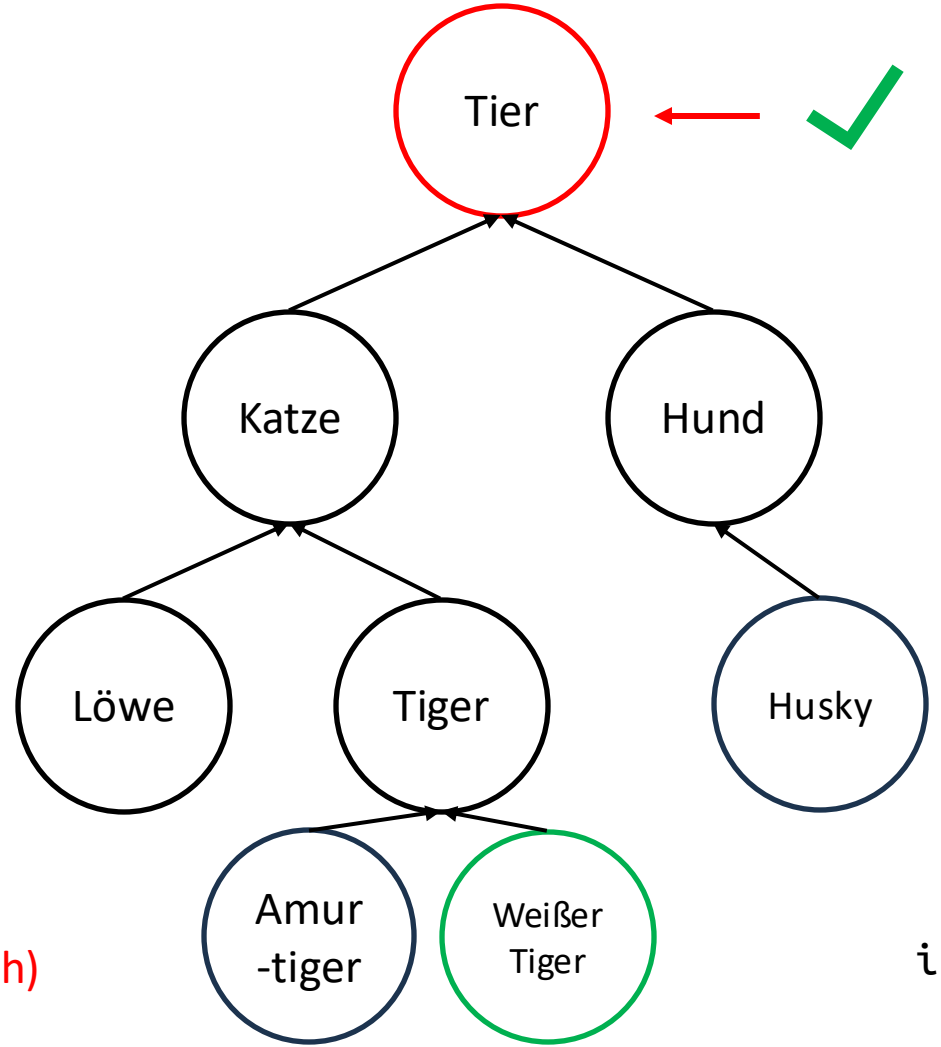
```
init.
```



Static type (unerheblich)

Dynamic type

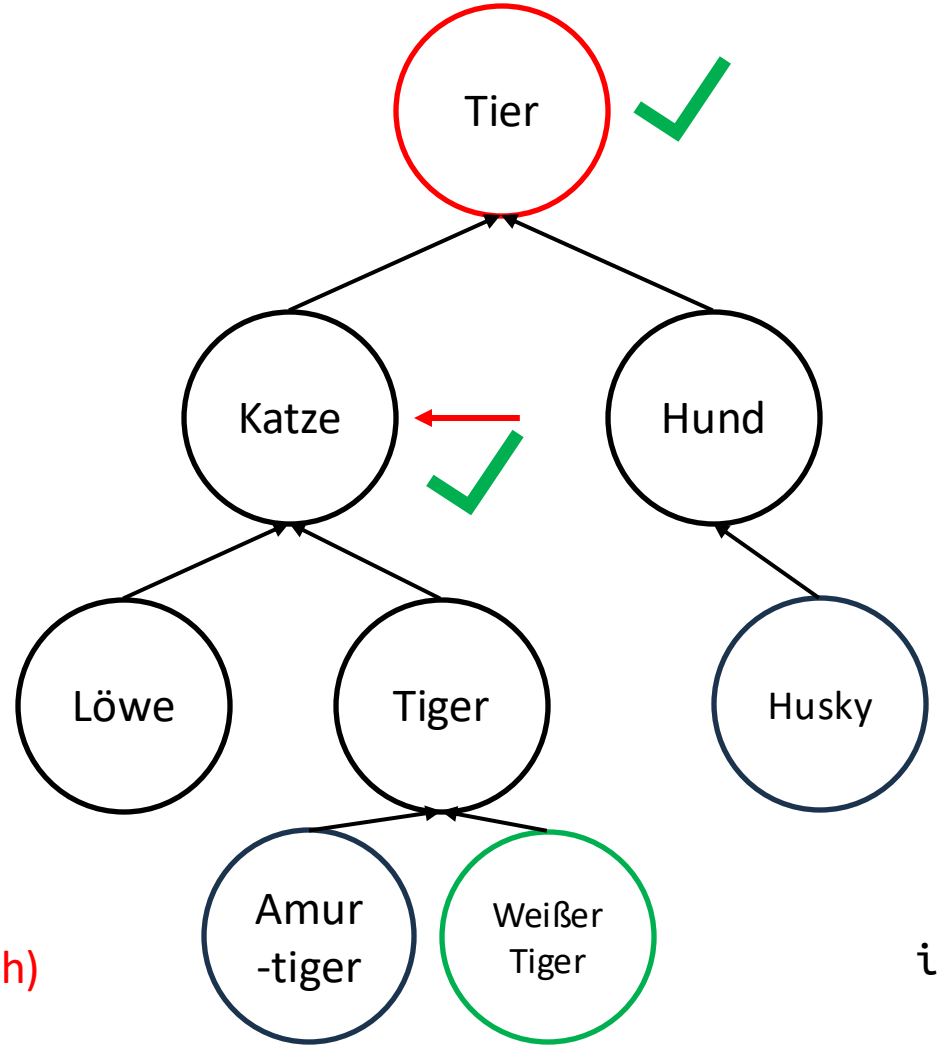
```
init.
```



Static type (unerheblich)

Dynamic type

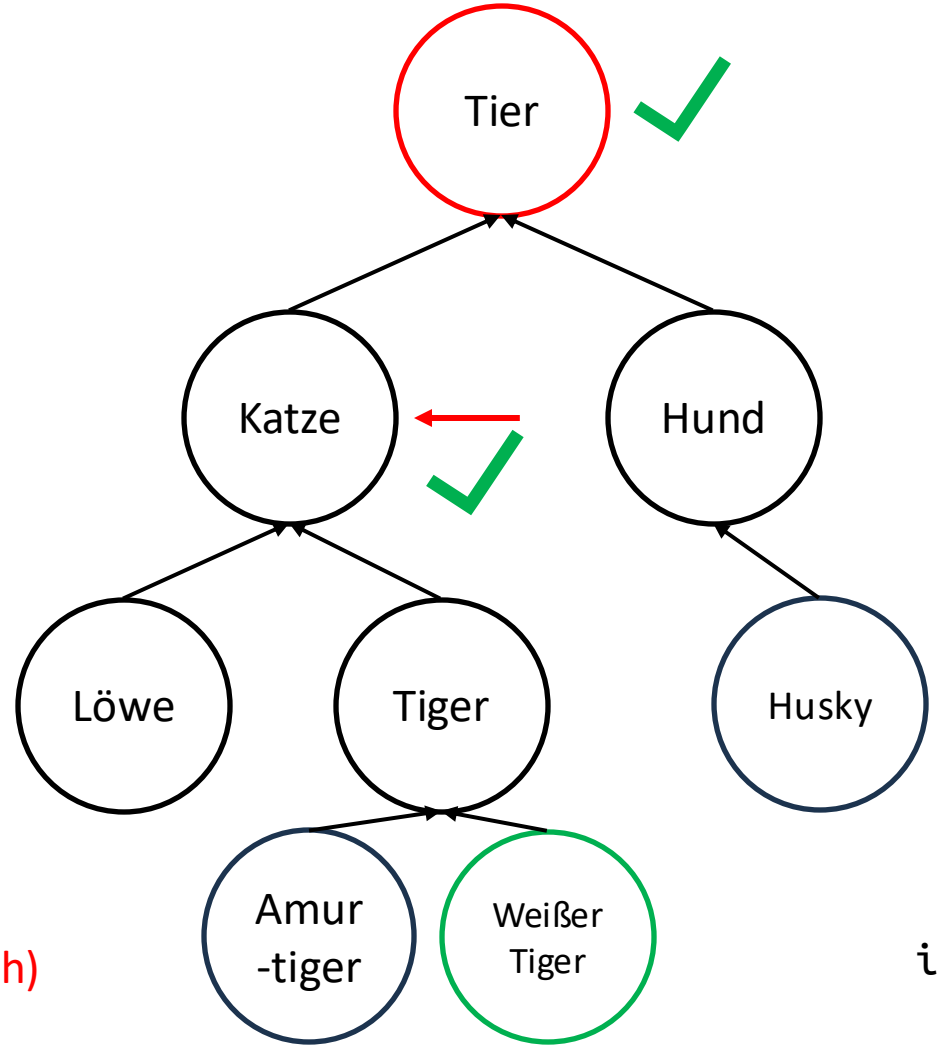
```
init.
```



Static type (unerheblich)

Dynamic type

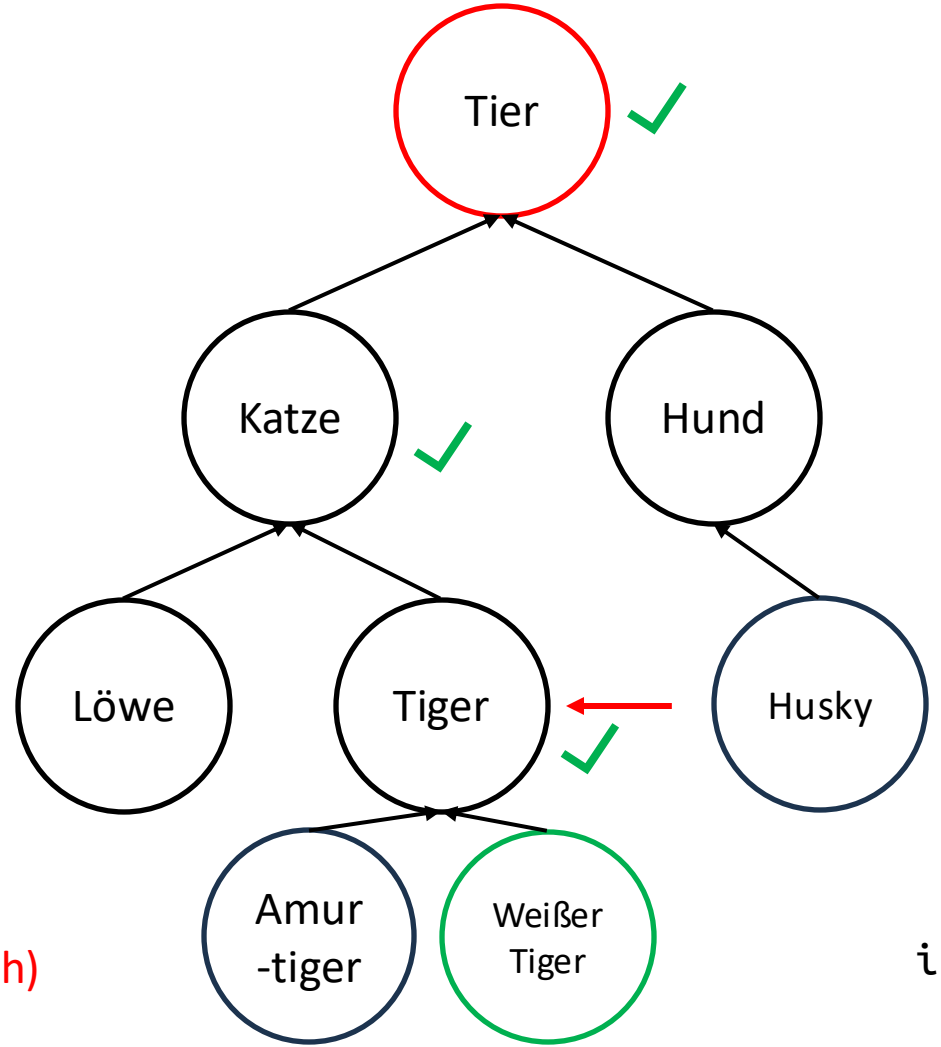
```
init.
```



Static type (unerheblich)

Dynamic type

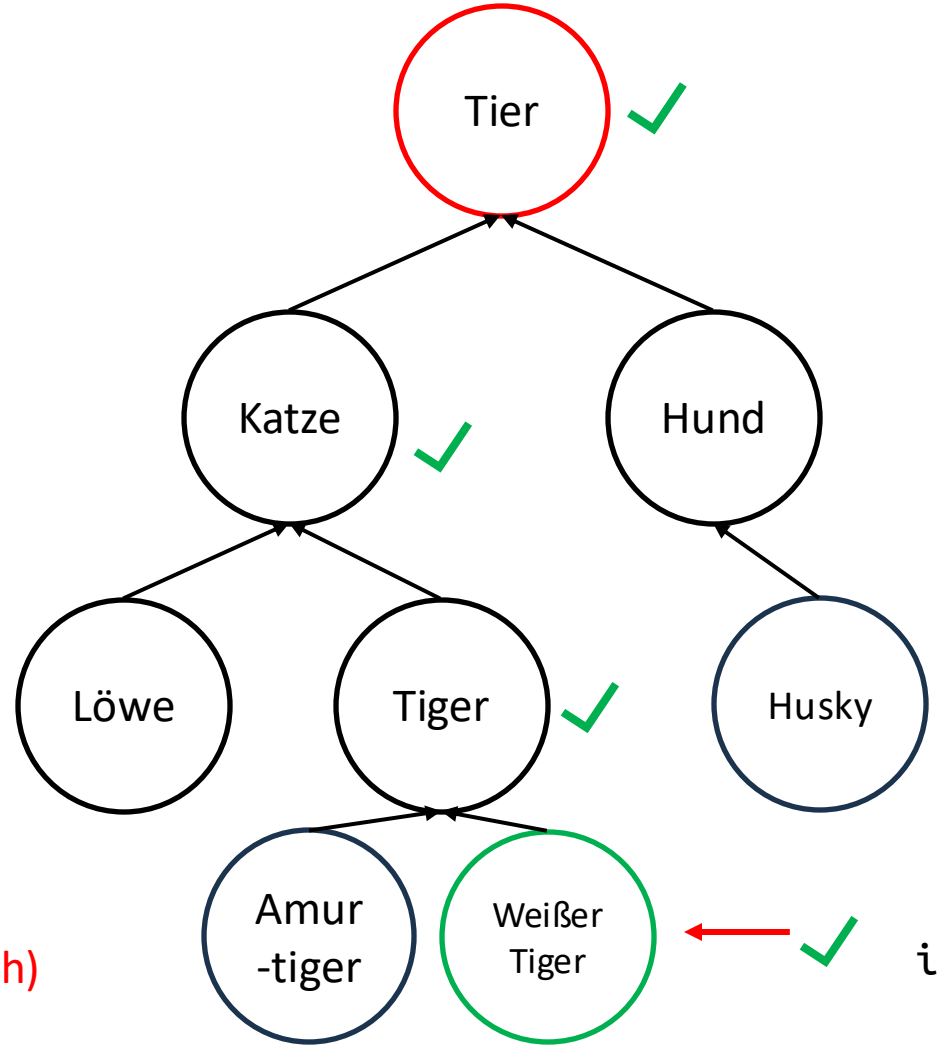
```
init.
```



Static type (unerheblich)

Dynamic type

```
init.
```



Static type (unerheblich)

Dynamic type

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()

Showcase