252-0027

Einführung in die Programmierung

Übungen

Woche 11: Vererbung, Loop-Invarianten

Timo Baumberger

Departement Informatik

ETH Zürich

Organisatorisches

- Mein Name: Timo Baumberger
- Website: timobaumberger.com
- Bei Fragen: tbaumberger@student.ethz.ch
  - Mails bitte mit «[EProg24]» im Betreff
- Neue Aufgaben: **Dienstag Abend** (im Normalfall)
- Abgabe der Übungen bis **Dienstag Abend (23:59)** Folgewoche
  - Abgabe immer via Git
  - Lösungen in separatem Projekt auf Git

Programm

- **Besprechung Bonusaufgabe u09**
- **Vererbung**
- **Casting**
  - Reference Type Narrowing
  - Reference Type Widening
  - Primitive Types
  - instanceof Keyword
- **Attributwahl / Methodenwahl / this & super**
- **Loop Invariante**
- **Scanner**

Bonusaufgabe u09

Aufgabe

- Mehrere Ordnungen in unterschiedlichen Mengen
- Ordnung im Array
- Einfüge-Ordnung unter den Elementen mit der gleichen FileID
- Anderes Beispiel für das gleiche Problem: Tische im Restaurant (Array) und Bestellungen des gleichen Gerichts (FileID)

Lösungsidee

- Speichere Elemente in zwei unterschiedlichen Mengen
- Speichere Elemente an erster freier Position im Array
- Speichere Elemente zusätzlich in einer Linked List
- Für jede FileID gibt es eine separate Linked List (notwendig?)

```java
public class WaitQueueServer {

    private final QueueEntry[] queue;
    private final int capacity;

    private int currentSize;

    /**
     * Initialize the queue array with capacity N
     * We also store the capacity of the queue array in the field capacity
     *
     * @param N is the length of the queue array
     */
    public WaitQueueServer(int N) {
        queue = new QueueEntry[N];
        capacity = N;
    }
```

```java
public class QueueEntry {

    int index;

    int fileID;
    char userID;
    boolean readOnly;

    QueueEntry next;
    QueueEntry prev;

    public QueueEntry(int index, int fileID, char userID, boolean readOnly, QueueEntry next, QueueEntry prev) {
        this.index = index;
        this.fileID = fileID;
        this.userID = userID;
        this.readOnly = readOnly;
        this.next = next;
        this.prev = prev;
    }

    public boolean isHead() {
        return this.prev == null;
    }

    public boolean isTail() {
        return this.next == null;
    }
}
```

```java
public Response add(int fileID, char userID, boolean readOnly) {
    // currentSize stores the current number of elements in the array queue
    if (currentSize == capacity) { // if currentSize == capacity, then queue is full
        return null;
    }
    // Precondition: currentSize != capacity
    // Based on the precondition we know that there will be a free spot in the queue array, hence
    // we already increase the size of the queue array.
    currentSize++;
    QueueEntry firstOccurrence = findFirstOccurringEntry(fileID);
    QueueEntry head = findHead(firstOccurrence);
    QueueEntry tail = findTail(firstOccurrence);

    // Find index / position in the queue array that is not used
    int i = 0;
    while (queue[i] != null) {
        i++;
    }

    // Create new entry
    QueueEntry newEntry = new QueueEntry(i, fileID, userID, readOnly, null, tail);
    queue[i] = newEntry;
    // If firstOccurrence == null, then we know that this entry will be the first entry with this fileID
    if (firstOccurrence == null) {
        return new Response(i, i);
    }

    // Update tail
    if (tail != null) {
        tail.next = newEntry;
    }
    tail = newEntry;
    return new Response(head.index, tail.index);
}
```

```java
public char[] pop(int fileID) {
    QueueEntry head = findHead(findFirstOccuringEntry(fileID));
    if (head == null) {
        return null;
    }
    if (head.readOnly) {
        LinkedList<Character> interResult = new LinkedList<>();
        QueueEntry next = head;
        while (next != null) {
            if (!next.readOnly) {
                next = next.next;
                continue;
            }
            interResult.add(next.userID);
            remove(next);
            next = next.next;
        }

        return linkedListToCharArray(interResult);
    } else {
        remove(head);
        return new char[] {head.userID};
    }
}

public char[] linkedListToCharArray(LinkedList<Character> list) {
    char[] charArray = new char[list.size()];
    for (int i = 0; i < list.size(); i++) {
        charArray[i] = list.get(i);
    }
    return charArray;
}
```

```java
public void remove(QueueEntry entry) {
    if (entry.prev != null) {
        entry.prev.next = entry.next;
    }
    if (entry.next != null) {
        entry.next.prev = entry.prev;
    }
    currentSize--;
    queue[entry.index] = null;
}
```

```java
public int[][] getQuickList() {
    if (currentSize == 0) {
        return null;
    }

    // find and store distinct fileIDs. ∀u,v ∈ fileIDs: u ≠ v
    // Would be more efficient with TreeSet, because ArrayList.contains() needs O(n) time where n is the size of the ArrayList
    ArrayList<Integer> fileIDs = new ArrayList<>();
    for (int i = 0; i < capacity; i++) {
        QueueEntry entry = queue[i];
        if (entry != null && !fileIDs.contains(entry.fileID)) {
            fileIDs.add(entry.fileID);
        }
    }
    Collections.sort(fileIDs);

    int size = fileIDs.size();
    int[][] result = new int[size][3];
    for (int i = 0; i < size; i++) {
        int fileID = fileIDs.get(i);
        QueueEntry firstOccurrence = findFirstOccurringEntry(fileID);
        QueueEntry head = findHead(firstOccurrence);
        QueueEntry tail = findTail(firstOccurrence);
        result[i][0] = fileID;
        result[i][1] = head.index;
        result[i][2] = tail.index;
    }

    return result;
}
```

Loop-Invariante

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
     n += k % 10;
     k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Loop-Invariante vorgehen:

- Code anschauen: was passiert?

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
     n += k % 10;
     k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Loop-Invariante vorgehen:

- Code anschauen: was passiert?
  - Initialisierung

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
    n += k % 10;
    k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Loop-Invariante vorgehen:

- Code anschauen: was passiert?
  - Initialisierung
  - Bis k == 0 ist, wird k in jeder Iteration durch 10 dividiert

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
    n += k % 10;
    k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Bsp:
k = 1234
k / 10 = 123

Loop-Invariante vorgehen:

- Code anschauen: was passiert?
  - Initialisierung
  - Bis k == 0 ist, wird k in jeder Iteration durch 10 dividiert

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
    n += k % 10;
    k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Bsp:
k = 1234
k / 10 = 123

Loop-Invariante vorgehen:

- Code anschauen: was passiert?
  - Initialisierung
  - Bis k == 0 ist, wird k in jeder Iteration durch 10 dividiert → **immer die letzte Ziffer wird “abgeschnitten”**

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
     n += k % 10;
     k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Loop-Invariante vorgehen:

- Code anschauen: was passiert?
  - Initialisierung
  - Bis k == 0 ist, wird k in jeder Iteration durch 10 dividiert → **immer die letzte Ziffer wird “abgeschnitten”**
  - In jeder Iteration wird k % 10 zu n addiert

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
    n += k % 10;
    k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Bsp:
k = 1234
k % 10 = 4

Loop-Invariante vorgehen:

- Code anschauen: was passiert?
  - Initialisierung
  - Bis k == 0 ist, wird k in jeder Iteration durch 10 dividiert → **immer die letzte Ziffer wird "abgeschnitten"**
  - In jeder Iteration wird k % 10 zu n addiert

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
    n += k % 10;
    k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Bsp:
k = 1234
k % 10 = 4

Loop-Invariante vorgehen:

- Code anschauen: was passiert?
  - Initialisierung
  - Bis k == 0 ist, wird k in jeder Iteration durch 10 dividiert → **immer die letzte Ziffer wird “abgeschnitten”**
  - In jeder Iteration wird k % 10 zu n addiert → **immer die letzte Ziffer wird dazu addiert**

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
    n += k % 10;
    k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Loop-Invariante vorgehen:

- Code anschauen: was passiert?
  - Initialisierung
  - Bis k == 0 ist, wird k in jeder Iteration durch 10 dividiert → immer die letzte Ziffer wird “abgeschnitten”
  - In jeder Iteration wird k % 10 zu n addiert → immer die letzte Ziffer wird dazu addiert
  - **=> n == alle Ziffern von i zusammen addiert**

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
    n += k % 10;
    k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Loop-Invariante vorgehen:

- Code anschauen: was passiert?
  - Initialisierung
  - Bis k == 0 ist, wird k in jeder Iteration durch 10 dividiert → immer die letzte Ziffer wird "abgeschnitten"
  - In jeder Iteration wird k % 10 zu n addiert → immer die letzte Ziffer wird dazu addiert
  - **=> n == alle Ziffern von i zusammen addiert.**
  - das ist die **Quersumme** von i

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
     n += k % 10;
     k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Loop-Invariante vorgehen:

- Code anschauen: was passiert?
  - Initialisierung
  - Bis k == 0 ist, wird k in jeder Iteration durch 10 dividiert → immer die letzte Ziffer wird "abgeschnitten"
  - In jeder Iteration wird k % 10 zu n addiert → immer die letzte Ziffer wird dazu addiert
  - **=> n == alle Ziffern von i zusammen addiert.**
  - das ist die **Quersumme** von I
- Postcondition anschauen

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
     n += k % 10;
     k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Loop-Invariante vorgehen:

- Code anschauen: was passiert?
  - Initialisierung
  - Bis k == 0 ist, wird k in jeder Iteration durch 10 dividiert → immer die letzte Ziffer wird “abgeschnitten”
  - In jeder Iteration wird k % 10 zu n addiert → immer die letzte Ziffer wird dazu addiert
  - **=> n == alle Ziffern von i zusammen addiert.**
  - das ist die **Quersumme** von I
- Postcondition anschauen
  - Unsere Vermutung war also richtig: die Quersumme von i wird berechnet

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
    n += k % 10;
    k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Loop-Invariante vorgehen (Fortsetzung):

- Am Schluss wollen wir auf die Postcondition kommen. Aber quersumme(i) == n stimmt am Anfang im Allgemeinen definitiv nicht
- quersumme(i) != 0

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
    n += k % 10;
    k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Loop-Invariante vorgehen (Fortsetzung):

- Am Schluss wollen wir auf die Postcondition kommen. Aber quersumme(i) == n stimmt am Anfang im Allgemeinen definitiv nicht
- quersumme(i) != 0
- **k** und **n** werden während dem Loop verändert, also müssen diese irgendwie in der Invariante vorkommen

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
     n += k % 10;
     k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

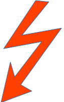
Loop-Invariante vorgehen (Fortsetzung):

- Am Schluss wollen wir auf die Postcondition kommen. Aber quersumme(i) == n stimmt am Anfang im Allgemeinen definitiv nicht
- quersumme(i) != 0
- **k** und **n** werden während dem Loop verändert, also müssen diese irgendwie in der Invariante vorkommen
- Versuch 1: quersumme(k) == n

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
    n += k % 10;
    k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Loop-Invariante vorgehen (Fortsetzung):

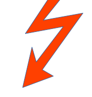
- Am Schluss wollen wir auf die Postcondition kommen. Aber quersumme(i) == n stimmt am Anfang im Allgemeinen definitiv nicht
- quersumme(i) != 0
- **k** und **n** werden während dem Loop verändert, also müssen diese irgendwie in der Invariante vorkommen
- Versuch 1: quersumme(k) == n
- Prüfen:
  - Anfangs: quersumme(i) == 0



```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
    n += k % 10;
    k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

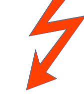
Loop-Invariante vorgehen (Fortsetzung):

- Am Schluss wollen wir auf die Postcondition kommen. Aber quersumme(i) == n stimmt am Anfang im Allgemeinen definitiv nicht
- quersumme(i) != 0
- **k** und **n** werden während dem Loop verändert, also müssen diese irgendwie in der Invariante vorkommen
- Versuch 1: quersumme(k) == n
- Prüfen:
  - Anfangs: quersumme(i) == 0
  - Wir haben links also quersumme(i) “zu viel”



```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
    n += k % 10;
    k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

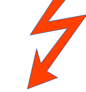
Loop-Invariante vorgehen (Fortsetzung):

- Am Schluss wollen wir auf die Postcondition kommen. Aber quersumme(i) == n stimmt am Anfang im Allgemeinen definitiv nicht
- quersumme(i) != 0
- **k** und **n** werden während dem Loop verändert, also müssen diese irgendwie in der Invariante vorkommen
- Versuch 1: quersumme(k) == n
- Prüfen:
  - Anfangs: quersumme(i) == 0 
  - Wir haben links also quersumme(i) “zu viel”
- Versuch 2: quersumme(k) - quersumme(i) == n

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
    n += k % 10;
    k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Loop-Invariante vorgehen (Fortsetzung):

- Am Schluss wollen wir auf die Postcondition kommen. Aber quersumme(i) == n stimmt am Anfang im Allgemeinen definitiv nicht
- quersumme(i) != 0
- **k** und **n** werden während dem Loop verändert, also müssen diese irgendwie in der Invariante vorkommen
- Versuch 1: quersumme(k) == n
- Prüfen:
  - Anfangs: quersumme(i) == 0
  - Wir haben links also quersumme(i) “zu viel”



- Versuch 2: quersumme(k) - quersumme(i) == n
- Prüfen:
  - Anfangs: quersumme(i) – quersumme(i) == 0

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
    n += k % 10;
    k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Loop-Invariante vorgehen (Fortsetzung):

- Am Schluss wollen wir auf die Postcondition kommen. Aber quersumme(i) == n stimmt am Anfang im Allgemeinen definitiv nicht
- quersumme(i) != 0
- **k** und **n** werden während dem Loop verändert, also müssen diese irgendwie in der Invariante vorkommen
- Versuch 1: quersumme(k) == n
- Prüfen:
  - Anfangs: quersumme(i) == 0
  - Wir haben links also quersumme(i) "zu viel"
- Versuch 2: quersumme(k) - quersumme(i) == n
- Prüfen:
  - Anfangs: quersumme(i) – quersumme(i) == 0
  - Ende: quersumme(0) – quersumme(i) == quersumme(i)

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
     n += k % 10;
     k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Loop-Invariante vorgehen (Fortsetzung):

- Am Schluss wollen wir auf die Postcondition kommen. Aber quersumme(i) == n stimmt am Anfang im Allgemeinen definitiv nicht
- quersumme(i) != 0
- **k** und **n** werden während dem Loop verändert, also müssen diese irgendwie in der Invariante vorkommen
- Versuch 1: quersumme(k) == n
- Prüfen:
  - Anfangs: quersumme(i) == 0 ϟ
  - Wir haben links also quersumme(i) "zu viel"
- Versuch 2: quersumme(k) - quersumme(i) == n
- Prüfen:
  - Anfangs: quersumme(i) – quersumme(i) == 0
  - Ende: quersumme(0) – quersumme(i) == quersumme(i) ϟ
  - Das Vorzeichen stimmt nicht

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
     n += k % 10;
     k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Loop-Invariante vorgehen (Fortsetzung):

- Versuch 3: quersumme(i) – quersumme(k) == n

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
     n += k % 10;
     k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Loop-Invariante vorgehen (Fortsetzung):

- Versuch 3: quersumme(i) – quersumme(k) == n
- Prüfen:
  - Anfangs: quersumme(i) – quersumme(i) == 0

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
    n += k % 10;
    k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Loop-Invariante vorgehen (Fortsetzung):

- Versuch 3: quersumme(i) – quersumme(k) == n
- Prüfen:
  - Anfangs: quersumme(i) – quersumme(i) == 0
  - Nach jeder Iteration:
  - quersumme(i) – quersumme(k/10) == n + k%10
  - "k ohne letzte Ziffer" "letzte Ziffer von k"

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
     n += k % 10;
     k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Loop-Invariante vorgehen (Fortsetzung):

- Versuch 3: quersumme(i) – quersumme(k) == n
- Prüfen:
  - Anfangs: quersumme(i) – quersumme(i) == 0
  - Nach jeder Iteration:
  - quersumme(i) – quersumme(k/10) == n + k%10
  - "k ohne letzte Ziffer" "letzte Ziffer von k"
  - Ende: quersumme(i) – quersumme(0) == quersumme(i)
- Korrekt !!

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
     n += k % 10;
     k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Loop-Invariante vorgehen (Fortsetzung):

Inv: quersumme(i) – quersumme(k) == n

- Es fehlt noch etwas!

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
    n += k % 10;
    k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Loop-Invariante vorgehen (Fortsetzung):

Inv: quersumme(i) – quersumme(k) == n

- Es fehlt noch etwas!
- Der Parameter von quersumme() muss positiv sein:
- i >= 0 && k >= 0
- (ein Hint, um darauf zu kommen, kann uns die Precondition geben)

```java
public static int compute(int i) {
  // Precondition: i >= 0
  int n;
  int k;
  n = 0;
  k = i;

  //Loop-Invariante:
  While (k != 0) {
    n += k % 10;
    k = k / 10;
  }

  //Postcondition: quersumme(i) == n
  return n;
} //quersumme(i) gibt die Quersumme von int i zurück (i >= 0)
```

Loop-Invariante vorgehen (Fortsetzung):

Inv: quersumme(i) – quersumme(k) == n

- Es fehlt noch etwas!
- Der Parameter von quersumme muss positiv sein:
- i >= 0 && k >= 0
- (ein Hint, um darauf zu kommen, kann uns die Precondition geben)

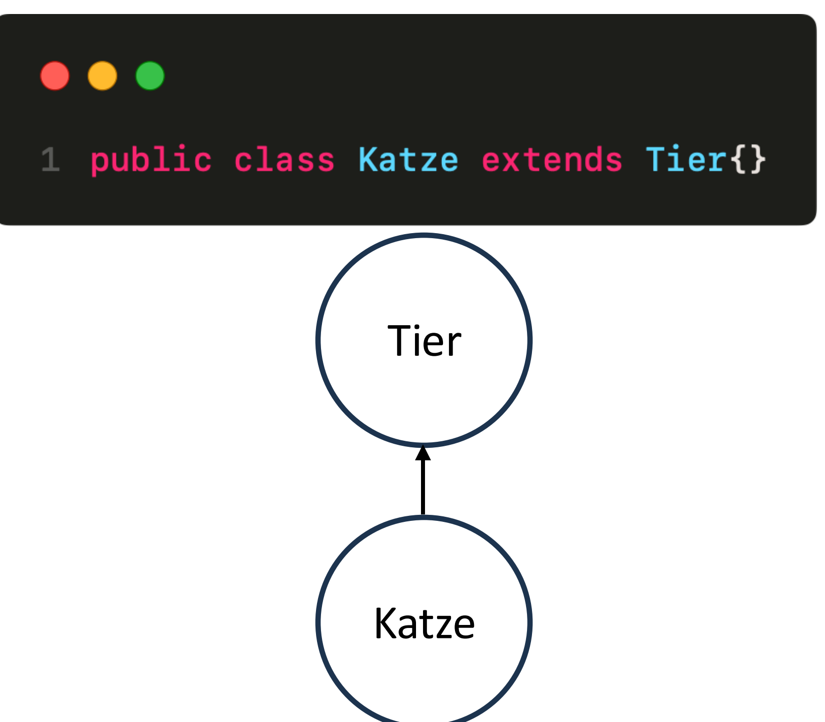
- Fertige Invariante:

quersumme(i) – quersumme(k) == n && i >= 0 && k >= 0

Vererbung / 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

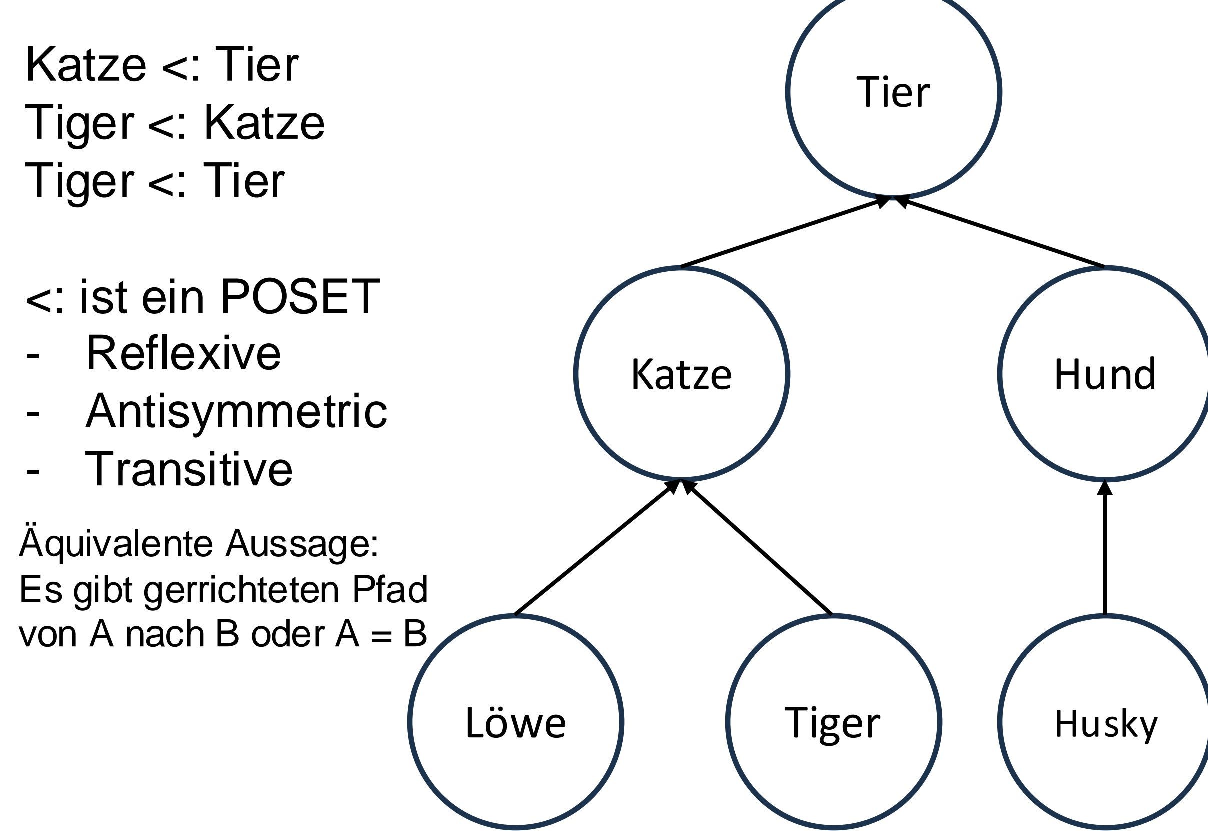


Katze <: Tier
Tiger <: Katze
Tiger <: Tier

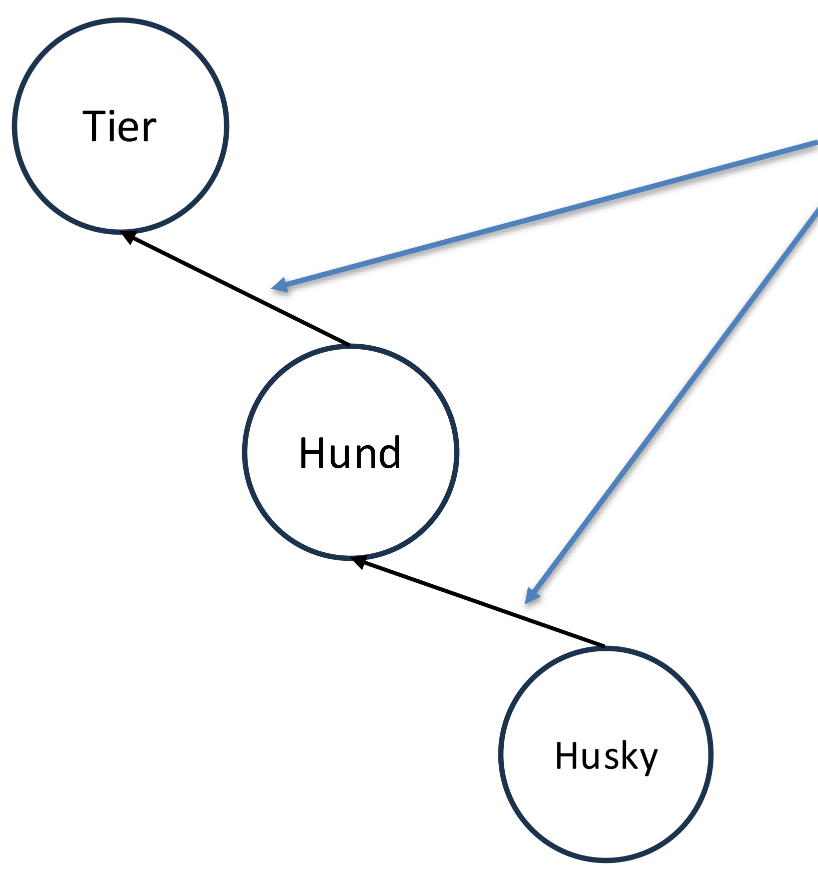
<: ist ein POSET

- Reflexive
- Antisymmetric
- Transitive

Äquivalente Aussage:
Es gibt gerrichteten Pfad
von A nach B oder A = B



Vererbungshierarchie

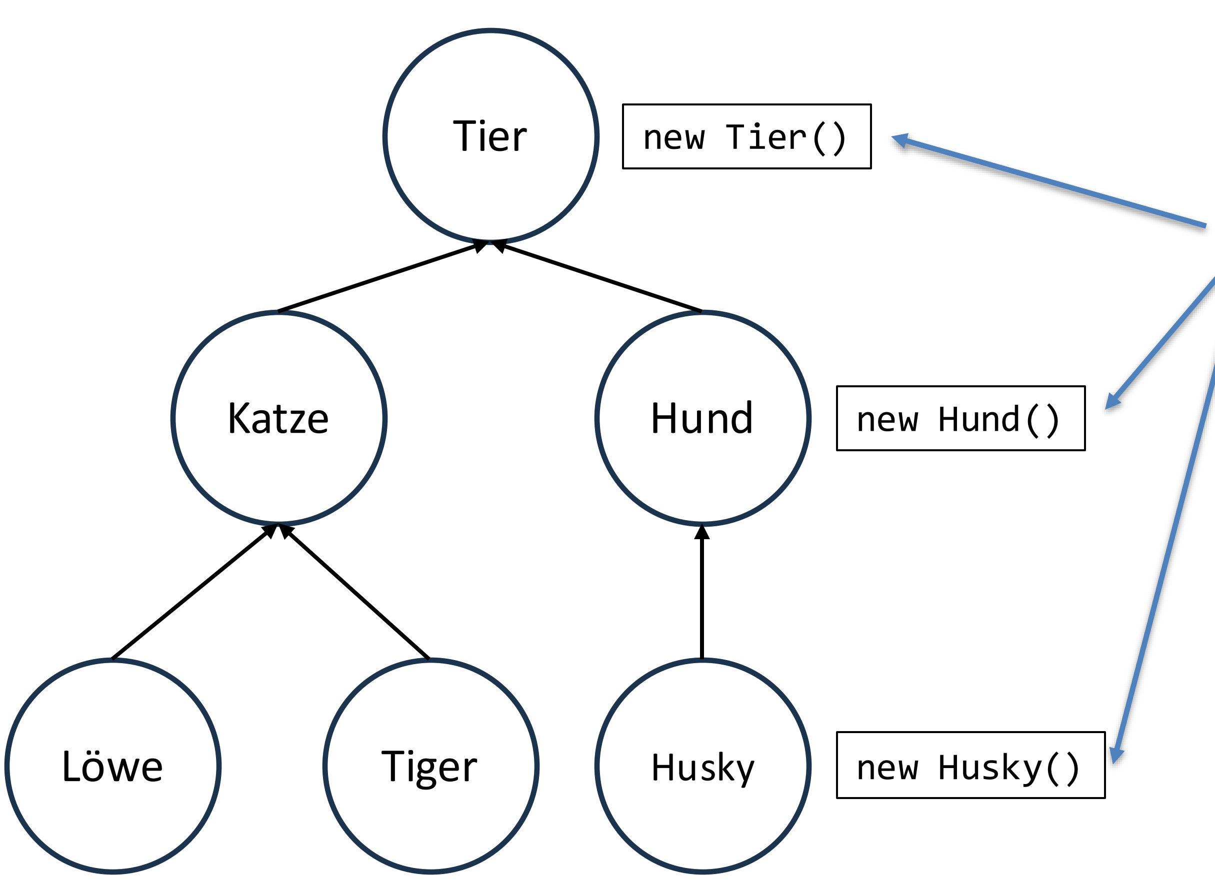


Die Pfeile sind eine "ist ein" - Beziehung

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

- Ein Husky ist ein Tier und ein Hund.

Objekt vs Referenz

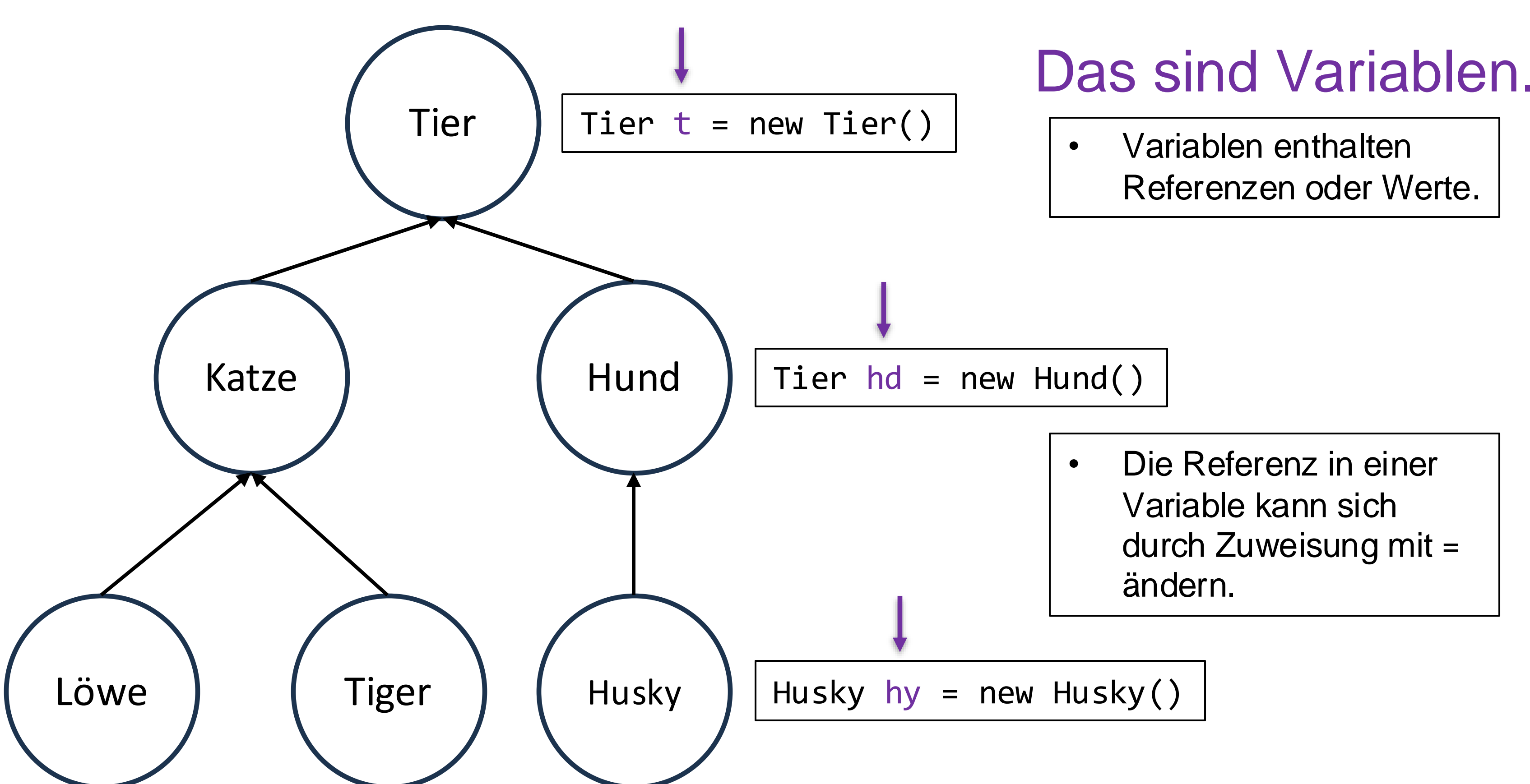


Das sind Objekte.

- 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

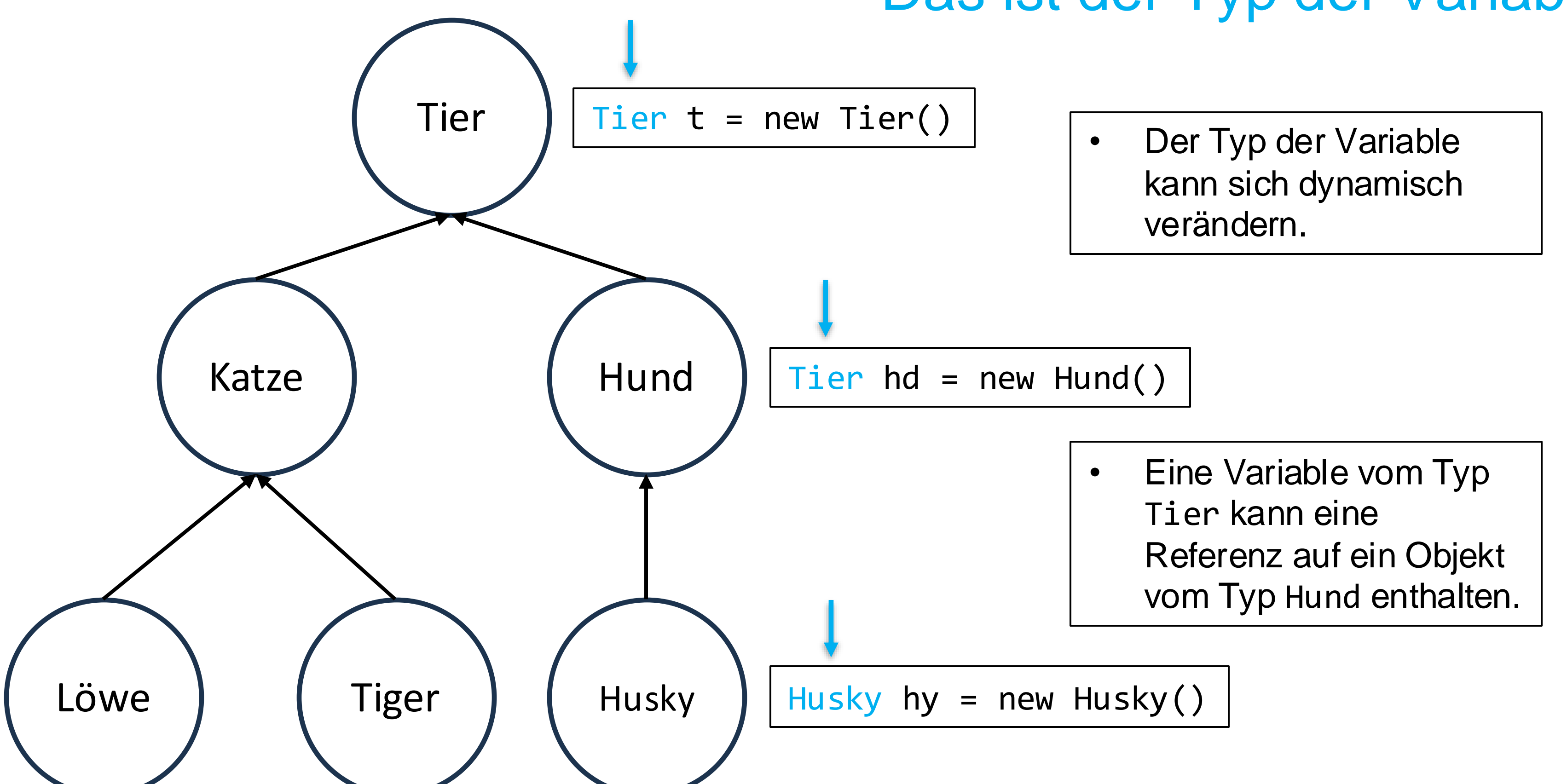


Das sind Variablen.

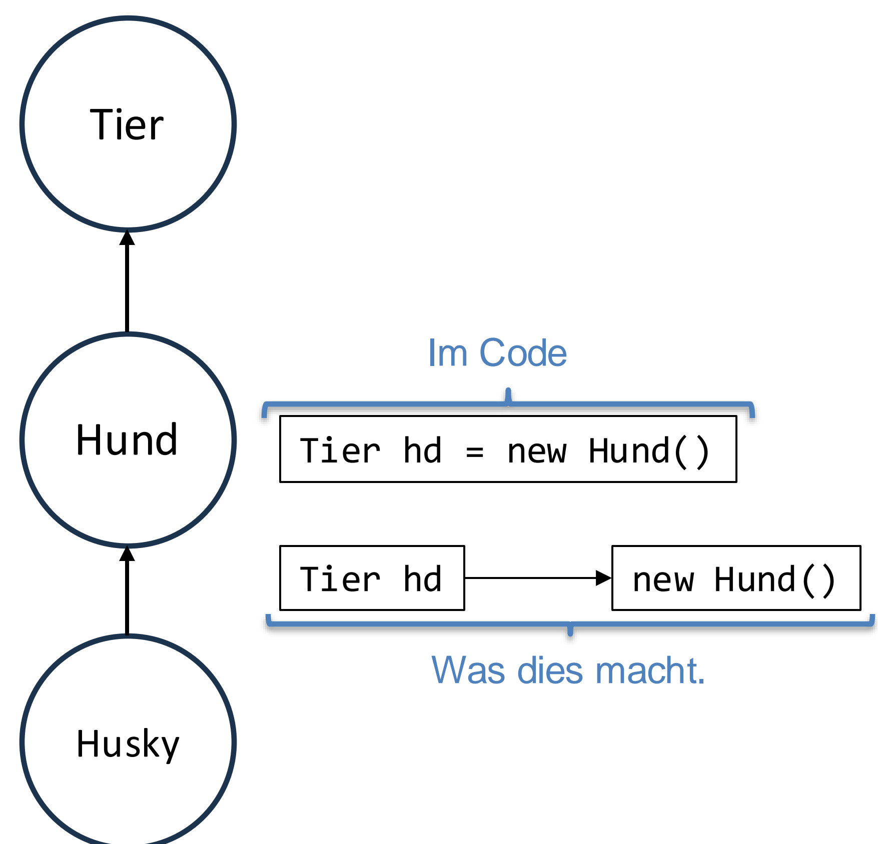
- Variablen enthalten Referenzen oder Werte.

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

Objekt vs Referenz

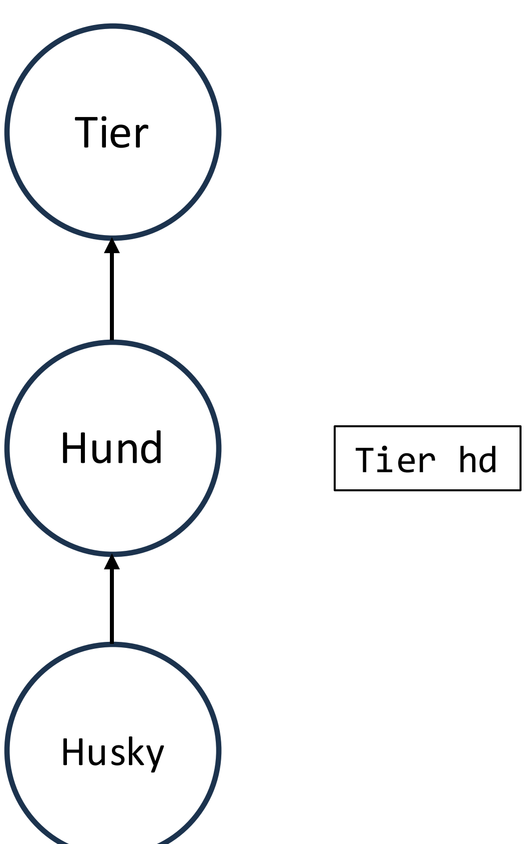


Objekt vs Referenz



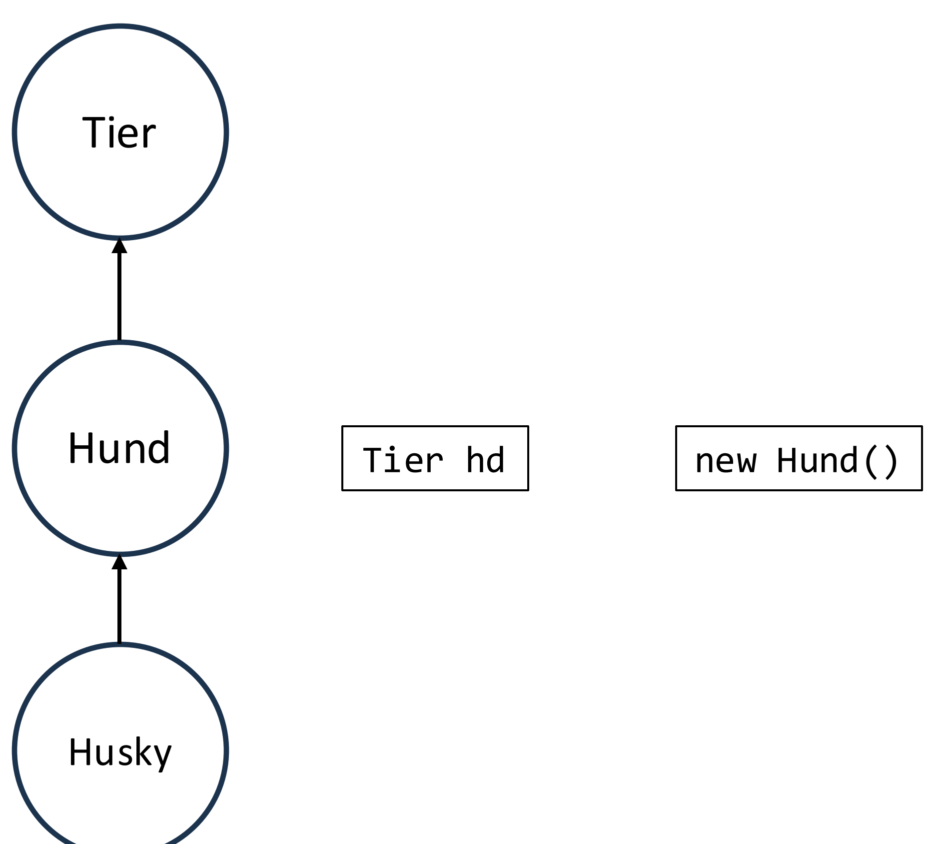
Objekt vs Referenz

`Tier hd`



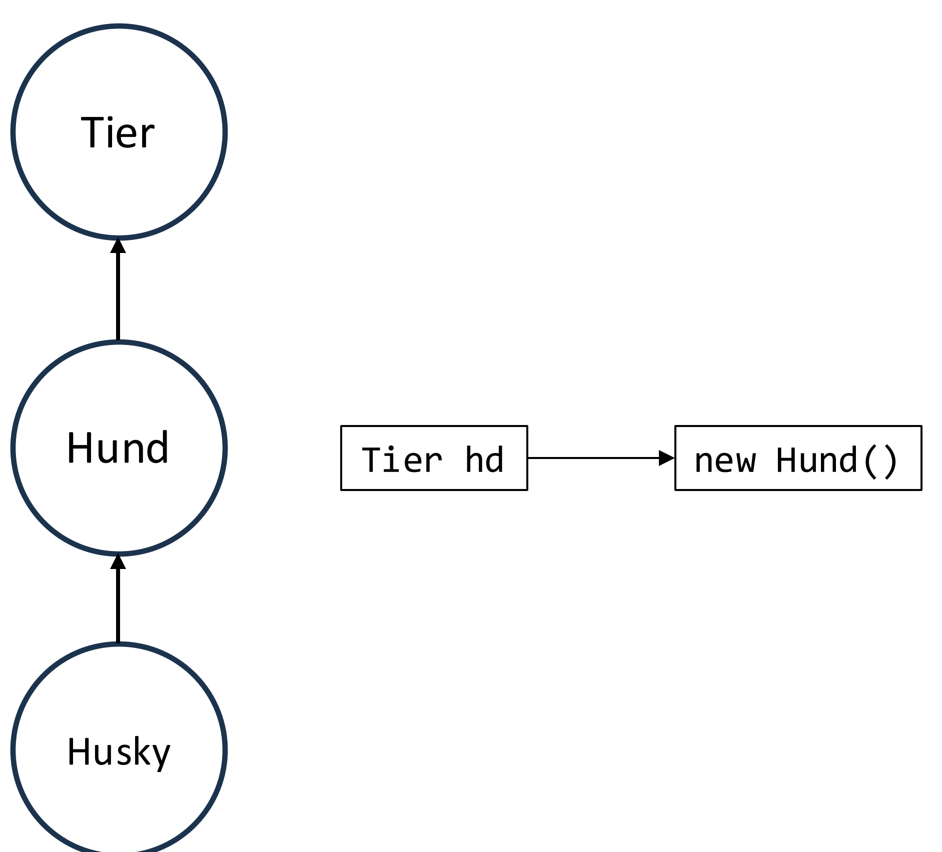
Objekt vs Referenz

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



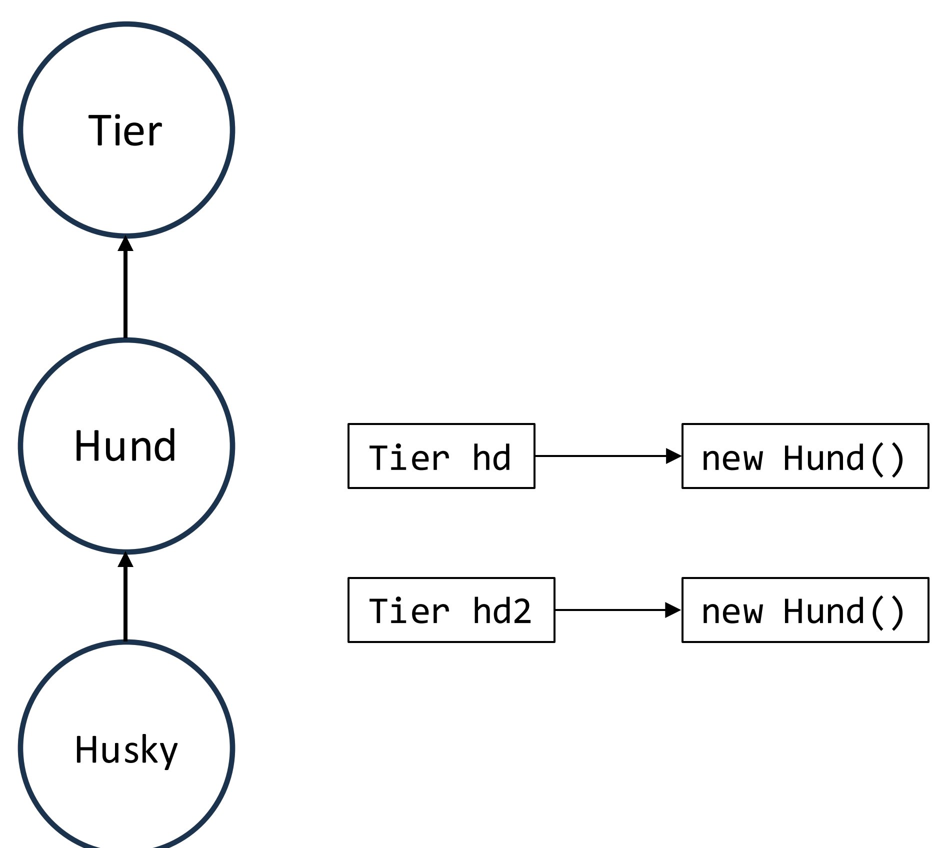
Objekt vs Referenz

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



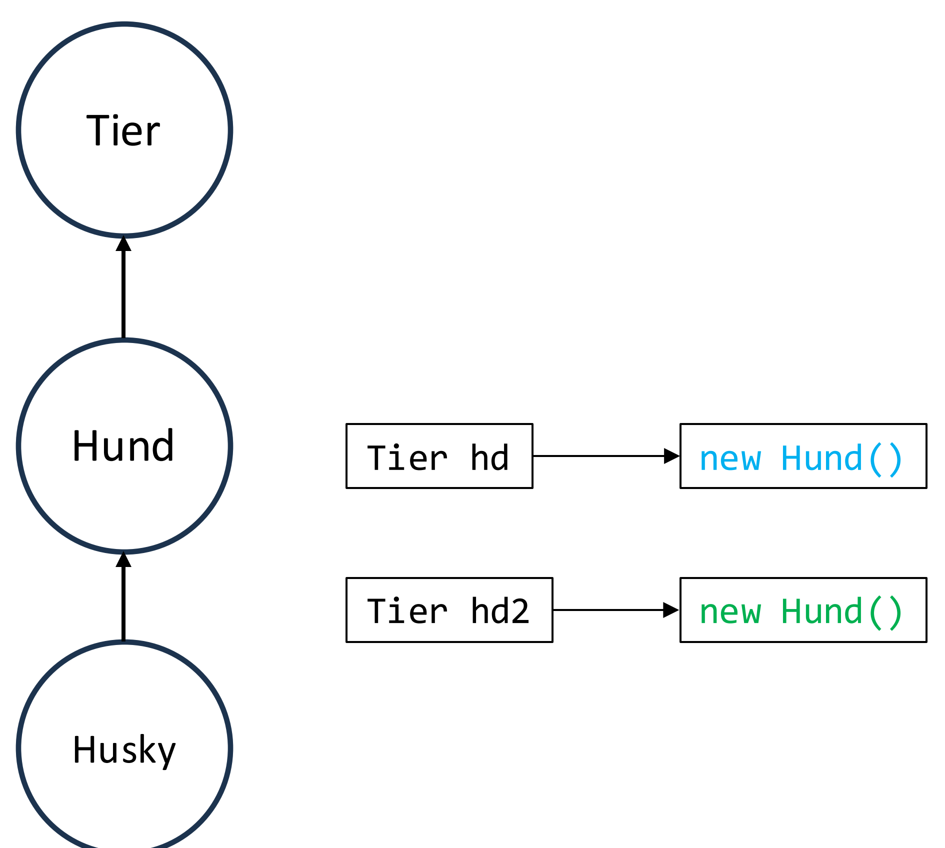
Objekt vs Referenz

```
Tier hd = new Hund();

Tier hd2 = new Hund();
```

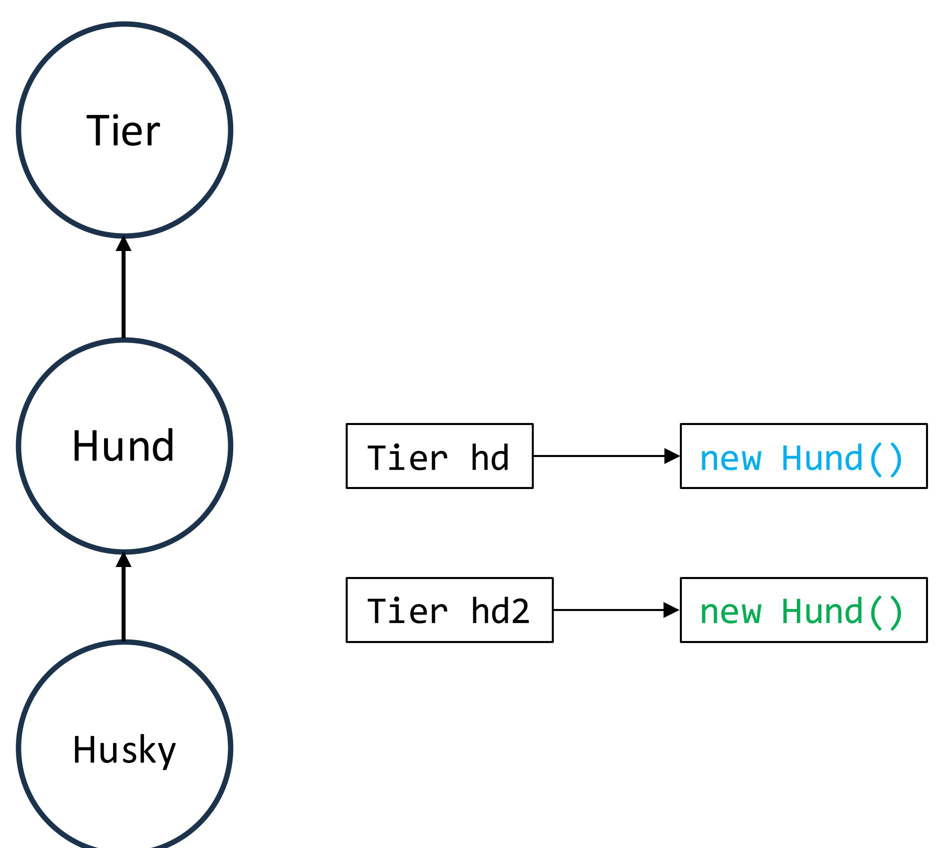


Objekt vs Referenz

```
Tier hd = new Hund();

Tier hd2 = new Hund();
```



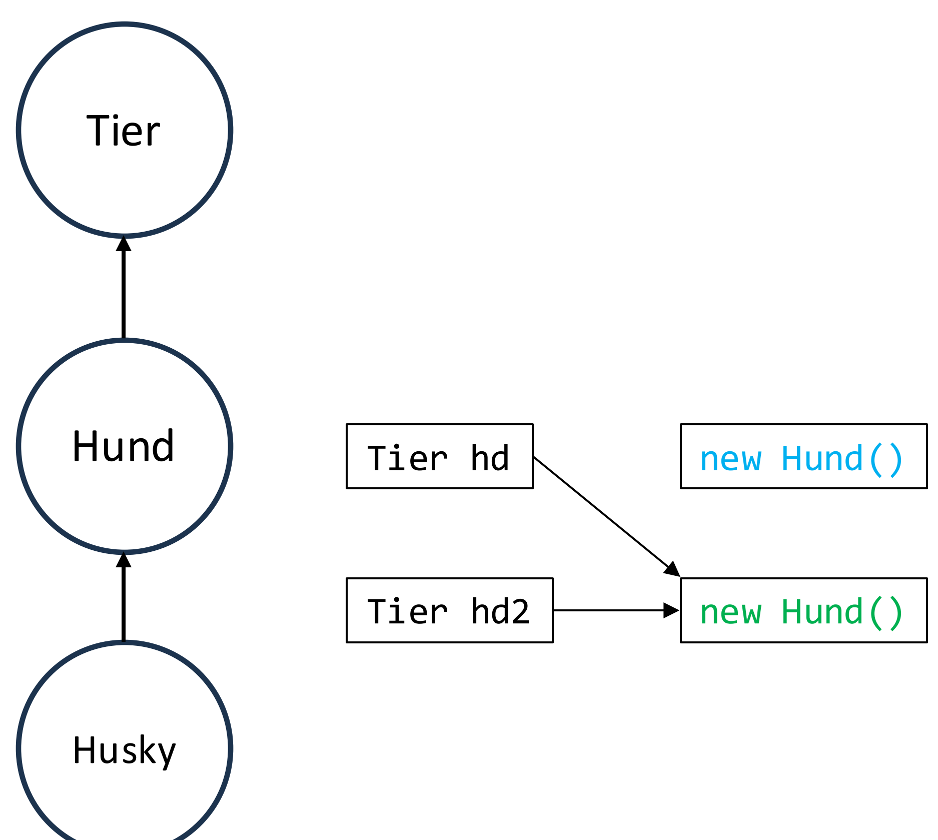
Objekt vs Referenz



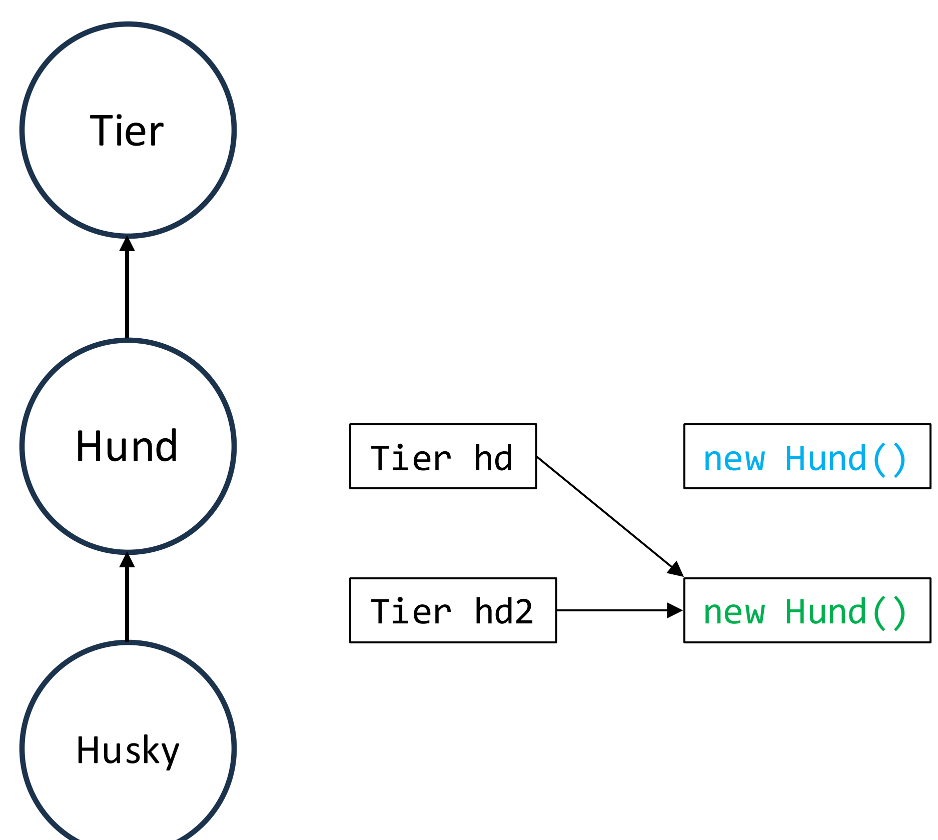

```
Tier hd = new Hund();

Tier hd2 = new Hund();

hd = hd2;
```

Objekt vs Referenz

```
Tier hd = new Hund();

Tier hd2 = new Hund();

hd = hd2;
```



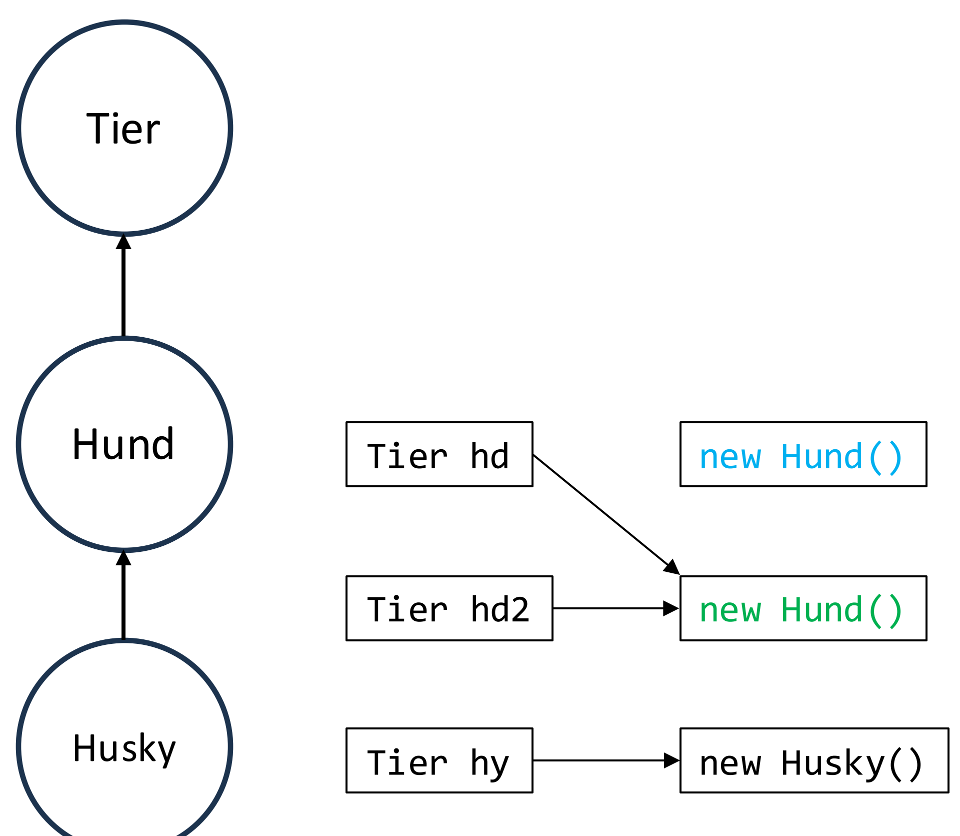
Objekt vs Referenz



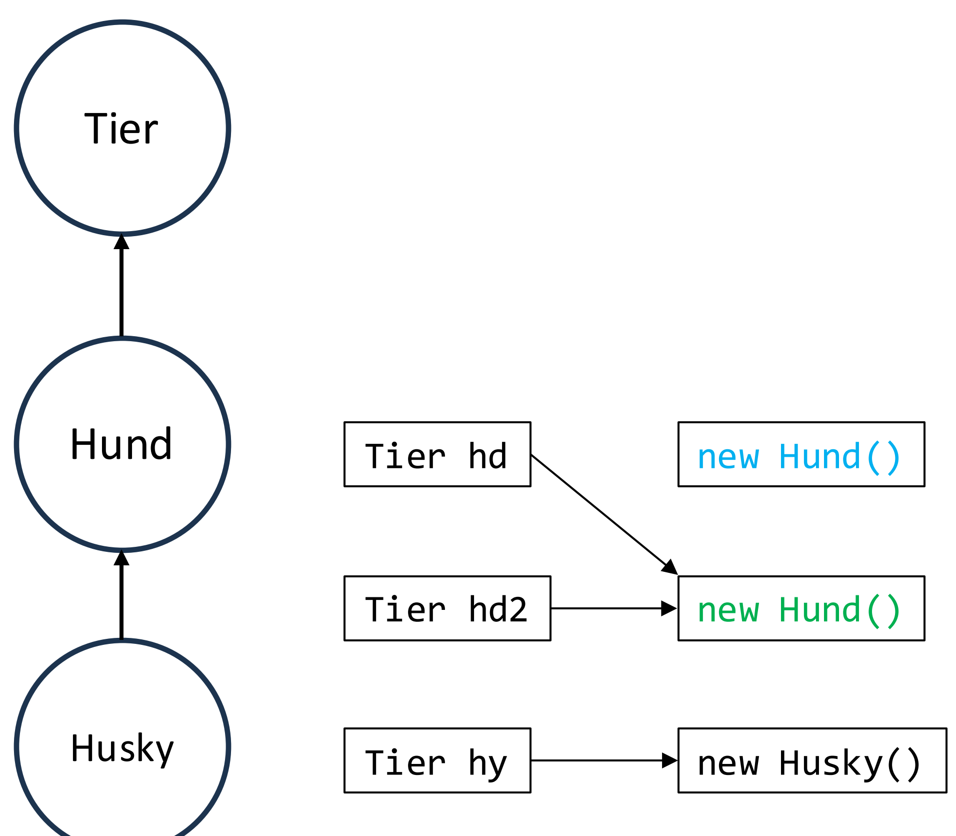

```
Tier hd = new Hund();

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();
```

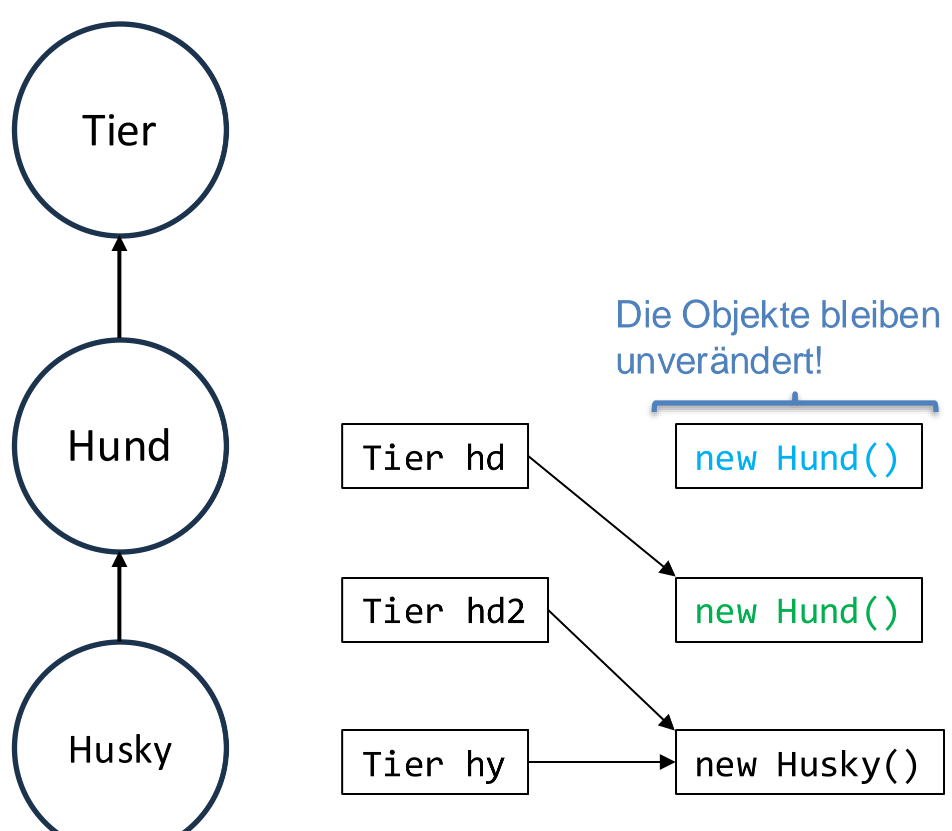


Objekt vs Referenz



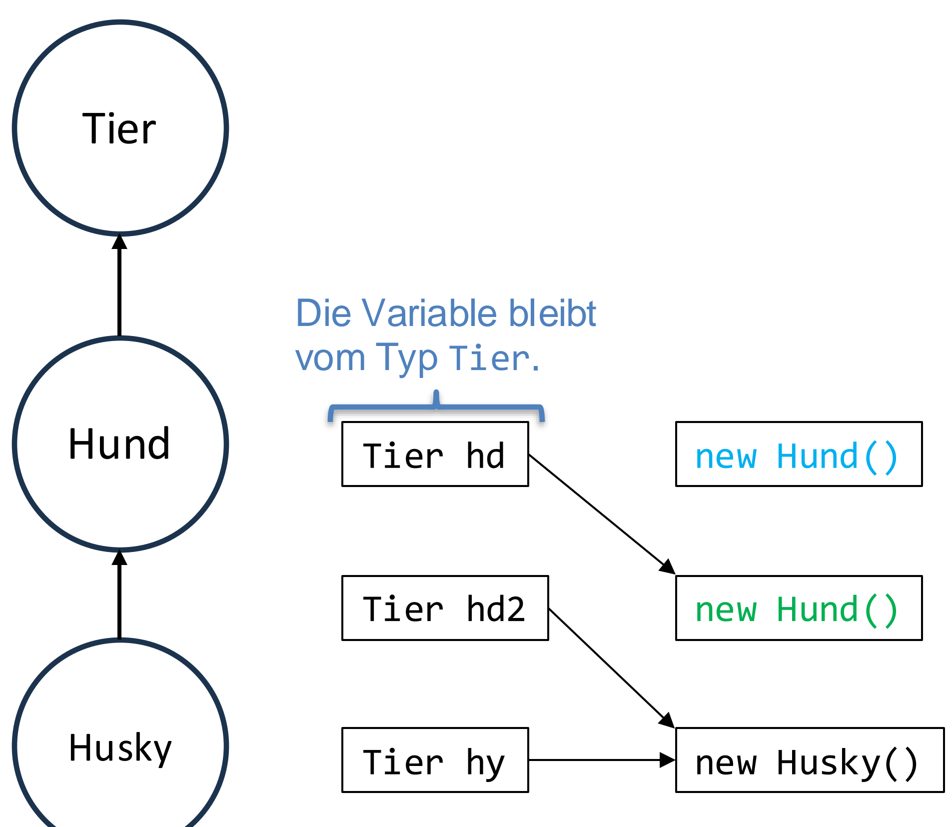

```
Tier hd = new Hund();

Tier hd2 = new Hund();

hd = hd2;

Tier hy = new Husky();

hd2 = hy;
```

Objekt vs Referenz



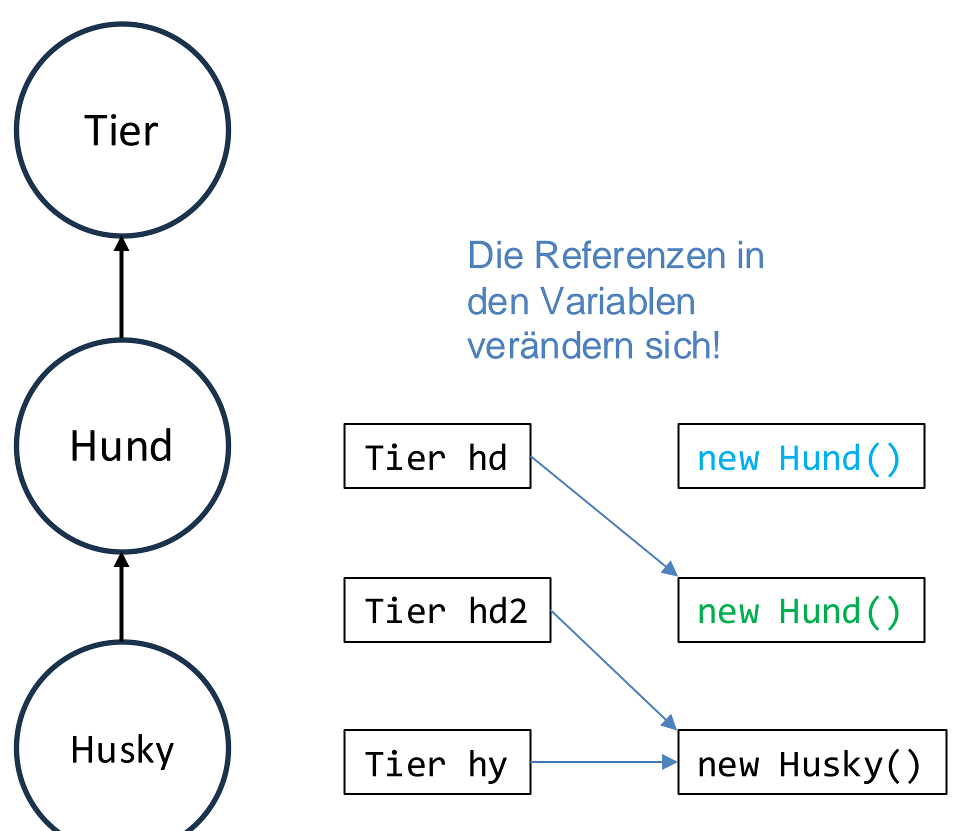

```
Tier hd = new Hund();

Tier hd2 = new Hund();

hd = hd2;

Tier hy = new Husky();

hd2 = hy;
```

Objekt vs Referenz



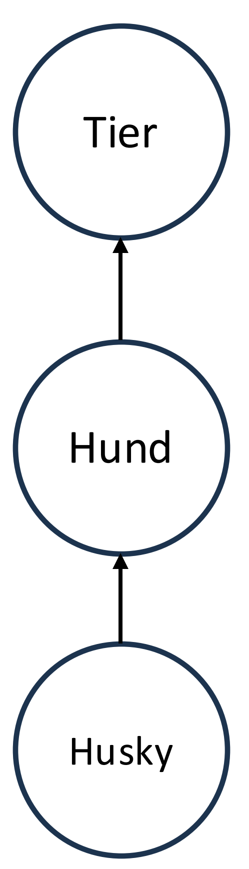

```
Tier hd = new Hund();

Tier hd2 = new Hund();

hd = hd2;

Tier hy = new Husky();

hd2 = hy;
```

Objekt vs Referenz




```
Tier hd = new Hund();

Tier hd2 = new Hund();

hd = hd2;

Tier hy = new Husky();

hd2 = hy;
```

Objekt vs Referenz

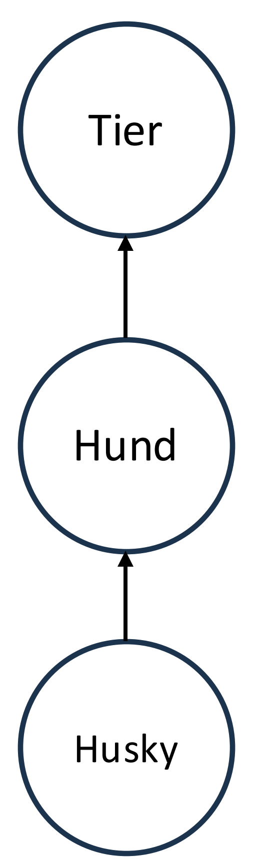


Wie unterscheiden wir zwischen:

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

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

Objekt vs Referenz



Wie unterscheiden wir zwischen:

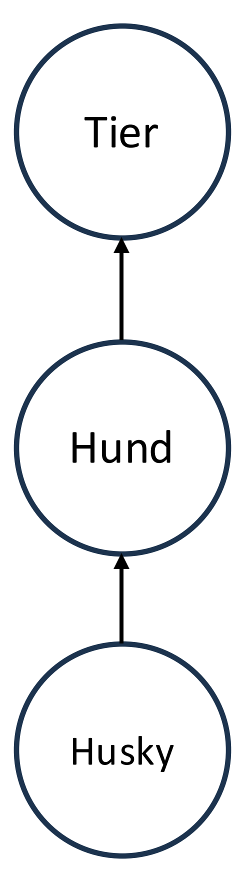
- **Typ der Variable**
- **Typ des Objekts** auf welches die **Referenz** in der Variable **zeigt**?

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

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

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

Objekt vs Referenz

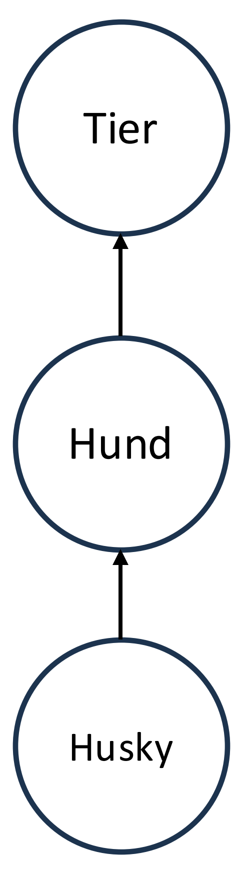


Wie unterscheiden wir zwischen:

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

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

Objekt vs Referenz

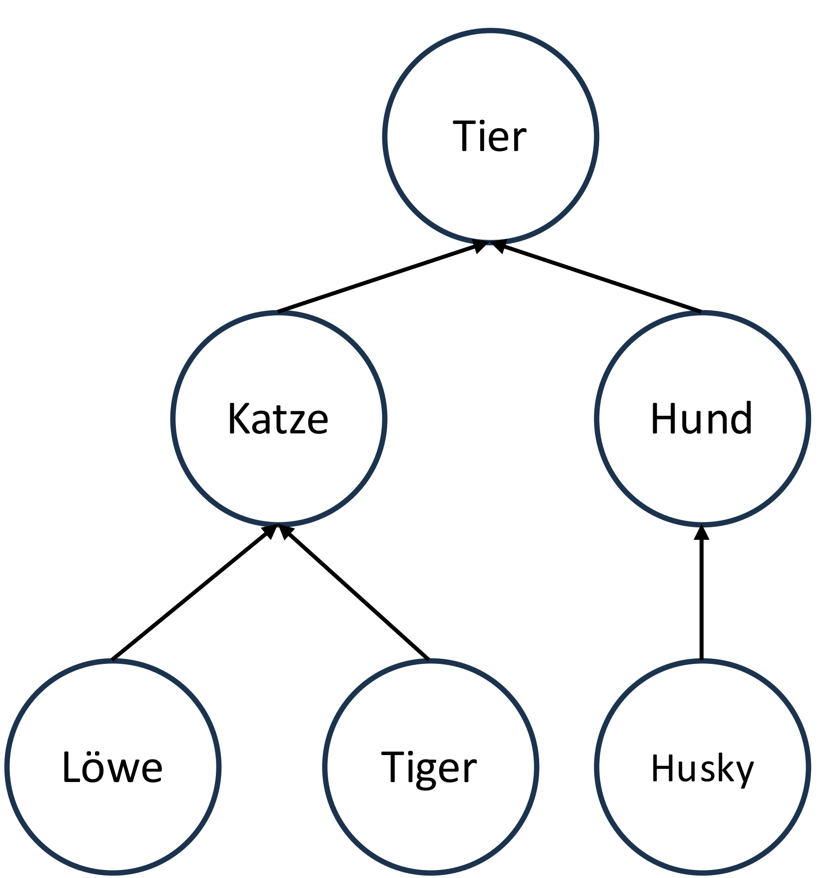


Wie unterscheiden wir zwischen:

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

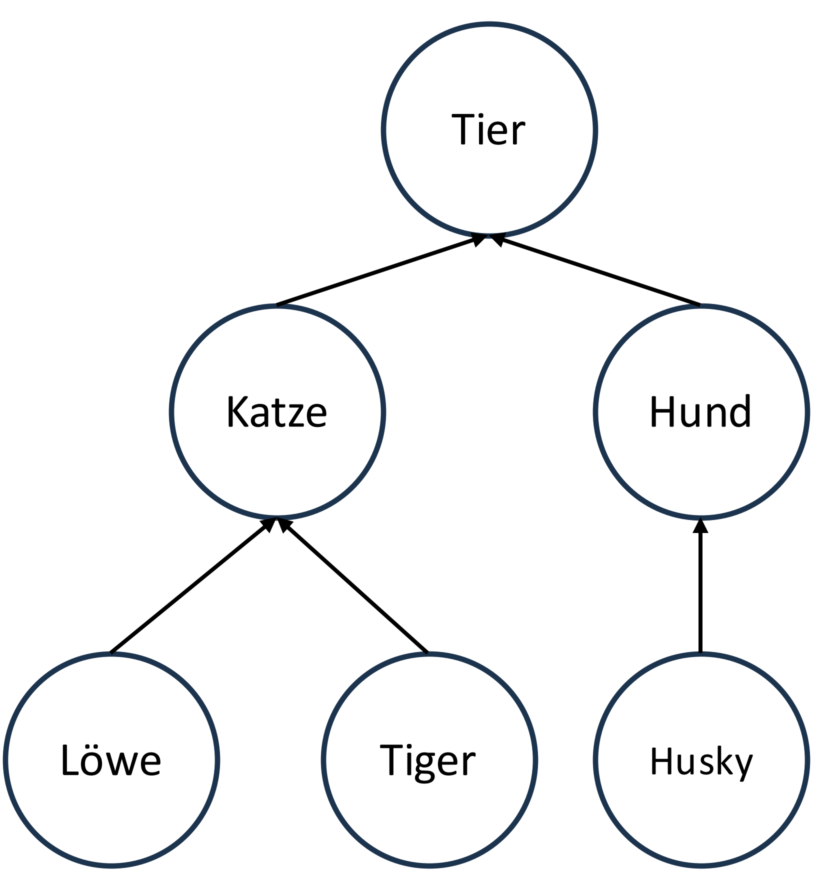
```
Tier hd = new Husky()
```

Misconception: Objekte



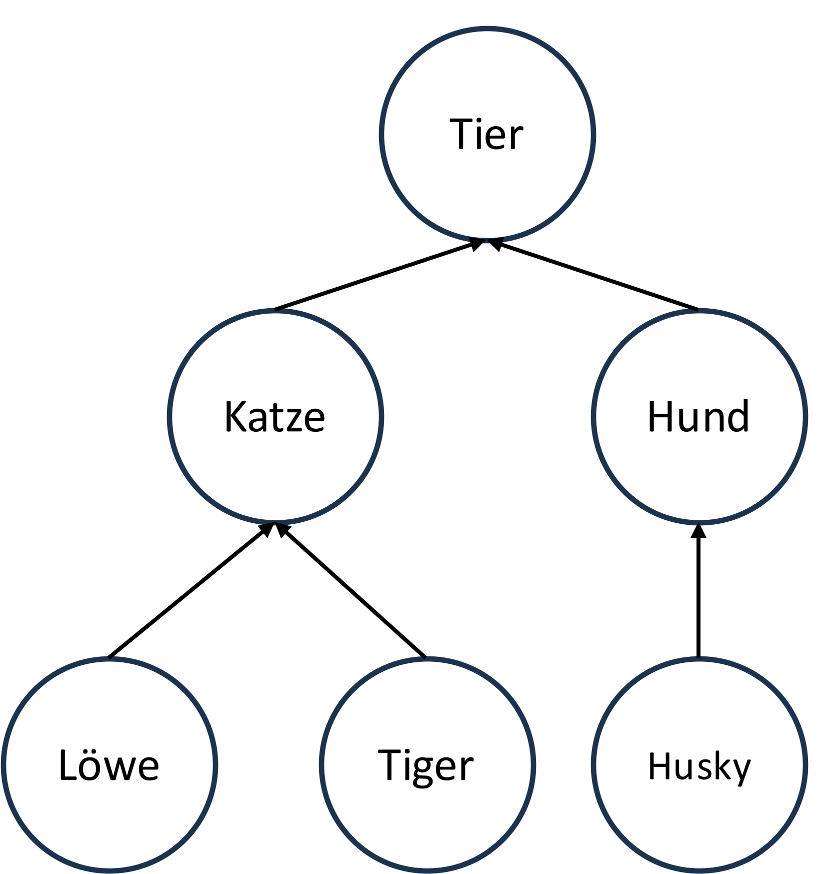
Ein Objekt der Subklasse ist **immer** auch vom Typ der Superklasse. (ist-ein Beziehung)

Misconception: Objekte



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

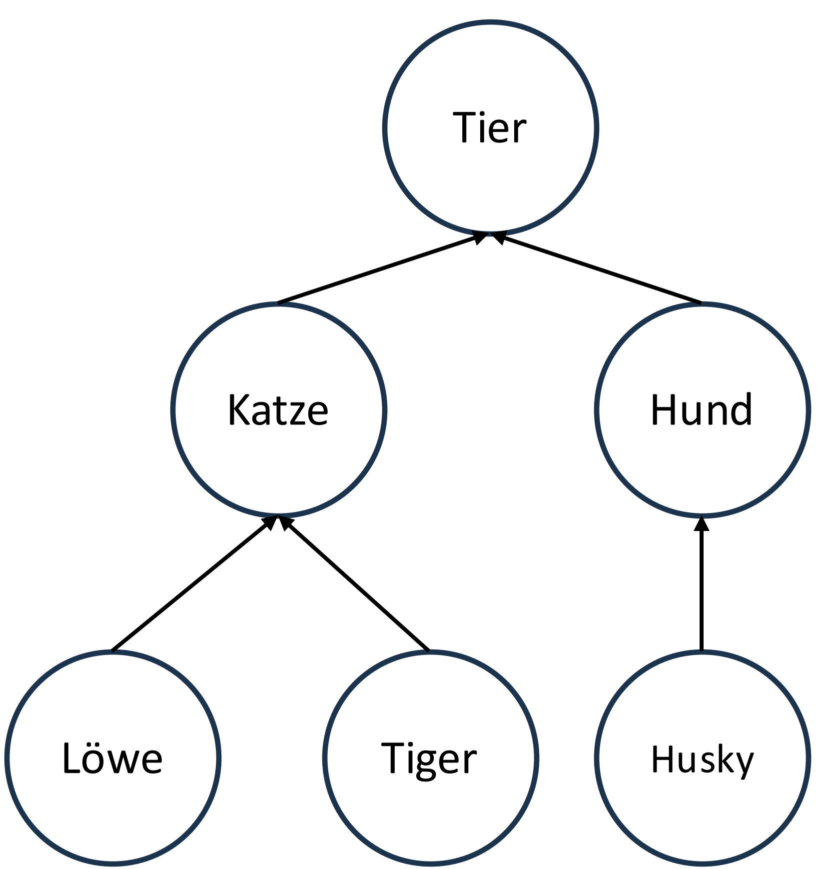
Was wirklich passiert



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

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

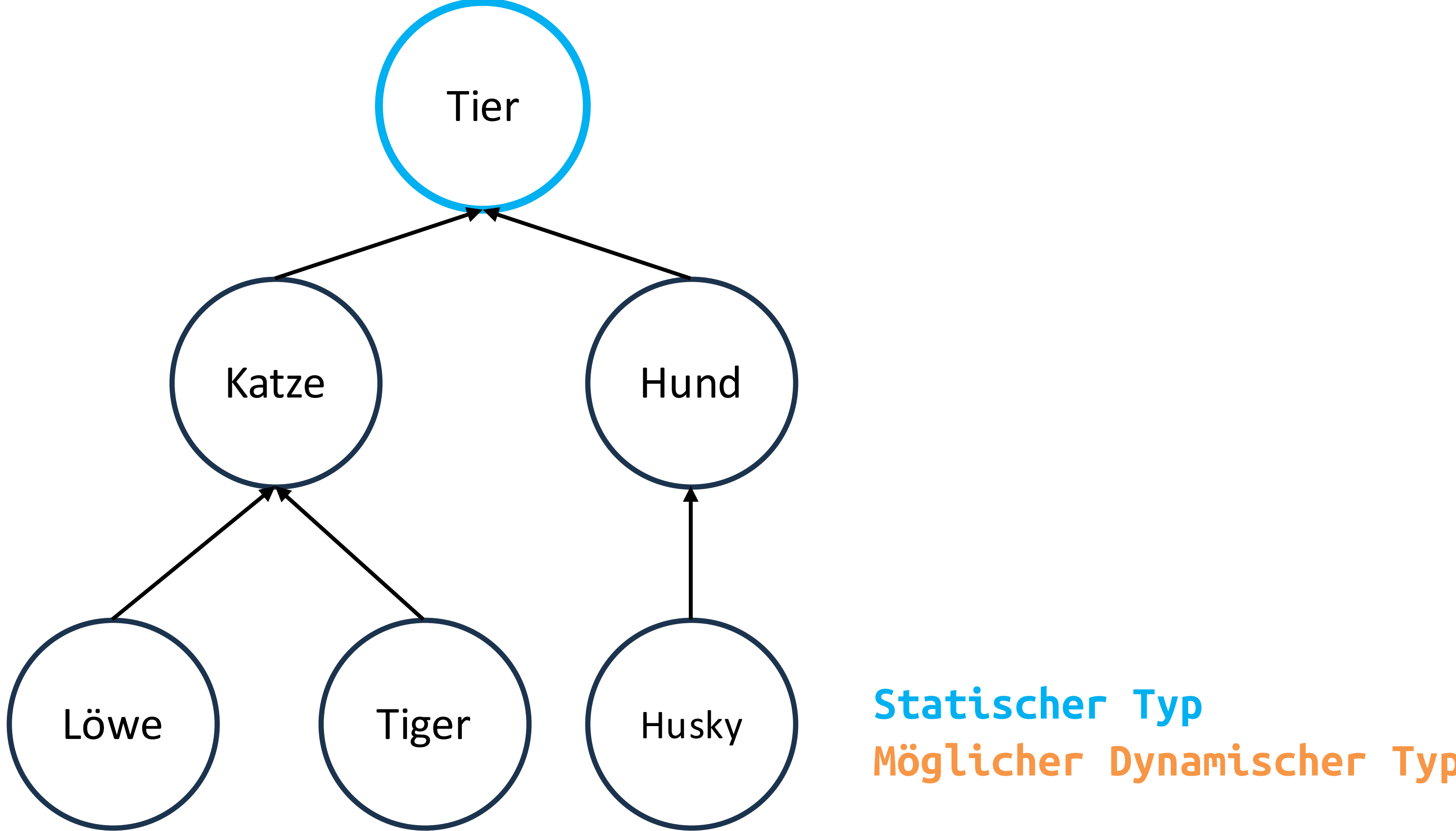
Was wirklich passiert



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

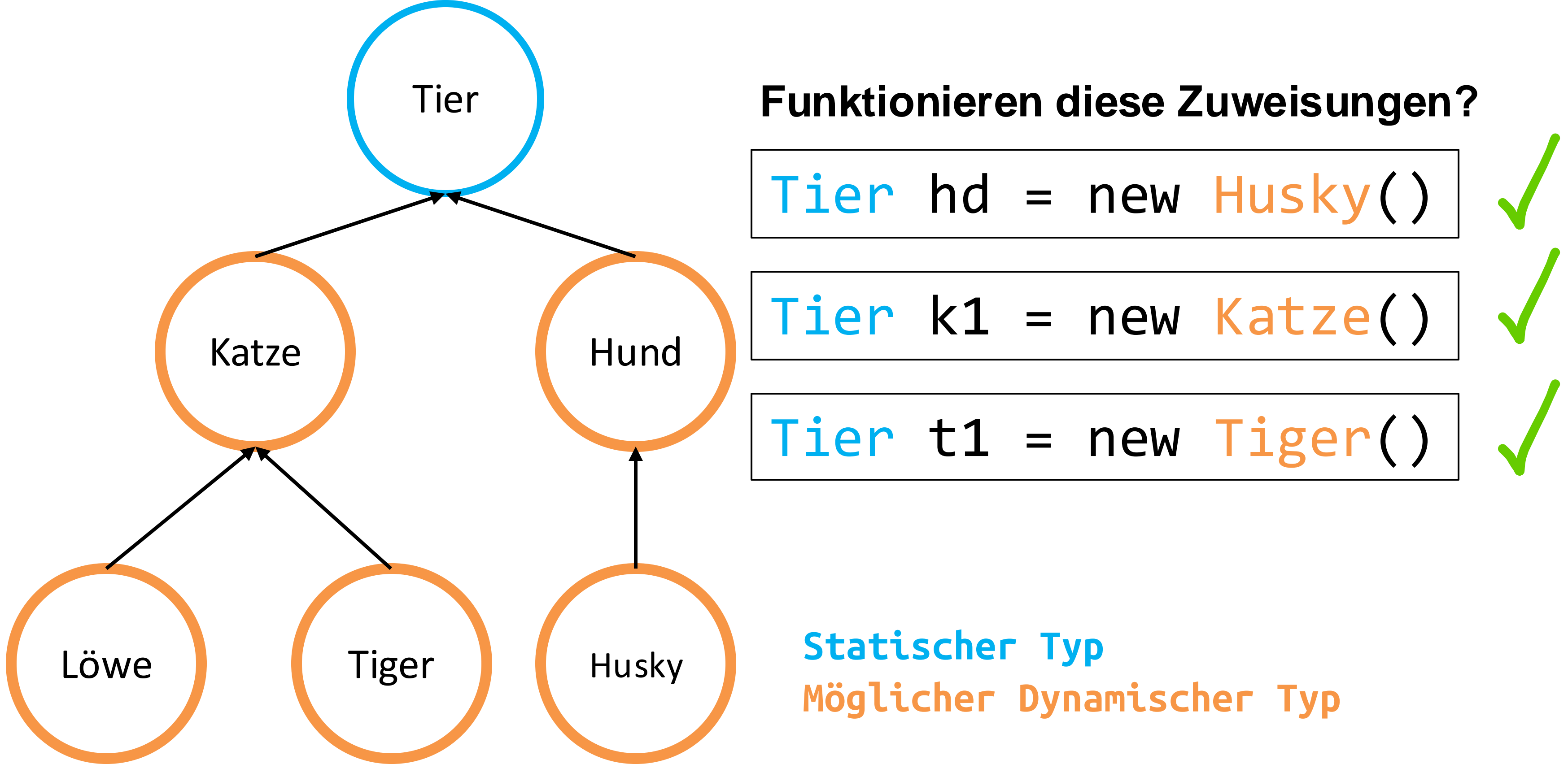
```
Husky hd = new Tier()
```

Verdeutlicht – Was für Typen akzeptiert eine Variable?

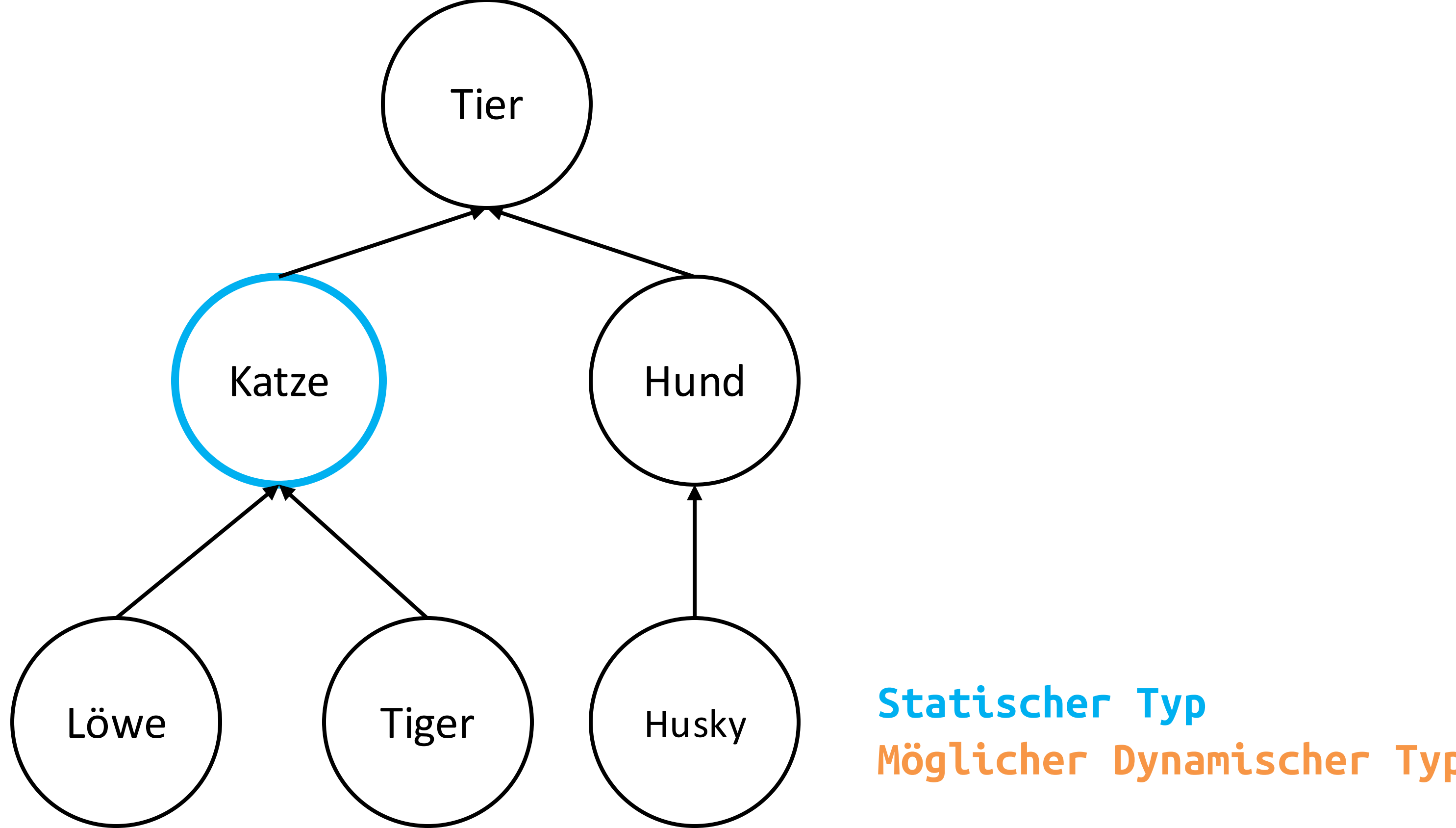


Statischer Typ
Möglicher Dynamischer Typ

Verdeutlicht – Was für Typen akzeptiert eine Variable?

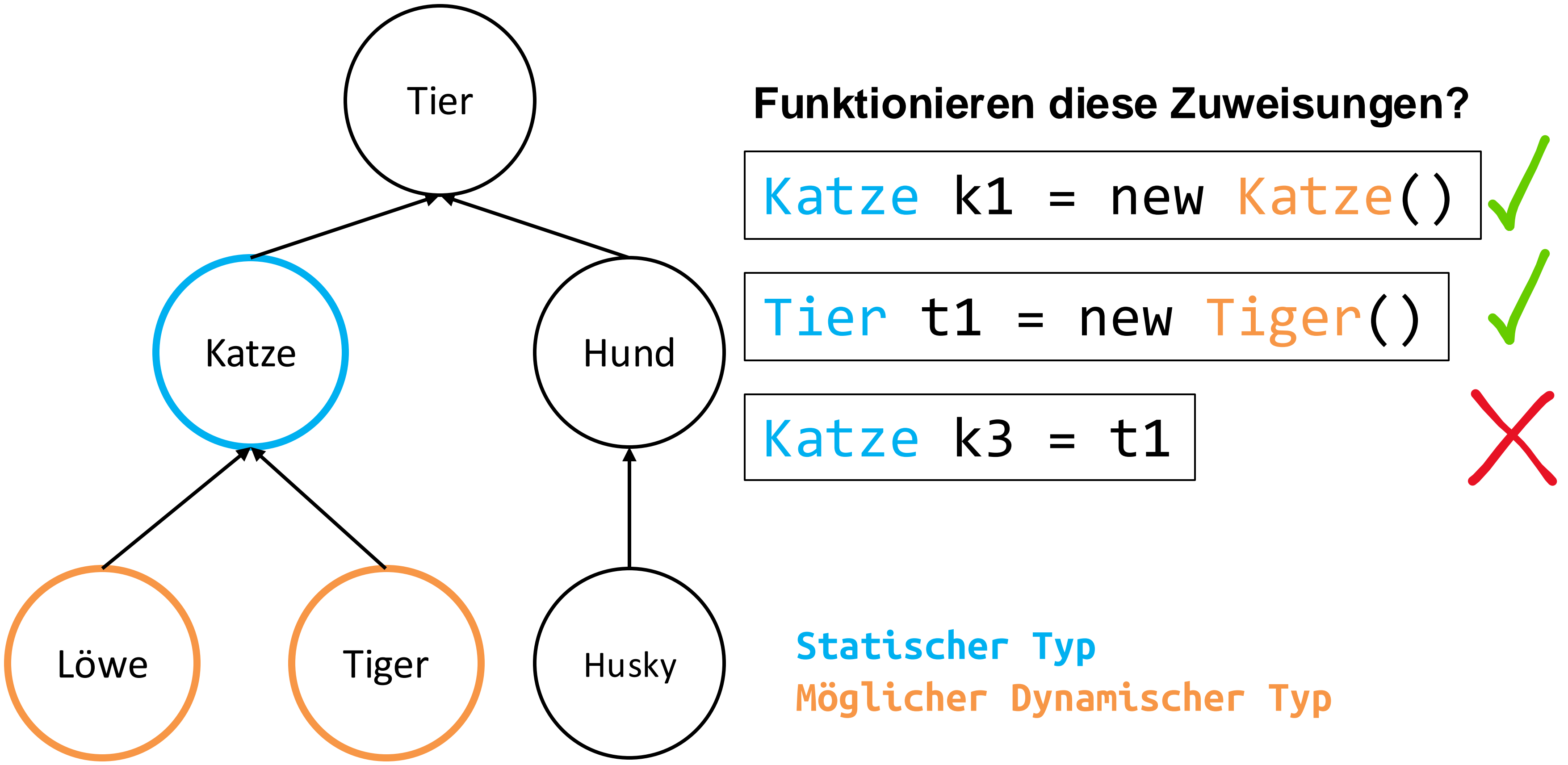


Verdeutlicht – Was für Typen akzeptiert eine Variable?

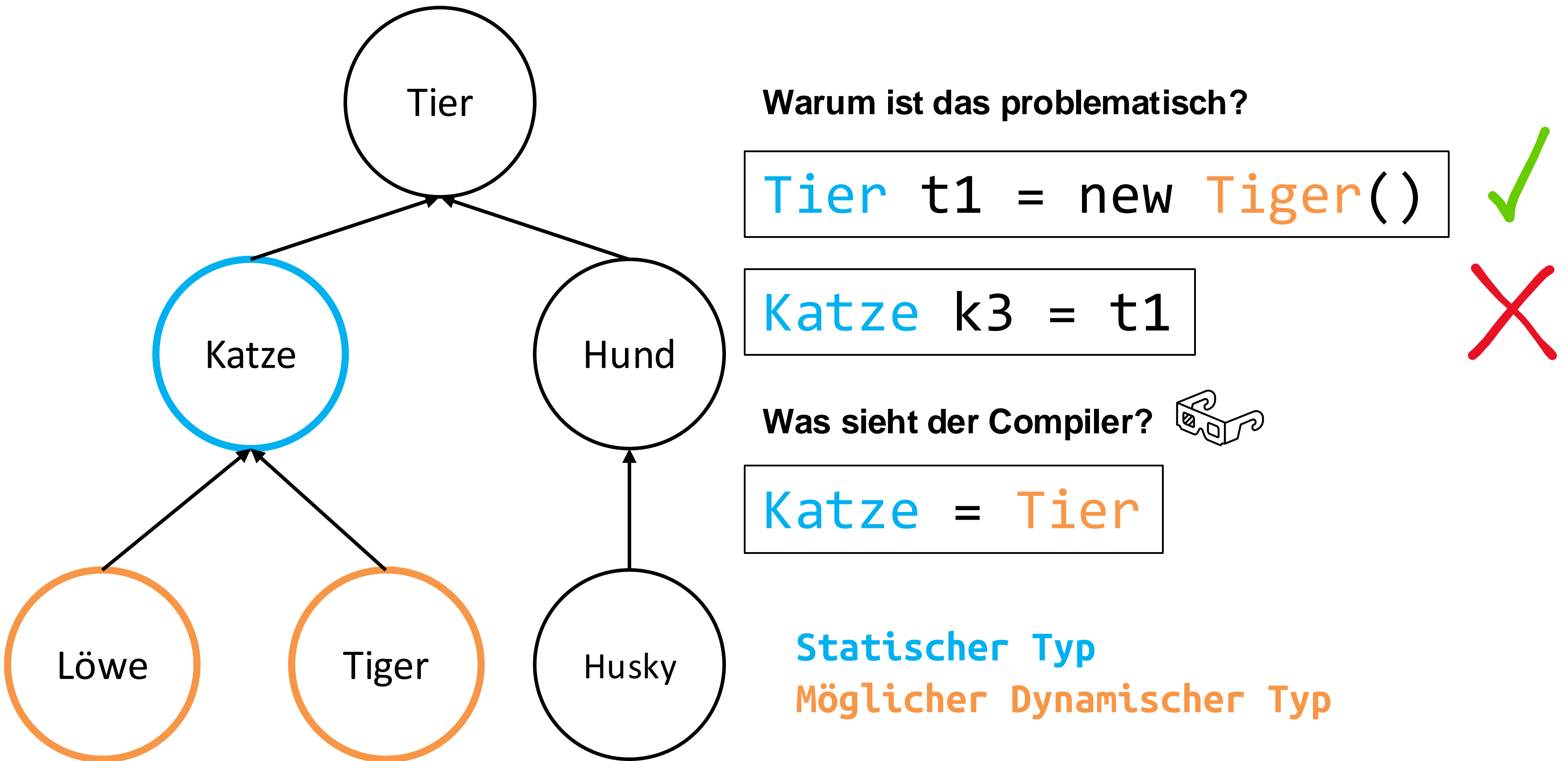


Statischer Typ
Möglicher Dynamischer Typ

Verdeutlicht – Was für Typen akzeptiert eine Variable?



Verdeutlicht – Was für Typen akzeptiert eine Variable?



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…

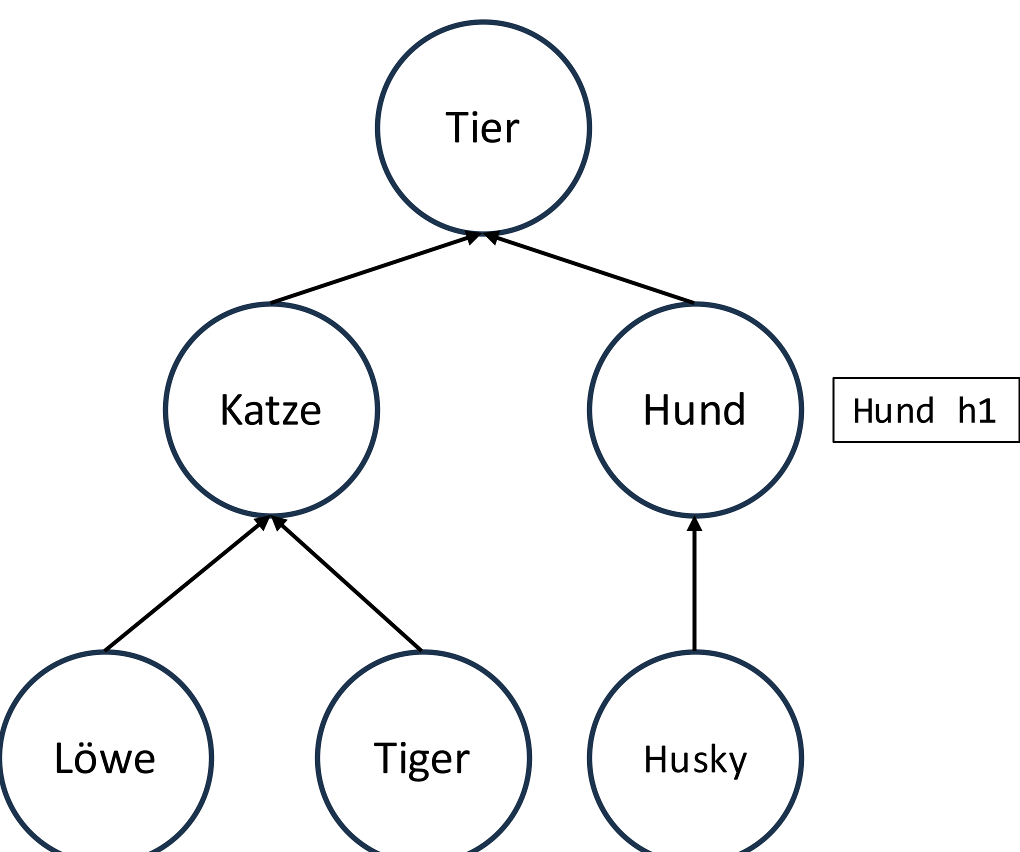
Wir müssen dem Compiler garantieren können, dass er hier immer eine Subklasse von Katze bekommt, damit die Zuweisung legal ist.

Intuition

```
Tier tier = zufallsTier();
// Compiler denkt sich: Ich kann keine Aussage machen, ob das richtig oder falsch ist.
Katze katze = tier;

public Tier zufallsTier() {
    return Math.random() > 0.5 ? new Katze() : new Tiger();
}
```

Casting

Casting




```
Hund h1;
```

Casting

```
Hund h1;
```

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

Tier

Tier h2

Katze

Hund

Hund h1

new Hund()

Löwe

Tiger

Husky

Casting

Tier

Katze

Hund

Löwe

Tiger

Husky

`Tier h2`

`Hund h1`

`new Hund()`

```
Hund h1;
```

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

```
h1 = h2;
```

Casting

Tier

Katze

Hund

Löwe

Tiger

Husky

`Tier h2`

`Hund h1`

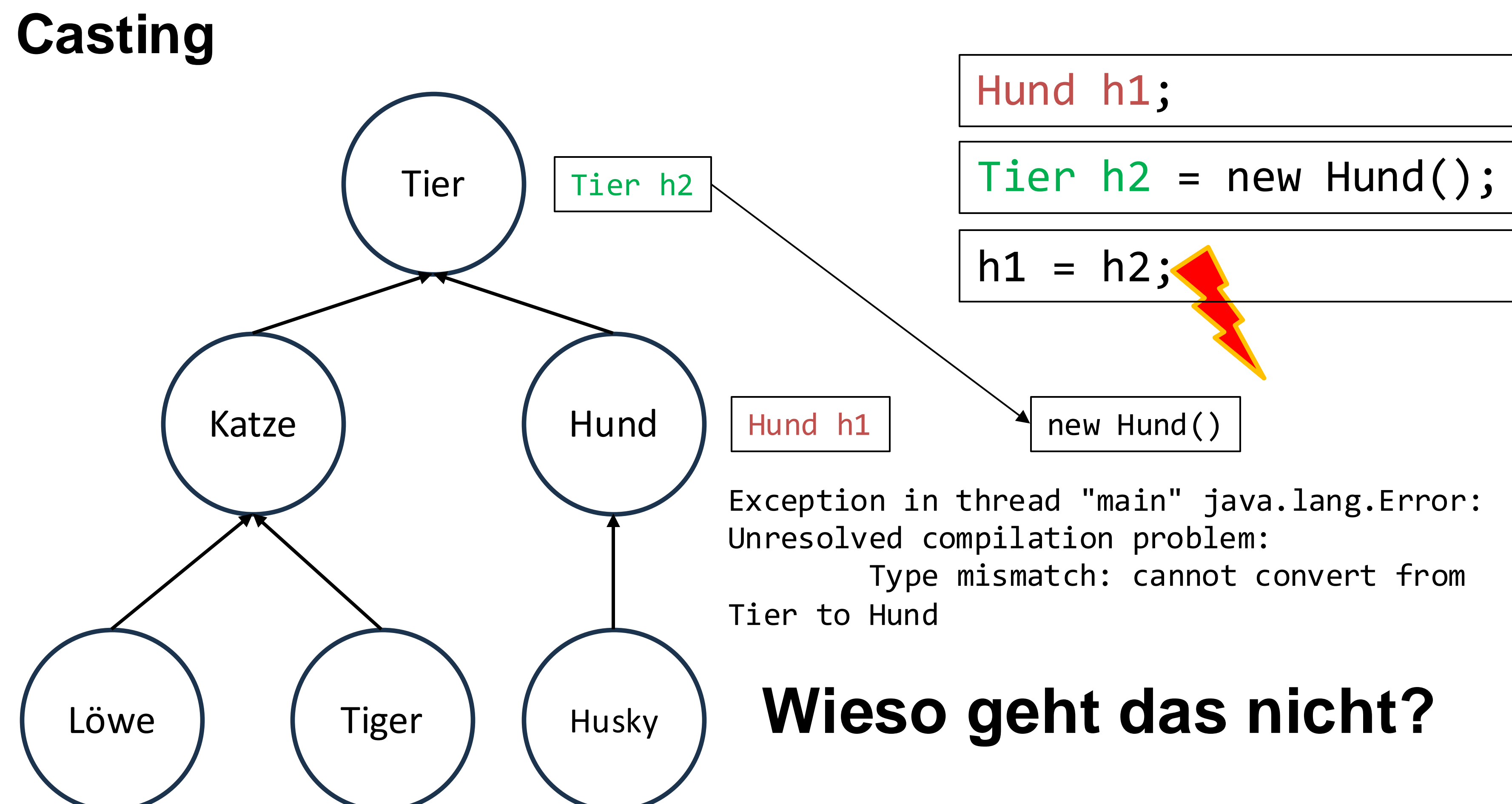
`new Hund()`

```
Hund h1;
```

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

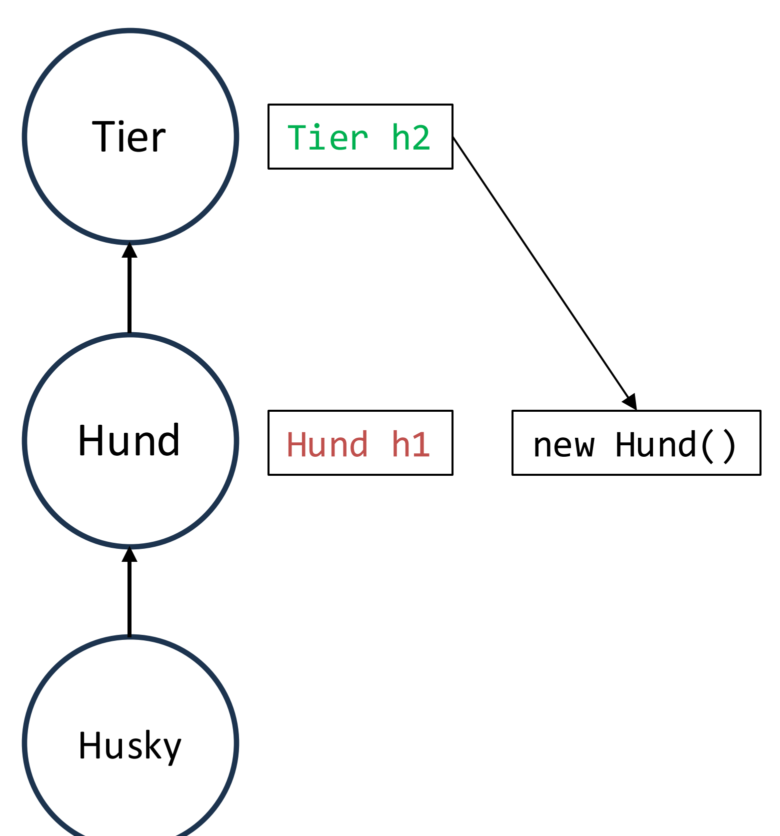
```
h1 = h2;
```

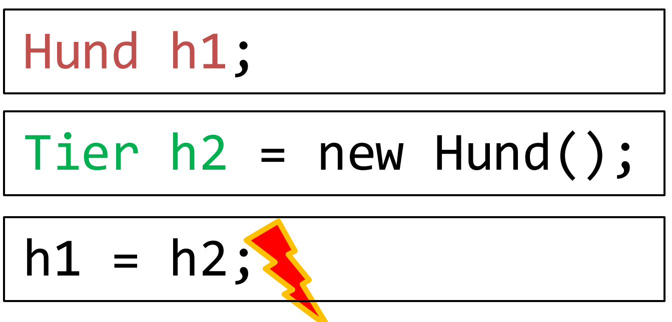
Casting



Wieso geht das nicht?

Casting



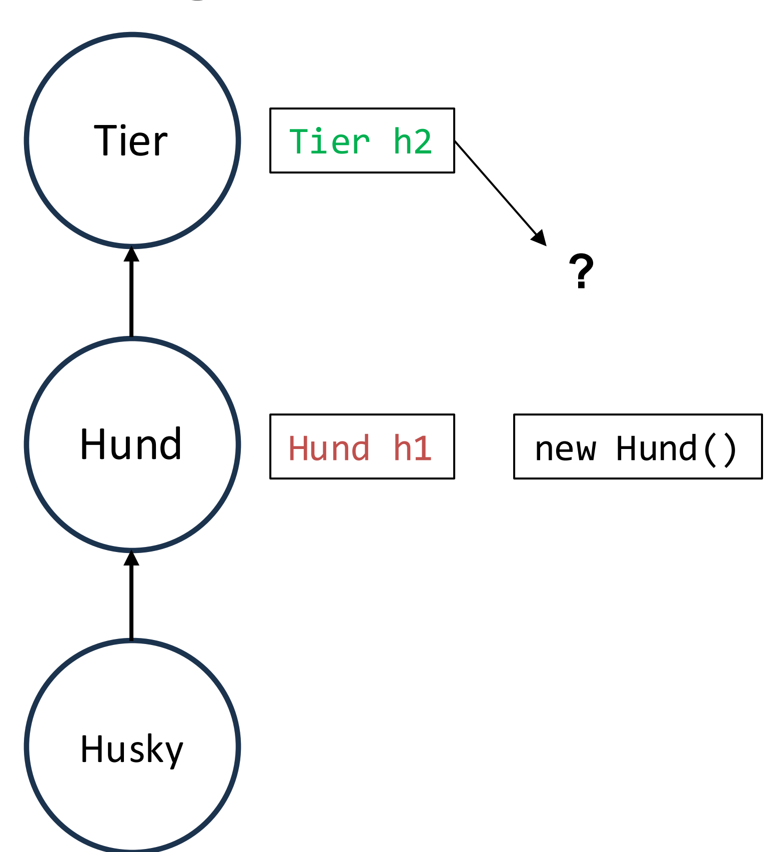


Wieso geht das nicht?

Von wo wissen wir, dass `h2` eine Referenz auf ein Objekt vom Typ `Hund` enthält?

Wir kennen den dynamischen Typ…

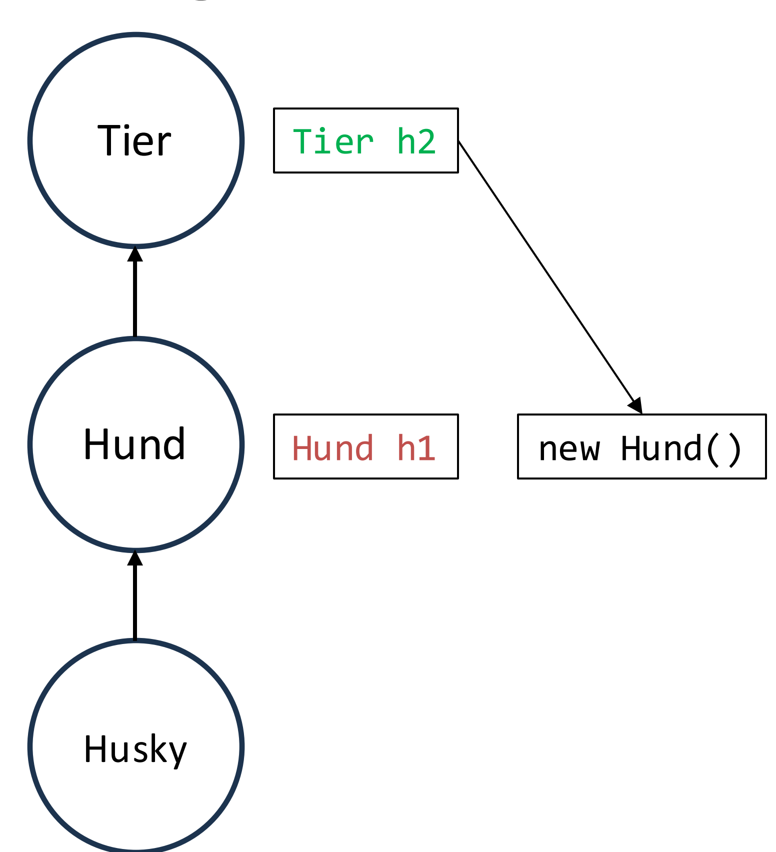
Casting




```
void methode1(Tier h2) {

    Hund h1;

    h1 = h2;
}
```

Hier kennen wir den dynamischen Typ nicht…

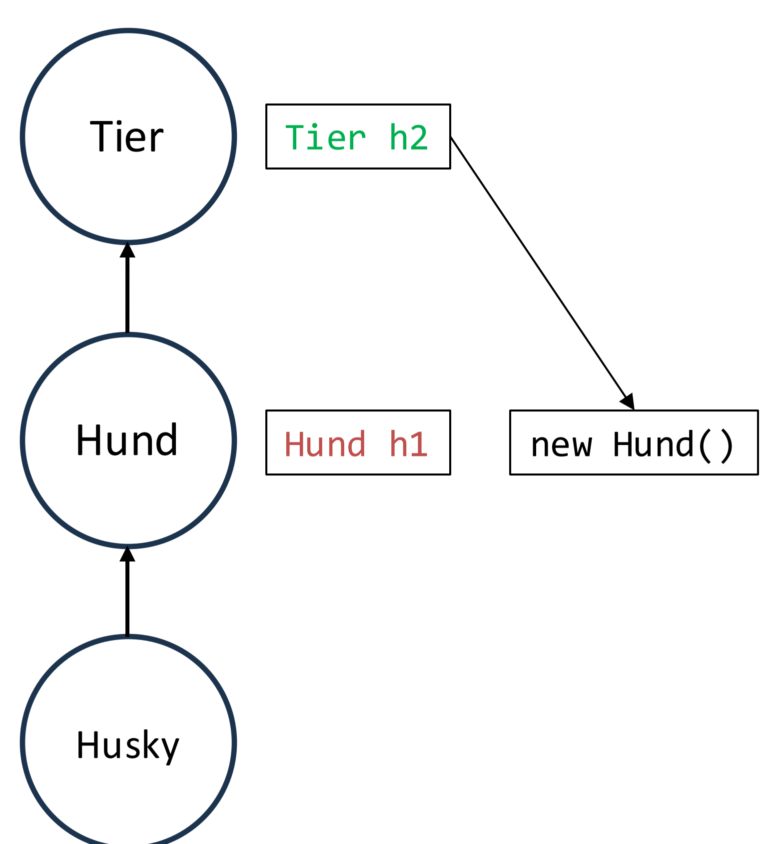
Casting




```
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

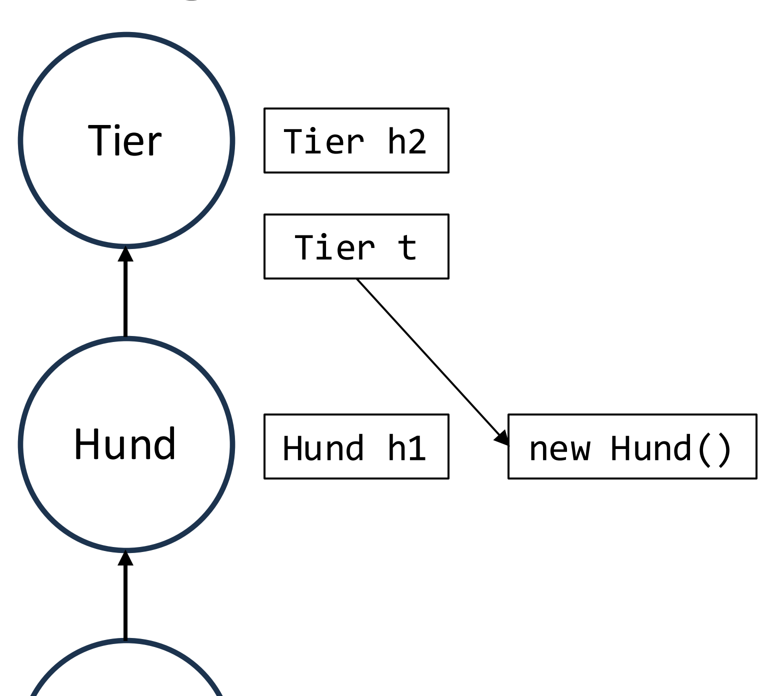



```
void methode1(Tier h2) {

    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

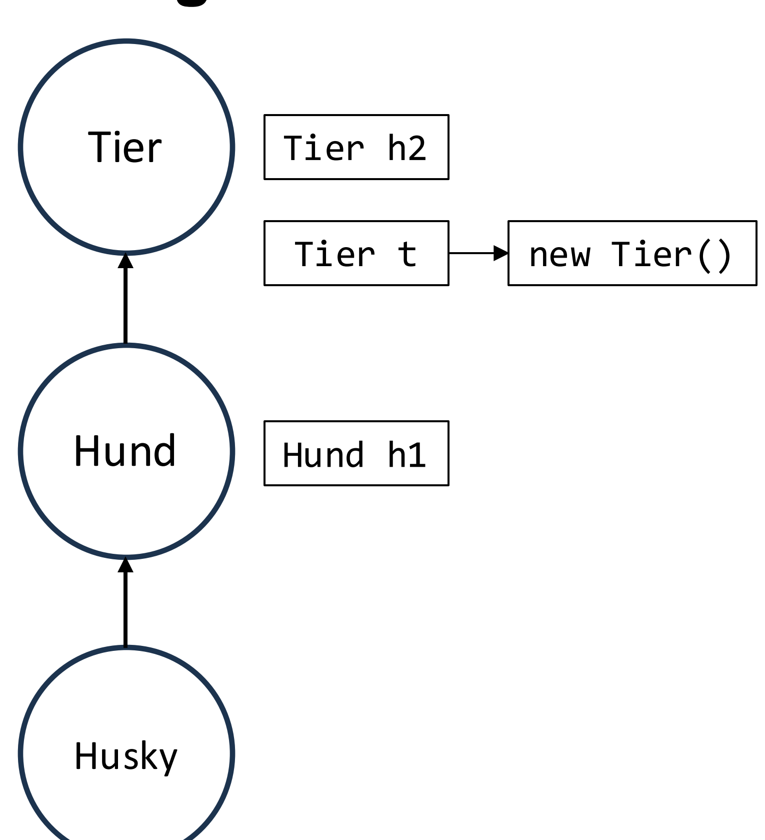



```
void methode1(Tier h2) {
    Hund h1;
    h1 = (Hund) h2;
}

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

Geht das? ✓

Diese Garantie, die Einschränkung auf einen typen weiter unten im Baum nennt man einen Downcast (oder Narrowing).

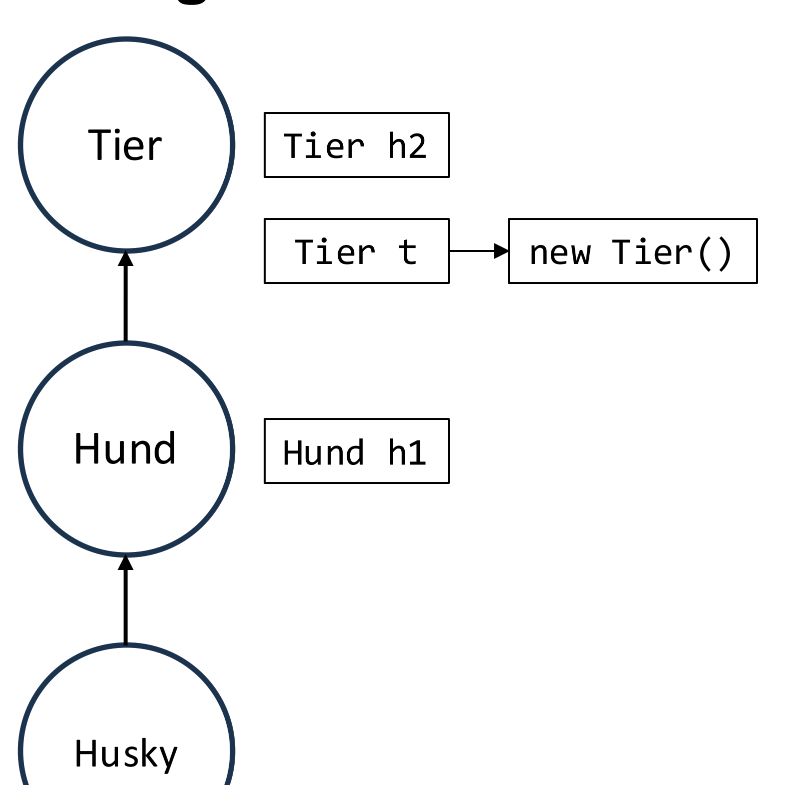
Casting




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

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

Geht das?

Casting

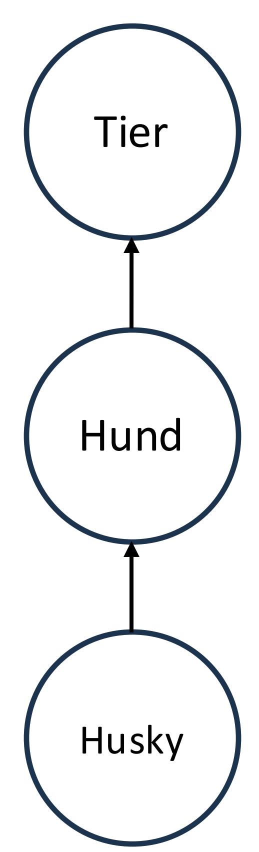



```
void methode1(Tier h2) {
    Hund h1;
    h1 = (Hund) h2;
}

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

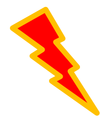
Geht das? ✗

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

Casting: Laufzeitfehler vs Compiler Fehler



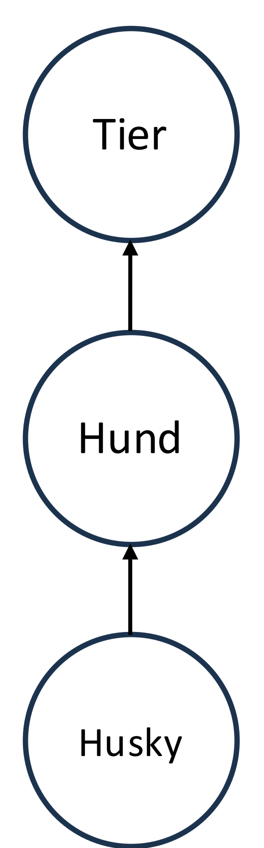

```
Hund h = (Hund) new Scanner();
```



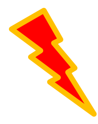
```
Exception in thread "main" java.lang.Error: Unresolved
compilation problem:
        Type mismatch: cannot convert from Tier to Hund
```

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

Casting: Laufzeitfehler vs Compiler Fehler




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

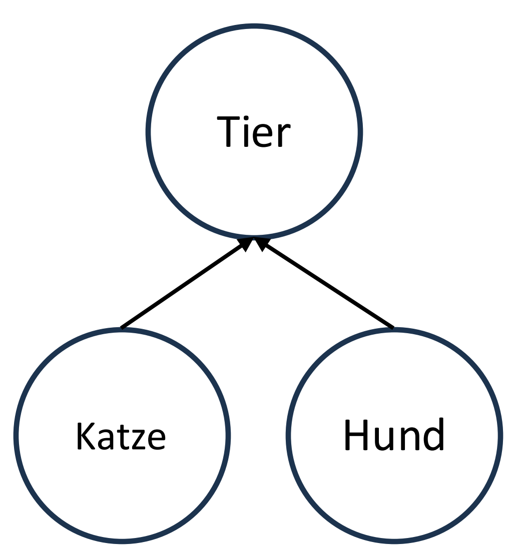


```
Exception in thread "main" java.lang.ClassCastException:
class Tier cannot be cast to class Hund
```

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

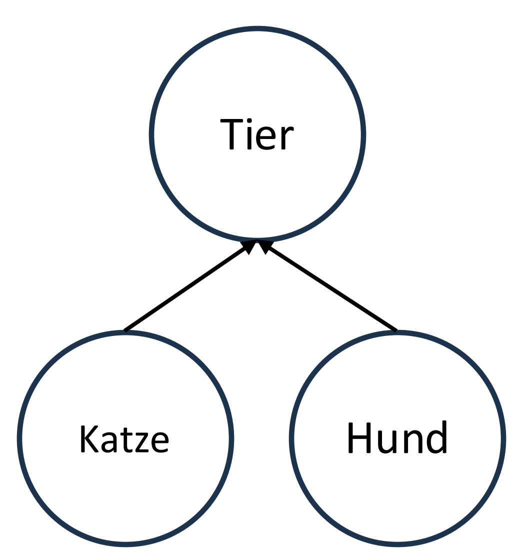
- Beim Ausführen gibt es einen Laufzeitfehler.
- Denkt an Intuition von vorher

Casting: Laufzeitfehler vs Compiler Fehler




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

h = k;
```

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
```

“Formale” Intuition

Wir nehmen an: A <: B und weder A <: C noch C <: A

Statement	Compile Error / Runtime Exception
B b = new A()	Nein / Nein
A a = (A) new B()	Nein / Ja (Grund: Incompatible Types)
A a = new B()	Ja / - (Grund: Cast / Versprechen an Compiler notwendig)
C c = (C) new A();	Ja / - (Grund: Inconvertible Types)
A a = (A) new C();	Ja / -

Wir nehmen nun an dass zur Variable a mit statischem Type A einer Variable b mit statischem Type B zu-gewiesen wird. Muss ein Cast gemacht werden?
Falls B <: A (nie, weil das Objekt (Objekt ist Instanz von Reference Type C) in b immer Subtype von B ist)
Falls A <: B und A != B (immer, Cast zu A muss gemacht werden, kann aber zu ClassCastException führen)
Weder A <: B noch B <: A (nie, führt aber immer zu Compile Error)

ClassCastException vermeiden

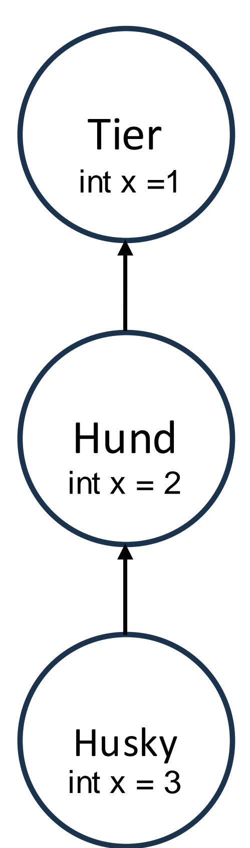
- **Rettung: instanceof Keyword**

```
1  null instanceof A  // immer false
2  A instanceof B // true, wenn A <: B
3  A instanceof B // false, wenn B <: A und B != A
4  A instanceof B // Compile Error, wenn weder A <: B noch B <: A
```

B muss ein Reference Type sein
A muss ein Objekt von einem Reference Type sein (keine Primitives)
Compile Error genau dann, wenn B b = (B) A; zu einem Compile Error führen würde

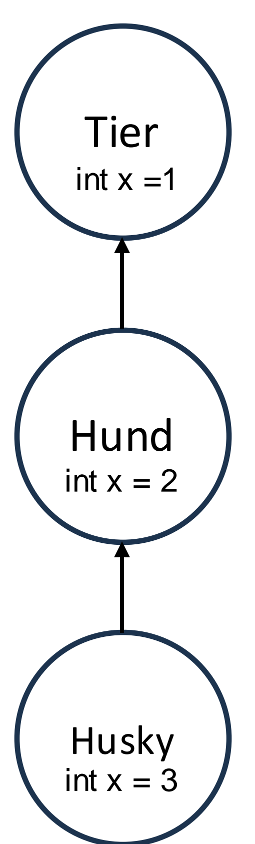
Attributwahl

Attribute:



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

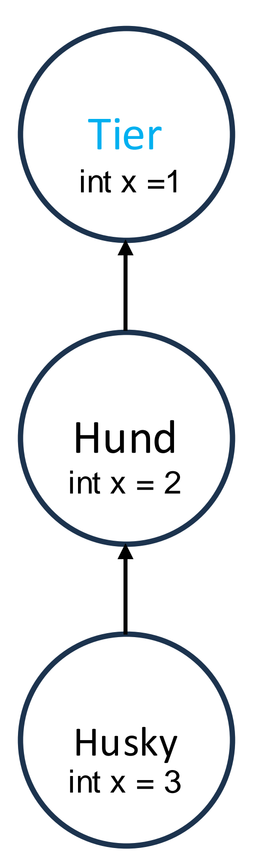
Attribute:



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

```
Tier t = new Tier();

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

Attribute:

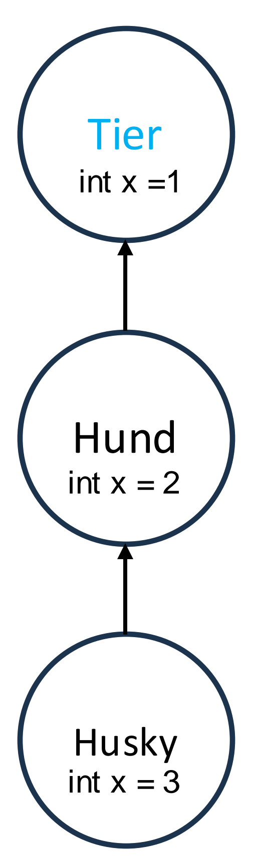


Regel: Attribute werden an Hand vom **statischen Typ** ausgewählt.

```
Tier t = new Tier();

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

Resultat: 1

Attribute:

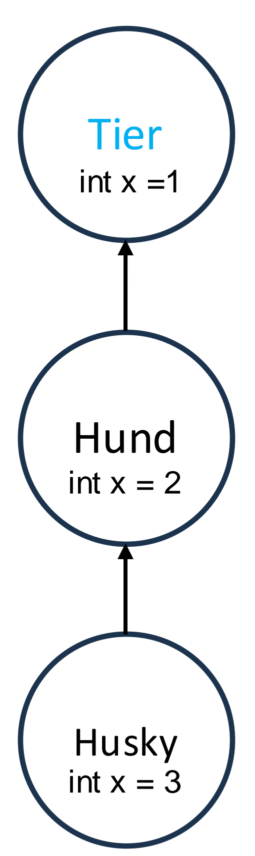


Regel: Attribute werden an Hand vom **statischen Typ** ausgewählt.

```
Tier t = new Hund();

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

Resultat: 1

Attribute:

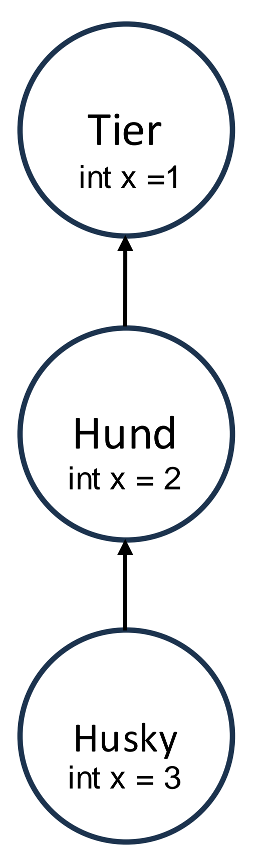


Regel: Attribute werden an Hand vom **statischen Typ** ausgewählt.

```
Tier t = new Husky();

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

Resultat: 1

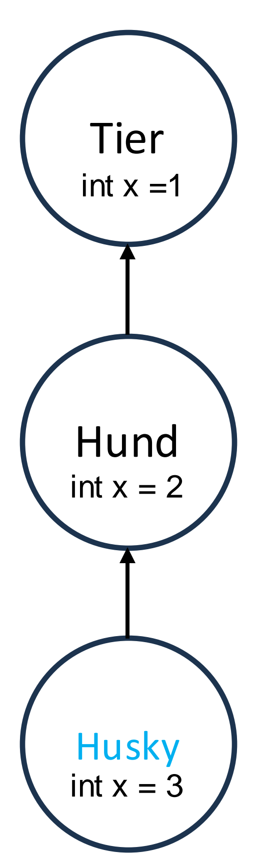
Attribute:



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

```
Husky t = new Husky();

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

Attribute:

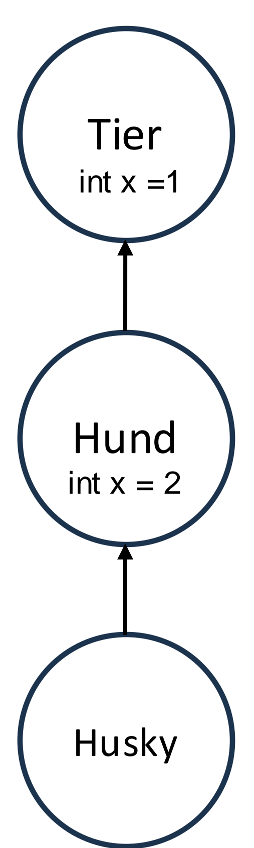


Regel: Attribute werden an Hand vom **statischen Typ** ausgewählt.

```
Husky t = new Husky();

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

Resultat: 3

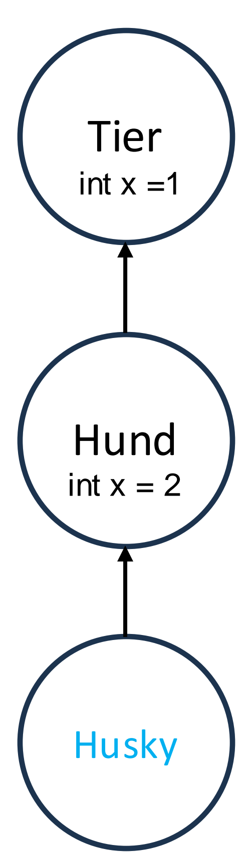
Attribute:



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

```
Husky t = new Husky();

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

Attribute:



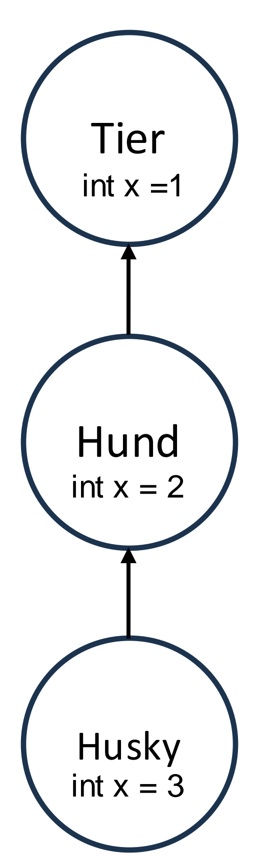
Regel: Attribute werden an Hand vom **statischen Typ** ausgewählt.

```
Husky t = new Husky();

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

Resultat: 2

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

Attribute:

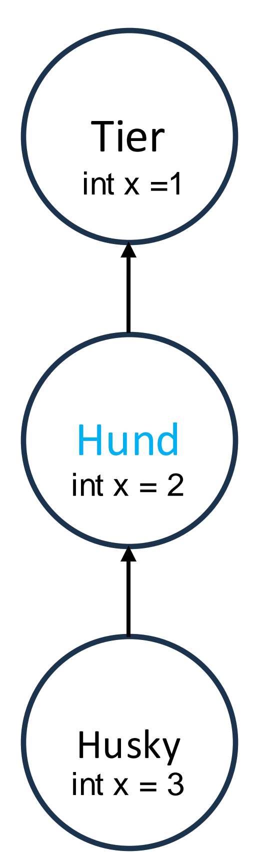


Regel: Attribute werden an Hand vom **statischen Typ** ausgewählt.

```
Husky t = new Husky();

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

Resultat: 2

Attribute:



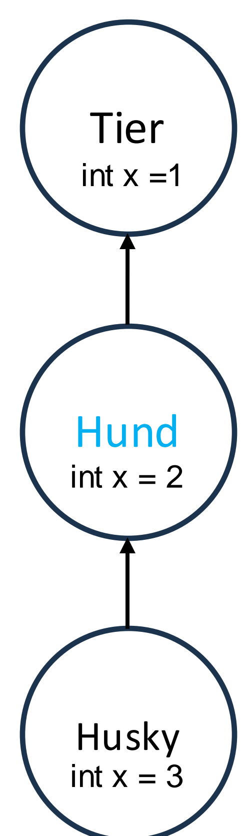
Regel: Attribute werden an Hand vom **statischen Typ** ausgewählt.

```
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:



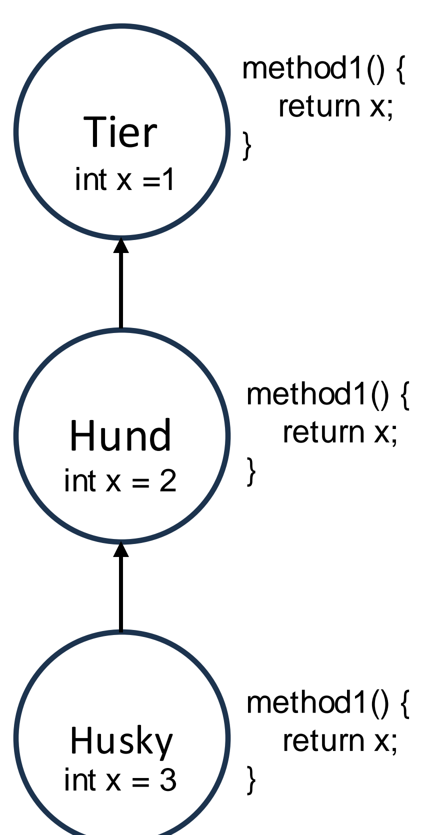

```
Husky t = new Husky();

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:



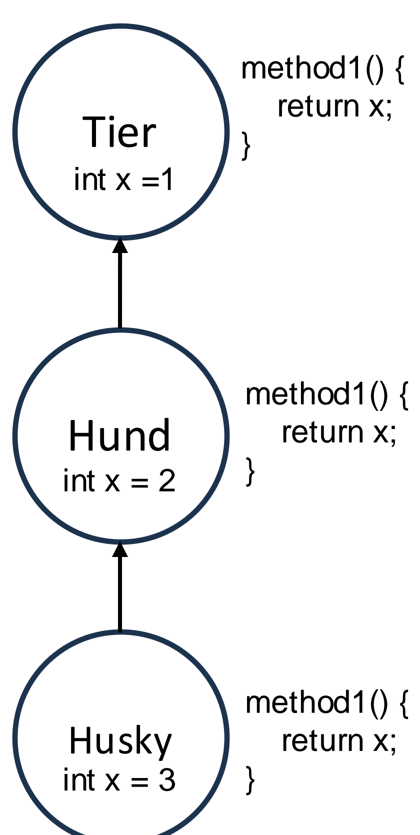
Regel: Methoden* werden an Hand vom **dynamischen Typ** ausgewählt.

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

```
Tier t = new Husky();

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

Resultat: 3

Methoden:



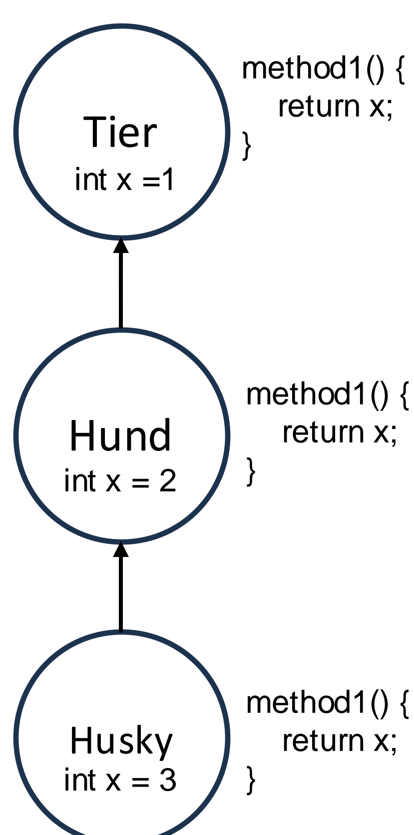
Regel: Methoden* werden an Hand vom **dynamischen Typ** ausgewählt.

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

```
Hund t = new Husky();

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

Resultat: 3

Methoden:



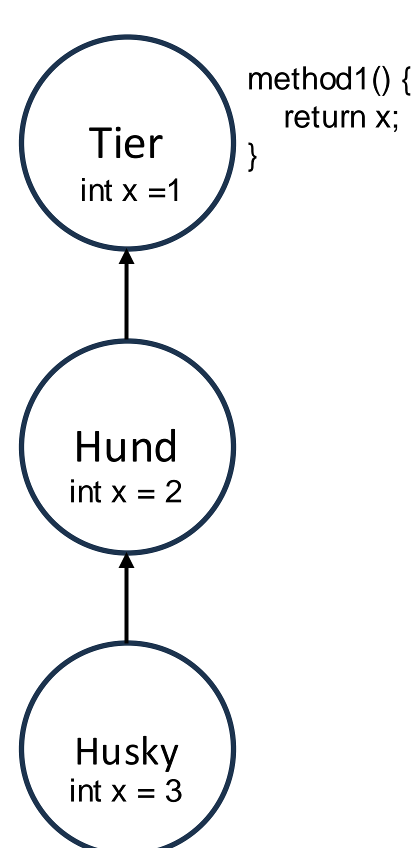
Regel: Methoden* werden an Hand vom **dynamischen Typ** ausgewählt.

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

```
Hund t = new Hund();

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

Resultat: 2

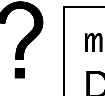
Methoden:



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

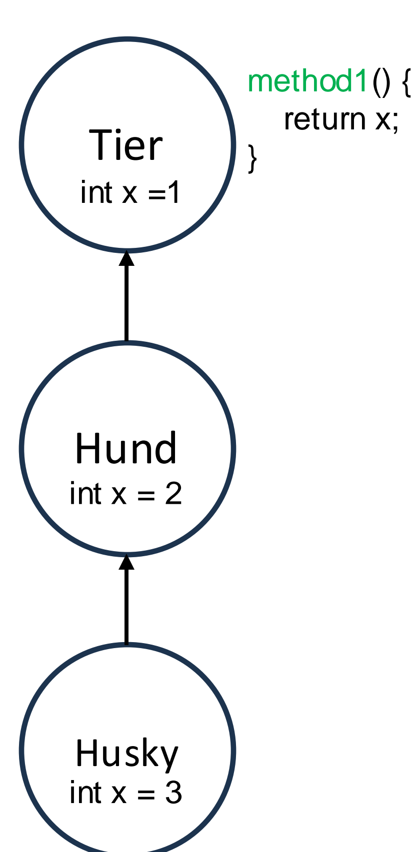
* ausser private, static und final Methoden (hier statischer Typ)

```
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.

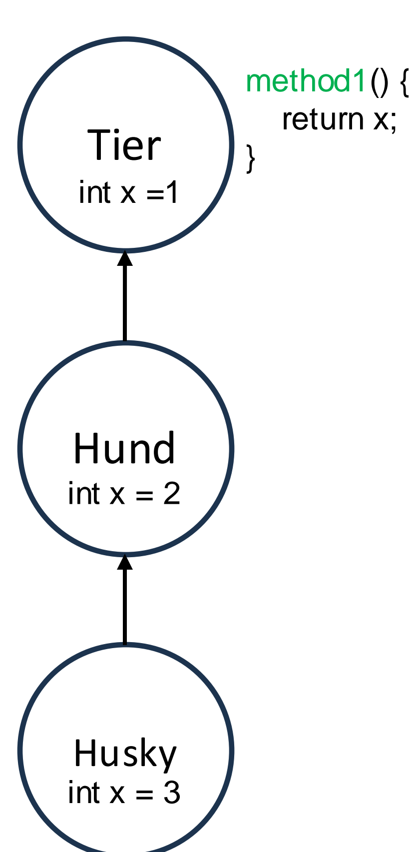
* ausser private, static und final Methoden (hier statischer Typ)

```
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.

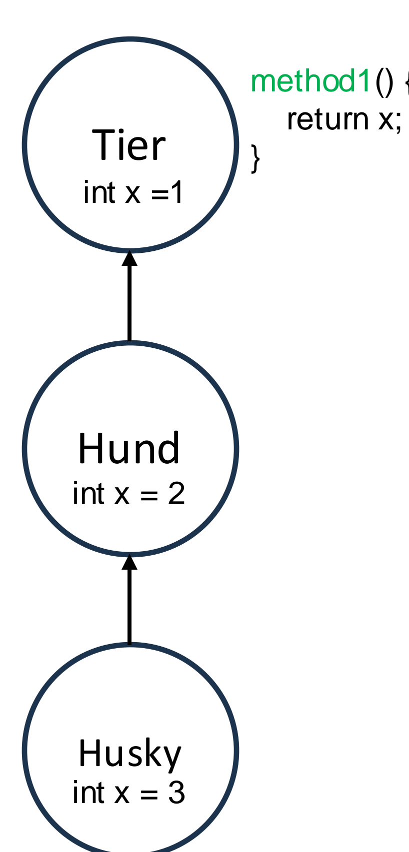
* ausser private, static und final Methoden (hier statischer Typ)

```
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.

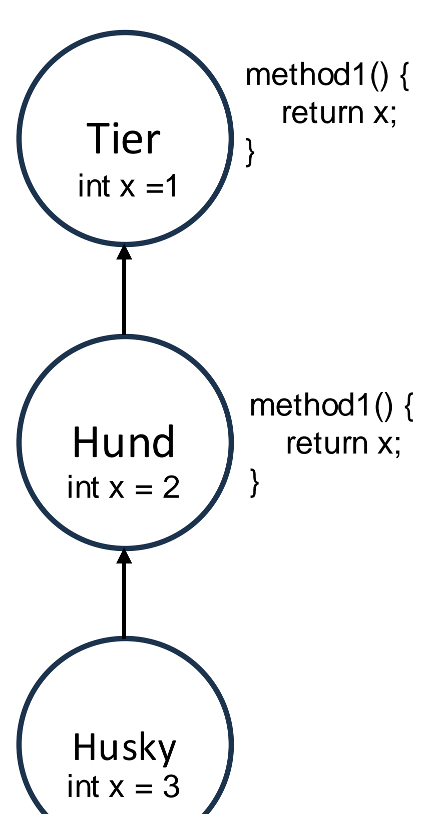
* ausser private, static und final Methoden (hier statischer Typ)

`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:



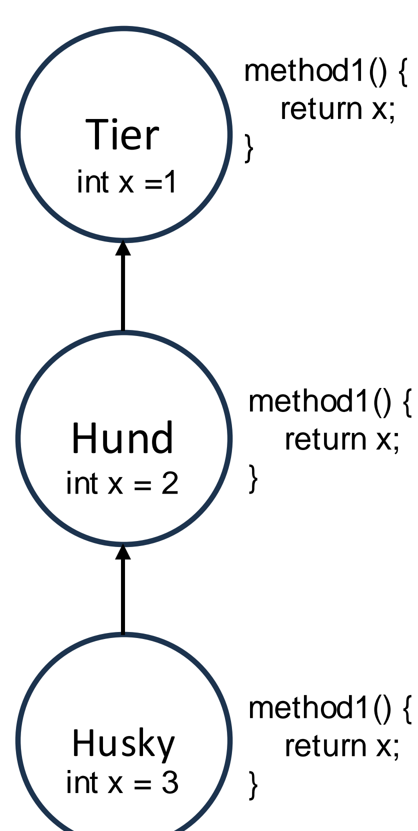
Regel: Methoden* werden an Hand vom **dynamischen** **Typ** ausgewählt.

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

```
Tier t = new Husky();

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

Resultat: 2

Methoden:



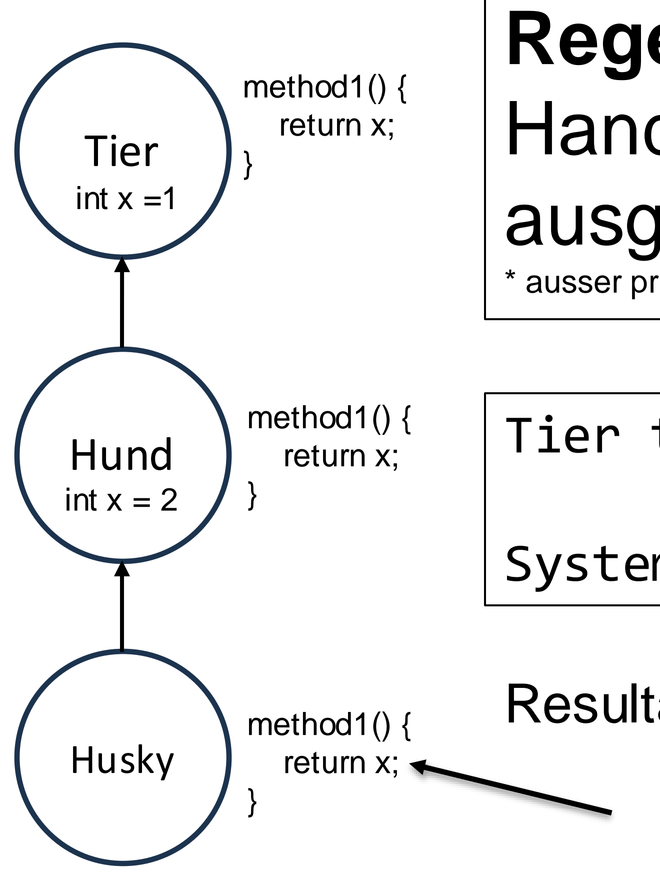
Regel: Methoden* werden an Hand vom **dynamischen Typ** ausgewählt.

* 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.

this und super

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

Keywords bei Objekten

- **`this` ist eine Referenz auf das Objekt, aus dem wir gerade arbeiten und kann auch ganz normal als Objektreferenz behandelt werden**
  - `this(…)`
    - Ruft den passenden Konstruktor auf
  - `this.field`
    - Gibt das angesprochene Objektattribut
  - `this.someMethod()`
    - Ruft die angesprochene Methode auf
  - `otherMethod(this, 7)`
    - Übergibt die Referenz auf das aktuelle Objekt

Problem Solving: Klassen

Hogwarts (2020 W11)

In dieser Aufgabe implementieren Sie das Punktesystem von Hogwarts, bei welchem Studenten eines Hauses Punkte verliehen oder abgezogen bekommen können und dadurch der kumulative Punktestand ihres Hauses sich verändert. Wir verwenden drei Klassen, School, House, und Student, für die Schule, Häuser, und Studenten

Die Klassen können folgendermassen verwendet werden:

```
School hogwarts = new School();

// Haeuser werden erstellt (Anzahl Haeuser und Namen sind nicht eingeschraenkt).
House hufflepuff = hogwarts.createHouse("Hufflepuff");
House ravenclaw  = hogwarts.createHouse("Ravenclaw");

// Studenten haben Vor- und Nachnamen.
Student hannah = new Student("Hannah", "Abbott");
Student newton = new Student("Newton", "Scamander");
Student luna   = new Student("Luna",   "Lovegood");
Student filius = new Student("Filius", "Flitwick");

// Studenten werden den Haeusern zugeordnet.
hufflepuff.assign(hannah);
hufflepuff.assign(newton);
ravenclaw.assign(luna);
ravenclaw.assign(filius);

// Punkte werden an Studenten vergeben. Punkte koennen auch negativ sein.
hannah.givePoints(10);
newton.givePoints(-5);
luna.givePoints(8);

// Informationen zu der Summe an Punkten und dem aktuellen Siegerhaus
// koennen immer abgefragt werden.
System.out.println("Siegerhaus:␣" + hogwarts.winner().name());
System.out.println("Siegerpunkte:␣" + hogwarts.winner().points());
System.out.println("Hogwarts␣Punkte␣Insgesamt:␣" + hogwarts.points());
```

1. Implementieren Sie den School-Konstruktor und die Methode createHouse(String name), welche als Parameter den gewünschten Namen des Hauses nimmt und ein House Objekt zurückgibt. Der Name eines Hauses darf nicht null sein oder bereits für die Schule vorhanden sein. Die Methode soll in diesen Fällen eine IllegalArgumentException werfen. Alle anderen Namen sind erlaubt. Implementieren Sie zusätzlich die Methode name() der Klasse House, welche den Namen des Hauses als String zurückgibt.

2. Implementieren Sie den Konstruktor von Student, welcher zwei Strings, den Vor- und Nachnamen (in dieser Reihenfolge) nimmt. Sie dürfen annehmen, dass es jeden Namen (Vor- und Nachname zusammen) nur einmal gibt. Vor- und Nachnamen sollen über die Methode firstName() beziehungsweise lastName() erhalten werden können. Implementieren Sie zusätzlich die Methode assign(Student student) der Klasse House, welche einen Studenten als Argument nimmt und ihn in dieses Haus einschreibt. Bei einem null Argument oder falls der Student bereits bei einem Haus der gleichen Schule eingeschrieben ist, dann soll die Methode eine IllegalArgumentException werfen.

Als letztes implementieren Sie das Punktesystem. Implementieren Sie dafür vier Methoden: Die Method points() von House gibt die Punkte eines Hauses zurück. Jedes Haus beginnt mit einem Punktestand von 0, wenn es erstellt wird. Dieser Punktestand kann sich dann durch die Leistungen der Studenten verändern. Die Methode givePoints(int points) von Student nimmt eine positive oder negative Anzahl Punkte, welche dem Studenten verliehen werden. Erhaltene Punkte zählen nur, wenn der Student einem Haus bereits zugewiesen wurde. Die erhaltenen Punkte werden dann den Häusern zugeschrieben, welchen der Student zugewiesen ist. Dabei können die Punkte eines Hauses nicht kleiner als 0 werden. Auch wenn einem Studenten mehr Punkte abgezogen werden, geht der Punktestand eines Hauses nur auf 0. Zum Beispiel, wenn Hufflepuff in der Summe 5 Punkte hat und Hannah -10 Punkte verliehen werden, dann werden nur -5 Punkte tatsächlich für Hufflepuff verrechnet, der Rest wird ignoriert. Zusätzlich implementieren Sie die Methode winner() von School, welche das Haus mit den meisten Punkten zurückgibt. Falls mehrere Häuser die gleiche Punktzahl haben, dann kann ein beliebiges dieser Häuser zurückgegeben werden. Falls es kein Haus gibt, dann soll die Methode eine IllegalArgumentException werfen. Und implementieren Sie die Methode points() von School, welche die Summe der Punktestände der Häuser zurückgibt.

Code Skelett:

School

Fields:

-

Konstruktor:

-

Methods:

House createHouse(String name)
House winner()
int points()

House

Fields:

-

Konstruktor:

-

Methods:

String name()
int points()
void assign(Student student)

Student

Fields:

-

Konstruktor:

Student(String firstName, String lastName)

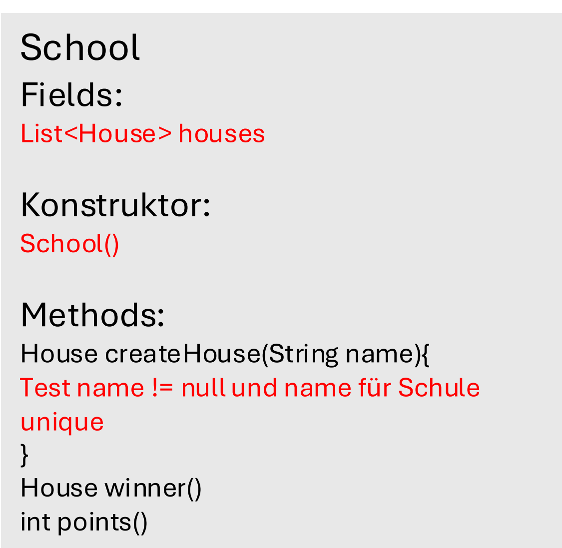
Methods:

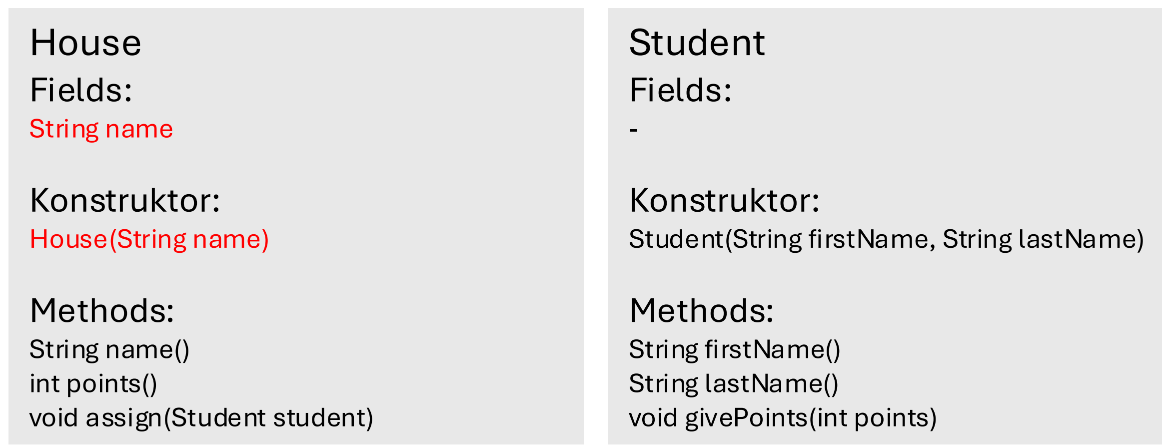
String firstName()
String lastName()
void givePoints(int points)

1. Implementieren Sie den School-Konstruktor und die Methode createHouse(String name), welche als Parameter den gewünschten Namen des Hauses nimmt und ein House Objekt zurückgibt. Der Name eines Hauses darf nicht null sein oder bereits für die Schule vorhanden sein. Die Methode soll in diesen Fällen eine IllegalArgumentException werfen. Alle anderen Namen sind erlaubt. Implementieren Sie zusätzlich die Methode name() der Klasse House, welche den Namen des Hauses als String zurückgibt.

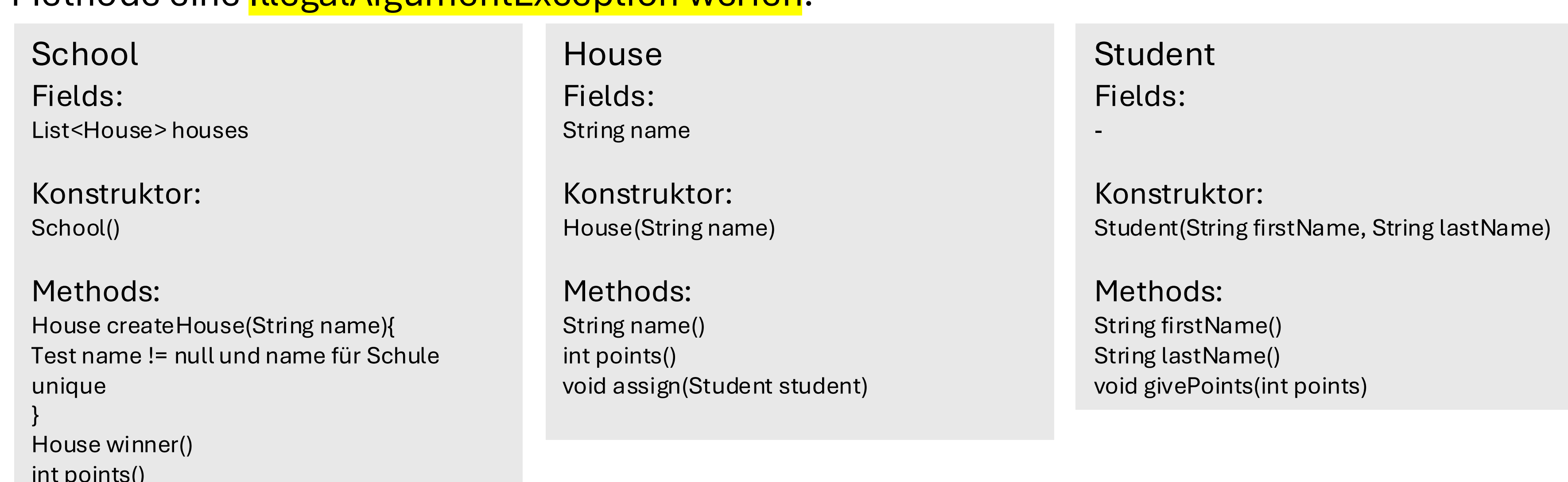
School	House	Student
Fields:	Fields:	Fields:
-	-	-
Konstruktor:	Konstruktor:	Konstruktor:
-	-	Student(String firstName, String lastName)
Methods:	Methods:	Methods:
House createHouse(String name)	String name()	String firstName()
House winner()	int points()	String lastName()
int points()	void assign(Student student)	void givePoints(int points)

1. Implementieren Sie den School-Konstruktor und die Methode createHouse(String name), welche als Parameter den gewünschten Namen des Hauses nimmt und ein House Objekt zurückgibt. Der Name eines Hauses darf nicht null sein oder bereits für die Schule vorhanden sein. Die Methode soll in diesen Fällen eine IllegalArgumentException werfen. Alle anderen Namen sind erlaubt. Implementieren Sie zusätzlich die Methode name() der Klasse House, welche den Namen des Hauses als String zurückgibt.

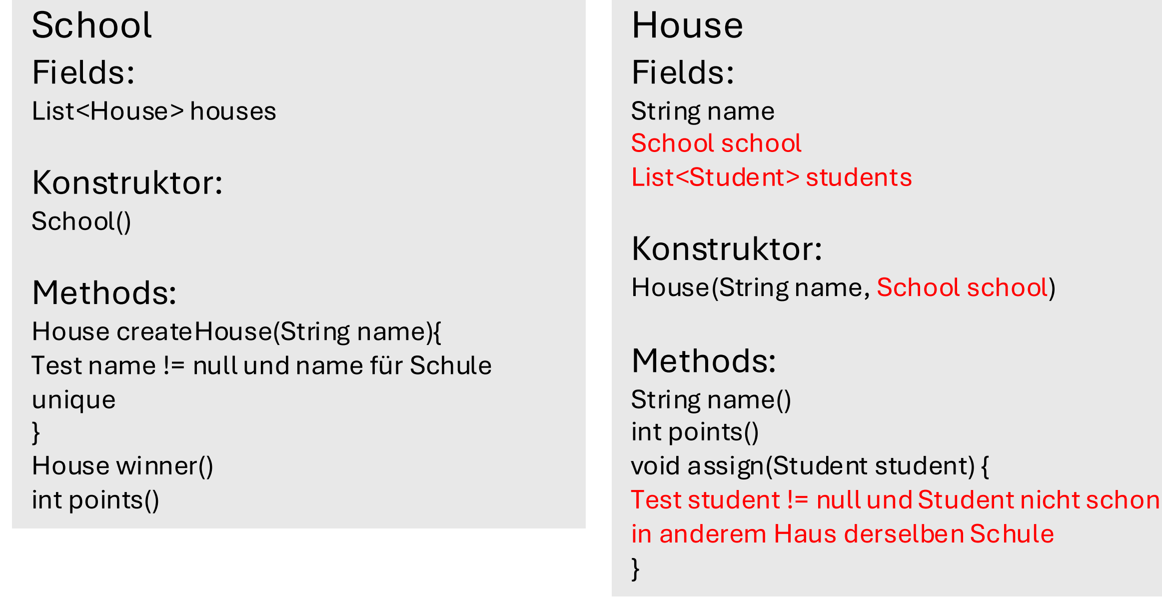




2. Implementieren Sie den Konstruktor von Student, welcher zwei Strings, den Vor- und Nachnamen (in dieser Reihenfolge) nimmt. Sie dürfen annehmen, dass es jeden Namen (Vor- und Nachname zusammen) nur einmal gibt. Vor- und Nachnamen sollen über die Methode firstName() beziehungsweise lastName() erhalten werden können. Implementieren Sie zusätzlich die Methode assign(Student student) der Klasse House, welche einen Studenten als Argument nimmt und ihn in dieses Haus einschreibt. Bei einem null Argument oder falls der Student bereits bei einem Haus der gleichen Schule eingeschrieben ist, dann soll die Methode eine IllegalArgumentException werfen.



2. Implementieren Sie den Konstruktor von Student, welcher zwei Strings, den Vor- und Nachnamen (in dieser Reihenfolge) nimmt. Sie dürfen annehmen, dass es jeden Namen (Vor- und Nachname zusammen) nur einmal gibt. Vor- und Nachnamen sollen über die Methode firstName() beziehungsweise lastName() erhalten werden können. Implementieren Sie zusätzlich die Methode assign(Student student) der Klasse House, welche einen Studenten als Argument nimmt und ihn in dieses Haus einschreibt. Bei einem null Argument oder falls der Student bereits bei einem Haus der gleichen Schule eingeschrieben ist, dann soll die Methode eine IllegalArgumentException werfen.



Student

Fields:

String firstName, lastName

Konstruktor:

Student(String firstName, String lastName)

Methods:

String firstName()
String lastName()
void givePoints(int points)

Als letztes implementieren Sie das Punktesystem. Implementieren Sie dafür vier Methoden: Die Method points() von House gibt die Punkte eines Hauses zurück. Jedes Haus beginnt mit einem Punktestand von 0, wenn es erstellt wird.

School

Fields:

List<House> houses

Konstruktor:

School()

Methods:

House createHouse(String name){
Test name != null und name für Schule unique
}
House winner()
int points()

House

Fields:

String name
School school
List<Student> students

Konstruktor:

House(String name , School school)

Methods:

String name()
int points()
void assign(Student student) {
Test student != null und Student nicht schon in anderem Haus derselben Schule
}

Student

Fields:

String firstName, lastName

Konstruktor:

Student(String firstName, String lastName)

Methods:

String firstName()
String lastName()
void givePoints(int points)

Als letztes implementieren Sie das Punktesystem. Implementieren Sie dafür vier Methoden: Die Method points() von House gibt die Punkte eines Hauses zurück. Jedes Haus beginnt mit einem Punktestand von 0, wenn es erstellt wird.

School

Fields:

List<House> houses

Konstruktor:

School()

Methods:

House createHouse(String name){
Test name != null und name für Schule unique
}
House winner()
int points()

House

Fields:

String name
School school
List<Student> students
int points

Konstruktor:

House(String name , School school)

Methods:

String name()
int points()
void assign(Student student) {
Test student != null und Student nicht schon in anderem Haus derselben Schule
}

Student

Fields:

String firstName, lastName

Konstruktor:

Student(String firstName, String lastName)

Methods:

String firstName()
String lastName()
void givePoints(int points)

Dieser Punktestand kann sich dann durch die Leistungen der Studenten verändern. Die Methode givePoints(int points) von Student nimmt eine positive oder negative Anzahl Punkte, welche dem Studenten verliehen werden. Erhaltene Punkte zählen nur, wenn der Student einem Haus bereits zugewiesen wurde. Die erhaltenen Punkte werden dann den Häusern zugeschrieben, welchen der Student zugewiesen ist. Dabei können die Punkte eines Hauses nicht kleiner als 0 werden.

School

Fields:

List<House> houses

Konstruktor:

School()

Methods:

House createHouse(String name){
Test name != null und name für Schule unique
}
House winner()
int points()

House

Fields:

String name
School school
List<Student> students
int points

Konstruktor:

House(String name , School school)

Methods:

String name()
int points()
void assign(Student student) {
Test student != null und Student nicht schon in anderem Haus derselben Schule
}

Student

Fields:

String firstName, lastName

Konstruktor:

Student(String firstName, String lastName)

Methods:

String firstName()
String lastName()
void givePoints(int points)

Dieser Punktestand kann sich dann durch die Leistungen der Studenten verändern. Die Methode givePoints(int points) von Student nimmt eine positive oder negative Anzahl Punkte, welche dem Studenten verliehen werden. Erhaltene Punkte zählen nur, wenn der Student einem Haus bereits zugewiesen wurde. Die erhaltenen Punkte werden dann den Häusern zugeschrieben, welchen der Student zugewiesen ist. Dabei können die Punkte eines Hauses nicht kleiner als 0 werden.

School

Fields:

List<House> houses

Konstruktor:

School()

Methods:

House createHouse(String name){
Test name != null und name für Schule unique
}
House winner()
int points()

House

Fields:

String name
School school
List<Student> students
int points

Konstruktor:

House(String name , School school)

Methods:

String name()
int points()
void assign(Student student) {
Test student != null und Student nicht schon in anderem Haus derselben Schule
}
void updatePoints(int change){
points = max(0, points + change)
}

Student

Fields:

String firstName, lastName
List<House> houses

Konstruktor:

Student(String firstName, String lastName)

Methods:

String firstName()
String lastName()
void givePoints(int points) {
Für jedes House in houses,
House.updatePoinst(points)
}

Zusätzlich implementieren Sie die Methode winner() von School, welche das Haus mit den meisten Punkten zurückgibt. Falls mehrere Häuser die gleiche Punktzahl haben, dann kann ein beliebiges dieser Häuser zurückgegeben werden. Falls es kein Haus gibt, dann soll die Methode eine IllegalArgumentException werfen. Und implementieren Sie die Methode points() von School, welche die Summe der Punktestände der Häuser zurückgibt.

School

Fields:

List<House> houses

Konstruktor:

School()

Methods:

House createHouse(String name){
Test name != null und name für Schule unique
}
House winner()
int points()

House

Fields:

String name
School school
List<Student> students
int points

Konstruktor:

House(String name , School school)

Methods:

String name()
int points()
void assign(Student student) {
Test student != null und Student nicht schon in anderem Haus derselben Schule
}
void updatePoints(int change){
points = max(0, points + change)
}

Student

Fields:

String firstName, lastName
List<House> houses

Konstruktor:

Student(String firstName, String lastName)

Methods:

String firstName()
String lastName()
void givePoints(int points) {
Für jedes House in houses, House.updatePoinst(points)
}

Zusätzlich implementieren Sie die Methode winner() von School, welche das Haus mit den meisten Punkten zurückgibt. Falls mehrere Häuser die gleiche Punktzahl haben, dann kann ein beliebiges dieser Häuser zurückgegeben werden. Falls es kein Haus gibt, dann soll die Methode eine IllegalArgumentException werfen. Und implementieren Sie die Methode points() von School, welche die Summe der Punktestände der Häuser zurückgibt.

School

Fields:

List<House> houses

Konstruktor:

School()

Methods:

House createHouse(String name){
Test name != null und name für Schule unique
}
House winner() {
Test houses not empty
return ein House mit max points
}
int points() {
return Summe der points aller House aus houses
}

House

Fields:

String name
School school
List<Student> students
int points

Konstruktor:

House(String name , School school)

Methods:

String name()
int points()
void assign(Student student) {
Test student != null und Student nicht schon in anderem Haus derselben Schule
}
void updatePoints(int change){
points = max(0, points + change)
}

Student

Fields:

String firstName, lastName
List<House> houses

Konstruktor:

Student(String firstName, String lastName)

Methods:

String firstName()
String lastName()
void givePoints(int points) {
Für jedes House in houses,
House.updatePoinst(points)
}

Jetzt Klassen implementieren

School

Fields:

List<House> houses

Konstruktor:

School()

Methods:

House createHouse(String name){
Test name != null und name für Schule unique
}
House winner() {
Test houses not empty
return ein House mit max points
}
int points() {
return Summe der points aller House aus houses
}

House

Fields:

String name
School school
List<Student> students
int points

Konstruktor:

House(String name , School school)

Methods:

String name()
int points()
void assign(Student student) {
Test student != null und Student nicht schon in anderem Haus derselben Schule
}
void updatePoints(int change){
points = max(0, points + change)
}

Student

Fields:

String firstName, lastName
List<House> houses

Konstruktor:

Student(String firstName, String lastName)

Methods:

String firstName()
String lastName()
void givePoints(int points) {
Für jedes House in houses,
House.updatePoinst(points)
}

```java
public class Student {
    String firstName, lastName;
    LinkedList<House> houses;
    public Student(String firstName, String lastName) {}

    public String firstName() {return null;}

    public String lastName() {return null;}

    public void givePoints(int points) {}
}
```

Student

Fields:

String firstName, lastName
List<House> houses

Konstruktor:

Student(String firstName, String lastName)

Methods:

String firstName()
String lastName()
void givePoints(int points) {
Für jedes House in houses,
House.updatePoinst(points)
}

```java
public class Student {
    String firstName, lastName;
    LinkedList<House> houses;
    public Student(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
        houses = new LinkedList<House>();
    }

    public String firstName() {return null;}

    public String lastName() {return null;}

    public void givePoints(int points) {}
}
```

Student

Fields:

String firstName, lastName
List<House> houses

Konstruktor:

Student(String firstName, String lastName)

Methods:

String firstName()
String lastName()
void givePoints(int points) {
Für jedes House in houses,
House.updatePoinst(points)
}

```java
public class Student {
    String firstName, lastName;
    LinkedList<House> houses;
    public Student(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
        houses = new LinkedList<House>();
    }
    public String firstName() {
        return firstName;
    }
    public String lastName() {
        return lastName;
    }
    public void givePoints(int points) {}
}
```

Student

Fields:

String firstName, lastName
List<House> houses

Konstruktor:

Student(String firstName, String lastName)

Methods:

String firstName()
String lastName()
void givePoints(int points) {
Für jedes House in houses,
House.updatePoinst(points)
}

```java
public class Student {
    String firstName, lastName;
    LinkedList<House> houses;
    public Student(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
        houses = new LinkedList<House>();
    }
    public String firstName() {
        return firstName;
    }
    public String lastName() {
        return lastName;
    }
    public void givePoints(int points) {
        for(House curHouse: houses) {
            curHouse.updatePoints(points);
        }
    }
}
```

Student

Fields:

String firstName, lastName

List<House> houses

Konstruktor:

Student(String firstName, String lastName)

Methods:

String firstName()

String lastName()

void givePoints(int points) {

Für jedes House in houses,

House.updatePoinst(points)

}

```java
public class House {

    String name;
    School school;
    LinkedList<Student> students;
    int points;

    public String name() {return null;}
    public int points() {return 0;}
    public void assign(Student student) {}
}
```

House

Fields:

String name
School school
List<Student> students
int points

Konstruktor:

House(String name , School school)

Methods:

String name()
int points()
void assign(Student student) {
Test student != null und Student nicht schon in anderem Haus derselben Schule
}
void updatePoints(int change){
points = max(0, points + change)
}

```java
public class House {

    String name;
    School school;
    LinkedList<Student> students;
    int points;

    public House(School school, String name) {
        this.name = name;
        this.school = school;
        this.students = new LinkedList<Student>();
        this.points = 0;
    }

    public String name() {return null;}
    public int points() {return 0;}
    public void assign(Student student) {}
}
```

House

Fields:

String name
School school
List<Student> students
int points

Konstruktor:

House(String name , School school)

Methods:

String name()
int points()
void assign(Student student) {
Test student != null und Student nicht schon in anderem Haus derselben Schule
}
void updatePoints(int change){
points = max(0, points + change)
}

```java
public class House {
    String name;
    School school;
    LinkedList<Student> students;
    int points;

    public House(School school, String name) {
        this.name = name;
        this.school = school;
        this.students = new LinkedList<Student>();
        this.points = 0;
    }

    public String name() {return name;}
    public int points() {return points;}
    public void assign(Student student) {}
}
```

House

Fields:

String name
School school
List<Student> students
int points

Konstruktor:

House(String name , School school)

Methods:

String name()
int points()
void assign(Student student) {
Test student != null und Student nicht schon in anderem Haus derselben Schule
}
void updatePoints(int change){
points = max(0, points + change)
}

```java
public class House {
    String name;
    School school;
    LinkedList<Student> students;
    int points;
    public House(School school, String name) {…}
    public String name() {return name;}
    public int points() {return points;}
    public void assign(Student student){
        if (student == null)
            throw new IllegalArgumentException();
        for (House curHouse: school.houses) {
            for(Student curStu: curHouse.students) {
                if (curStu == student)
                    throw new IllegalArgumentException();
            }
        }
        students.add(student);
        student.houses.add(this);
    }
}
```

House

Fields:

String name
School school
List<Student> students
int points

Konstruktor:

House(String name , School school)

Methods:

String name()
int points()
void assign(Student student) {
Test student != null und Student nicht schon in anderem Haus derselben Schule
}
void updatePoints(int change){
points = max(0, points + change)
}

```java
public class House {
    String name;
    School school;
    LinkedList<Student> students;
    int points;
    public House(School school, String name) {…}
    public String name() {return name;}
    public int points() {return points;}
    public void assign(Student student){…}
    public void updatePoints(int change) {
        this.points = Math.max(0, this.points + change);
    }

}
```

House

Fields:

String name
School school
List<Student> students
int points

Konstruktor:

House(String name , School school)

Methods:

String name()
int points()
void assign(Student student) {
Test student != null und Student nicht schon in anderem Haus derselben Schule
}
void updatePoints(int change){
points = max(0, points + change)
}

```java
public class School {
    LinkedList<House> houses;
    public School () {
        this.houses = new LinkedList<House>();
    }

    public House createHouse(String name) {return null;}
    public House winner() {return null;}
    public int points() {{return 0;}
}
```

School

Fields:

List<House> houses

Konstruktor:

School()

Methods:

House createHouse(String name){
Test name != null und name für Schule unique
}
House winner() {
Test houses not empty
return ein House mit max points
}
int points() {
return Summe der points aller House aus houses
}

```java
public class School {
    LinkedList<House> houses;
    public School () {
        this.houses = new LinkedList<House>();
    }
    public House createHouse(String name) {
        if(name == null)
            throw new IllegalArgumentException();
        for(House h : houses) {
            if(h.name().equals(name)) {
                throw new IllegalArgumentException();
            }
        }
        House newHouse = new House(this, name);
        houses.add(newHouse);
        return newHouse;
    }
    public House winner() {return null;}
    public int points() {{return 0;}
}
```

School

Fields:

List<House> houses

Konstruktor:

School()

Methods:

House createHouse(String name){
Test name != null und name für Schule unique
}

House winner() {
Test houses not empty
return ein House mit max points
}

int points() {
return Summe der points aller House aus houses
}

```java
public class School {
    LinkedList<House> houses;
    public School () {
        this.houses = new LinkedList<House>();
    }
    public House createHouse(String name) {…}
    public House winner() {
        if (houses.size() == 0)
            throw new IllegalArgumentException();
        House maxHouse = houses.get(0);
        for(House house: houses) {
            if(house.points() > maxHouse.points())
                maxHouse = house;
        }
        return maxHouse;
    }
    public int points() {{return 0;}
}
```

School

Fields:

List<House> houses

Konstruktor:

School()

Methods:

House createHouse(String name){
Test name != null und name für Schule unique
}

House winner() {
Test houses not empty
return ein House mit max points
}

int points() {
return Summe der points aller House aus houses
}

```java
public class School {
    LinkedList<House> houses;
    public School () {
        this.houses = new LinkedList<House>();
    }
    public House createHouse(String name) {…}
    public House winner() {…}
    public int points() {
        int sum = 0;
        for(House house: houses) {
            sum += house.points;
        }
        return sum;
    }
}
```

School

Fields:

List<House> houses

Konstruktor:

School()

Methods:

House createHouse(String name){
Test name != null und name für Schule unique
}
House winner() {
Test houses not empty
return ein House mit max points
}
int points() {
return Summe der points aller House aus houses
}

Scanner

Scanner: Methoden

Scanner

- **`next()`:** Wenn ein nächster `String` existiert im `Scanner`, dann wird dieser eingelesen und zurückgegeben. Sonst gibt es eine `NoSuchElementException`.

- **`nextInt()`:** Wenn ein nächster `Int` existiert im `Scanner`, dann wird dieser eingelesen und zurückgegeben. Sonst gibt es eine `NoSuchElementException`.

- **`nextBoolean()`:** Wenn ein nächster `Boolean` existiert im `Scanner`, dann wird dieser eingelesen und zurückgegeben. Sonst gibt es eine `NoSuchElementException`.

- **`nextDouble()`:** Wenn ein nächster `Double` existiert im `Scanner`, dann wird dieser eingelesen und zurückgegeben. Sonst gibt es eine `NoSuchElementException`.

Scanner: Methoden

Scanner

- **`hasNext():`** Prüft ob es im `Scanner` einen nächsten `String` gibt.
- **`hasNextInt():`** Prüft ob es im `Scanner` einen nächsten `Int` gibt.
- **`hasNextBoolean():`** Prüft ob es im `Scanner` einen nächsten `Boolean` gibt.
- **`hasNextDouble():`** Prüft ob es im `Scanner` einen nächsten `Double` gibt.

Scanner: LOCALE

Scanner

- Beim Einlesen von `double` kommt es zu Problemen…

```
jshell> scanner.nextDouble()
|  Exception java.util.InputMismatchException
|        at Scanner.throwFor (Scanner.java:947)
|        at Scanner.next (Scanner.java:1602)
|        at Scanner.nextDouble (Scanner.java:2573)
|        at (#3:1)
```

- Im Deutschen werden Kommazahlen mit , statt mit . benutzt. Deshalb kann ein Scanner nicht `4.5` einlesen.

- **Lösung:** `scanner.useLocale(Locale.US)`.

Vorbesprechung

Aufgabe 1: Loop-Invarianten

1. Um die Loop-Invariante einfacher schreiben zu können, dürfen Sie `min(arr, i)` benutzen. Hier steht `min(arr, i)` für das minimale Element in dem Array `arr` von Index `0` bis und mit Index `i`. Alternativ könnte man auch formale Notation benutzen, in dem man mit Quantoren arbeitet. Zum Beispiel, falls $\texttt{m} = \texttt{min(arr, i)}$, dann könnten Sie äquivalent folgendes schreiben

$$\forall\, 0 \leq \texttt{j} \leq \texttt{i}\, (\texttt{arr[j]} \leq \texttt{m})$$

```
int min(int[] arr) {
  // Precondition: arr != null 0 < arr.length
  int m = arr[0];
  int i = 1;

  // Loop-Invariante:
  while (i < arr.length) {
    if (arr[i] < m) {
      m = arr[i];
    }

    i++;
  }

  // Postcondition: m = min(arr, arr.length)
  return m;
}
```

Aufgabe 1: Loop-Invarianten

2.
```
String append(String str1, String str2) {
  // Precondition: str1 != null && str2 != null
  String s1 = str1;
  String s2 = str2;

  // Loop-Invariante:
  while (!s2.equals("")) {
    s1 = s1 + s2.charAt(0);
    s2 = s2.substring(1);
  }

  // Postcondition: s.equals(str1 + str2)
  return s1;
}
```

Achtung: Die Bedingung `str1 != null && str2 != null` ist wichtig, damit Aufrufe wie `s2.equals()`, `s2.charAt(0)` und `s2.substring(1)` überhaupt möglich sind. Der Aufruf `s2.substring(1)` produziert das gleiche Resultat wie `s2.substring(1, s2.length())`.

Aufgabe 2: Database

In dieser Aufgabe implementieren Sie für eine Datenbank von Personengesundheitsdaten das Deklassifizieren von Einträgen (Task a) und das Verlinken von Einträgen (Task b). Alle Unteraufgaben können separat gelöst werden.
Die Datenbank selber ist bereits mit der Klasse `Database` implementiert. Die Datenbank hält eine Liste von Einträgen, welche durch die Klasse `Item` repräsentiert werden. Die folgenden 4 Paragraphen erklären alle in der Vorlage gegebenen Klassen im Detail.

`Item` Die Klasse `Item` repräsentiert einen Datenbankeintrag mit 4 Attributen: eine ID (int), ein Alter (int), einen Gesundheitswert (int), und ein Sicherheitslevel, welches durch die Klasse `Level` repräsentiert wird. Alter und Gesundheitswert sind immer ≥ 0. Die Methoden `Item.getID()`, `Item.getAge()`, `Item.getHealth()`, `Item.getLevel()` geben jeweils die ID, das Alter, den Gesundheitswert, und das Sicherheitslevel eines Eintrags zurück. Die Methode `Item.setHealth(int newHealth)` setzt den Gesundheitswert auf `newHealth`. Die anderen Attribute können nicht geändert werden.

`Level` Die Klasse `Level` repräsentiert ein Sicherheitslevel. Ein Sicherheitslevel wird über eine Liste von Integern definiert, welches in einem Attribut der Klasse `Level` gespeichert wird und von der Methode `Level.getPoints()` zurückgegeben wird. Ein Level `A` ist *verwandt* mit einem Level `B`, falls die Summe der Werte in `A.getPoints()` gleich der Summe der Werte in `B.getPoints()` ist. Zum Beispiel ist das Level $[1, 2, 3, 4]$ verwandt mit den Levels $[10]$ und $[4, 6]$ (die Summe ist überall 10), aber nicht mit dem Level $[4, 5]$.

Aufgabe 2: Database

`ItemFactory` Die Klasse `ItemFactory` wird verwendet, um Datenbankeinträge zu erstellen. Die Methode `ItemFactory.createItem(Level level, int id, int age, int health)` gibt ein Exemplar der Klasse `Item` zurück, deren Attribute mit den Argumenten initialisiert wurden.

`Database` Die Klasse `Database` repräsentiert eine Datenbank und hat folgende vorgegebene Methoden:

- `Database.getItemFactory()` gibt ein Exemplar von `ItemFactory` zurück. Die `ItemFactory` `I` ist assoziiert mit der Datenbank `D`, falls `I` von `D.getItemFactory()` zurückgegeben wird.
- `Database.add(Item item)` fügt der Datenbank den Eintrag `item` hinzu.
- `Database.getItems()` gibt die Liste aller Einträge zurück, welcher der Datenbank hinzugefügt wurden. Sie dürfen annehmen, dass für eine Datenbank `D` alle Einträge in `D.getItems()` eine einzigartige ID haben, über `D.add` hinzugefügt wurden, über `D.getItemFactory()` erstellt wurden, und keiner anderen Datenbank hinzugefügt werden. Ein hinzugefügter Eintrag wird nie wieder entfernt.

1. Implementieren Sie die Methode `ItemFactory.createDeclass(Level level, int id, int targetId)`, die einen *Deklassifikationseintrag* zurückgibt. Ein Deklassifikationseintrag ist selber ein Eintrag, also ein Exemplar der Klasse `Item`. Ein Deklassifikationseintrag hat damit auch eine ID, ein Sicherheitslevel, ein Alter, und einen Gesundheitswert, welche von den

Aufgabe 2: Database

entsprechenden getter-Methoden zurückgegeben werden. ID und Sicherheitslevel eines Deklassifikationseintrags sind jeweils das `id` und `level` Argument des `createDeclass` Aufrufs, mit welchem der Eintrag erstellt wurde. Das Alter und der Gesundheitswert eines Deklassifikationseintrags sind jeweils das Alter und der Gesundheitswert des *Zieleintrags* vom Deklassifikationseintrag. Der Zieleintrag von einem Deklassifikationseintrag `D` ist der Eintrag `E`, so dass

- `E.getID()` gleich dem Parameter `targetId` ist, mit welchem `D` erstellt wurde; *und*
- `E` aus der Datenbank ist, mit welcher die `ItemFactory` assoziiert ist, mit welcher `D` erstellt wurde.

Falls es keinen Zieleintrag gibt, wird eine `IllegalArgumentException` von der Methode `createDeclass` geworfen. Beachten Sie, dass Zieleinträge selber Deklassifikationseinträge sein können. Ein Aufruf der Methode `Item.setHealth(h)` auf einem Deklassifikationseintrag hat keinen Effekt; dies wird nicht in den Tests überprüft.

Ein Deklassifikationseintrag `R` *erreicht* einen Eintrag `A`, falls entweder `A` der Zieleintrag von `R` ist oder falls der Zieleintrag von `R` ein Deklassifikationseintrag ist, welcher `A` erreicht. Die Methode `createDeclass` wirft eine `IllegalArgumentException`, falls der zurückzugebene Deklassifikationseintrag `R` einen Eintrag erreicht, dessen Level verwandt ist mit dem Level von `R`. Zur Erinnerung: Der Paragraph über die Klasse `Level` erklärt, wann zwei Level verwandt sind.

Aufgabe 2: Database

entsprechenden getter-Methoden zurückgegeben werden. ID und Sicherheitslevel eines Deklassifikationseintrags sind jeweils das `id` und `level` Argument des `createDeclass` Aufrufs, mit welchem der Eintrag erstellt wurde. Das Alter und der Gesundheitswert eines Deklassifikationseintrags sind jeweils das Alter und der Gesundheitswert des *Zieleintrags* vom Deklassifikationseintrag. Der Zieleintrag von einem Deklassifikationseintrag `D` ist der Eintrag `E`, so dass

- `E.getID()` gleich dem Parameter `targetId` ist, mit welchem `D` erstellt wurde; *und*
- `E` aus der Datenbank ist, mit welcher die `ItemFactory` assoziiert ist, mit welcher `D` erstellt wurde.

Falls es keinen Zieleintrag gibt, wird eine `IllegalArgumentException` von der Methode `createDeclass` geworfen. Beachten Sie, dass Zieleinträge selber Deklassifikationseinträge sein können. Ein Aufruf der Methode `Item.setHealth(h)` auf einem Deklassifikationseintrag hat keinen Effekt; dies wird nicht in den Tests überprüft.

Ein Deklassifikationseintrag `R` *erreicht* einen Eintrag `A`, falls entweder `A` der Zieleintrag von `R` ist oder falls der Zieleintrag von `R` ein Deklassifikationseintrag ist, welcher `A` erreicht. Die Methode `createDeclass` wirft eine `IllegalArgumentException`, falls der zurückzugebene Deklassifikationseintrag `R` einen Eintrag erreicht, dessen Level verwandt ist mit dem Level von `R`. Zur Erinnerung: Der Paragraph über die Klasse `Level` erklärt, wann zwei Level verwandt sind.

Aufgabe 2: Database

2. Implementieren Sie die Methode `Database.createLink(List<Integer> ids)`. Der Methodenaufruf `D.createLink(ids)` *verlinkt* alle Einträge der Datenbank `D` miteinander, welche eine ID haben, die im Argument `ids` enthalten ist. Wenn `E.setHealth(h)` auf einem Eintrag `E` aufgerufen wird, dann wird der Gesundheitswert aller Einträge, welche mit `E` verlinkt sind, auf das Argument `h` gesetzt. Einträge können beliebig oft verlinkt werden und verlinken ist transitiv, das heisst, wenn ein Eintrag `A` mit einem Eintrag `B` verlinkt ist und `B` mit einem Eintrag `C` verlinkt ist, dann ist `A` auch mit `C` verlinkt. Verlinken ist auch immer symmetrisch, das heisst, wenn `A` mit `B` verlinkt ist, dann ist auch `B` mit `A` verlinkt. Zusätzlich ist verlinken reflexiv, das heisst, ein Eintrag ist immer mit sich selber verlinkt.

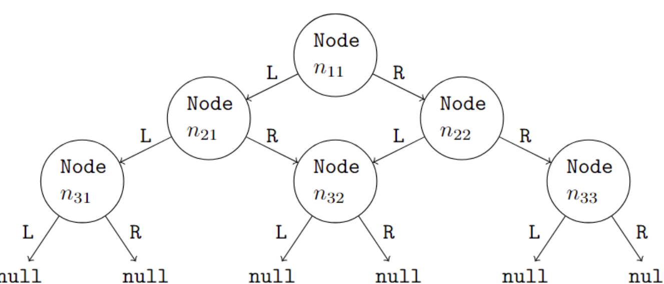
   Der Aufruf `D.createLink(ids)` soll eine `IllegalArgumentException` werfen, falls es eine ID im Argument `ids` gibt, für welche es keinen Eintrag mit der gleichen ID in der Datenbank `D` gibt.

Wir geben zwei Testdateien zur Verfügung. "DatabaseTest.java" enthält Tests, welche wir an einer Prüfung geben würden. "GradingDatabaseTest.java" enthält Tests, welche wir zum Korrigieren einer Prüfung verwenden würden. Testen Sie Ihre Lösung zuerst ausgiebig mit "DatabaseTest.java" (am besten fügen Sie selber neue Tests hinzu) und dann können Sie "GradingDatabaseTest.java" verwenden, um zu sehen wie Ihre Lösung an einer Prüfung abgeschnitten hätte.

Aufgabe 3: Pyramide

Die Klasse `Node` repräsentiert einen Knoten in einem gerichteten Graphen, wobei es für jeden Knoten n_1 höchstens zwei gerichtete Kanten von n_1 zu anderen Knoten n_2, n_3 geben kann (n_2 und n_3 können gleich sein). Wir unterscheiden dabei zwischen dem linken und dem rechten Knoten. Die Methode `Node.getLeft()` gibt den linken Knoten und `Node.getRight()` den rechten Knoten zurück (als `Node`-Objekt). Wenn der linke Knoten von n_1 nicht existiert, dann gibt `Node.getLeft()` `null` zurück (analog für den rechten Knoten).

Das Ziel dieser Aufgabe ist, für ein `Node`-Objekt zu entscheiden, ob der durch das `Node`-Objekt definierte Graph einer Pyramide entspricht. Zum Beispiel entspricht der folgende Graph einer Pyramide.



Aufgabe 3: Pyramide

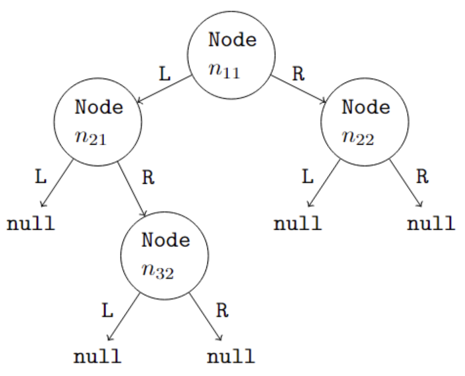
Beachten Sie, dass der rechte Knoten von n_{21} gleich ist wie der linke Knoten von n_{22} (das heisst die Node-Objekte sind gleich!). Ein Graph (wie oben repräsentiert) definiert eine Pyramide genau dann, wenn folgende Bedingungen gelten:

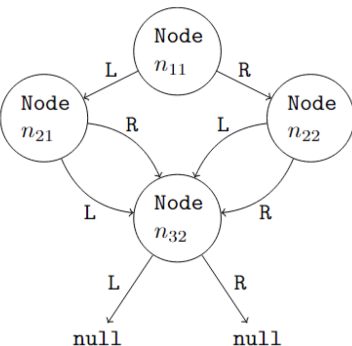
- Der Graph kann in $k \geq 1$ Stufen (Stufe 1, Stufe 2,..., Stufe k) aufgeteilt werden, wobei Stufe i aus i unterschiedlichen Knoten n_{i1}, n_{i2}, ..., n_{ii} besteht. Falls der Graph k Stufen hat, dann hat dieser genau $\frac{k(k+1)}{2}$ unterschiedliche Knoten (Knoten aus verschiedenen Stufen sind unterschiedlich).
- Für Stufe i ($1 \leq i < k$) gilt: der linke Knoten von n_{ij} ($1 \leq j \leq i$) ist durch $n_{(i+1)j}$ gegeben und der rechte Knoten von n_{ij} ist durch $n_{(i+1)(j+1)}$ gegeben.
- Für Stufe k gilt: es gibt keinen linken und keinen rechten Knoten für n_{kj} ($1 \leq j \leq k$).

Die folgenden Graphen entsprechen zum Beispiel keinen Pyramiden:
Implementieren Sie die `boolean isPyramid(Node node)`-Methode, welche, für den Graph G durch `node` definiert, entscheidet, ob G eine Pyramide definiert. Sie dürfen annehmen, dass G keine Zyklen hat. Die Methode soll eine `IllegalArgumentException` werfen, wenn das Argument `null` ist.

Tipp: Prüfen Sie die Bedingungen Stufe für Stufe, beginnend bei Stufe 1.

Aufgabe 3: Pyramide





Aufgabe 4: Rechnungen (erweitert)

In dieser Aufgabe erweitern Sie eine vorherige Aufgabe, in welcher ein System für Stromverbräuche Rechnungen erstellt. Konkret gibt es drei Erweiterungen: (1) Es sollen auch nicht korrekt formatierte Eingabedateien gehandhabt werden. (2) Ein Kunde kann eine beliebige Anzahl von Verbrauchswerten haben. (3) Es gibt eine neue Unteraufgabe b. In der folgenden Aufgabenbeschreibung für Unteraufgabe a sind die Änderung in **bold** markiert.

a) Vervollständigen Sie die `process`-Methode in der Klasse `Bills`. Die Methode hat zwei Argumente: einen `Scanner`, von dem Sie den Inhalt der Eingabedatei lesen sollen, und einen `PrintStream`, in welchen Sie die unten beschriebenen Informationen schreiben.

   Ihr Programm muss **auch mit manchen nicht korrekt formatierten Eingabedatein umgehen**. **Die Aufgabestellung gibt an, wie mit nicht korrekt formatierten Eingaben umzugehen ist**. Ein Beispiel einer korrekt formatierten Datei finden Sie im Projekt unter dem Namen "Data.txt". Exceptions im Zusammenhang mit Ein- und Ausgabe können Sie ignorieren.

   Eine valide Eingabedatei enthält Zeilen, die entweder den Tarif, der angewendet werden soll, oder die Daten für den Stromverbrauch eines Kunden beschreiben. Der Verbrauch eines Kunden ist niemals grösser als 100000 Kilowattstunden.

   Eine Tarifbeschreibung hat folgendes Format:

$$\texttt{Tarif}\textvisiblespace n \textvisiblespace l_1 \textvisiblespace p_1 \ldots l_n \textvisiblespace p_n$$

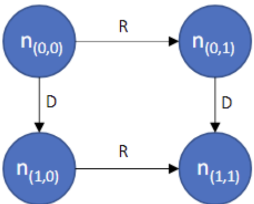
Nachbesprechung

Aufgabe 1: Square Grid

In dieser Aufgabe betrachten wir gerichtete Graphen, wobei es für jeden Knoten g höchstens zwei gerichtete Kanten von g zu anderen Knoten f, h geben kann (f, g, h können gleich sein). Wir unterscheiden dabei zwischen der rechten und der unteren Kante (und damit dem rechten und dem unteren Knoten).

Die Klasse `Node` repräsentiert einen Knoten in einem solchen Graphen. Die Methode `Node.getRight()` (bzw. `Node.getDown()`) gibt den rechten Knoten (bzw. unteren Knoten) zurück (als `Node`-Objekt). Wenn der rechte Knoten von n_0 nicht existiert, dann gibt `Node.getRight()` `null` zurück (analog für den unteren Knoten). Die Methode `Node.setRight(Node r)` (bzw. `Node.setDown(Node d)`) setzt den rechten (bzw. unteren) Knoten.

Das Ziel der Aufgabe ist, einen von einem `Node`-Objekt definierten Graphen zu analysieren. Konkret geht es darum, die Grösse des grössten quadratischen Gitters in dem Graphen zu bestimmen, der mit dem übergebenen `Node`-Objekt beschrieben wird, welches den gleichen Ursprungsknoten wie der Graph hat.



Aufgabe 1: Square Grid

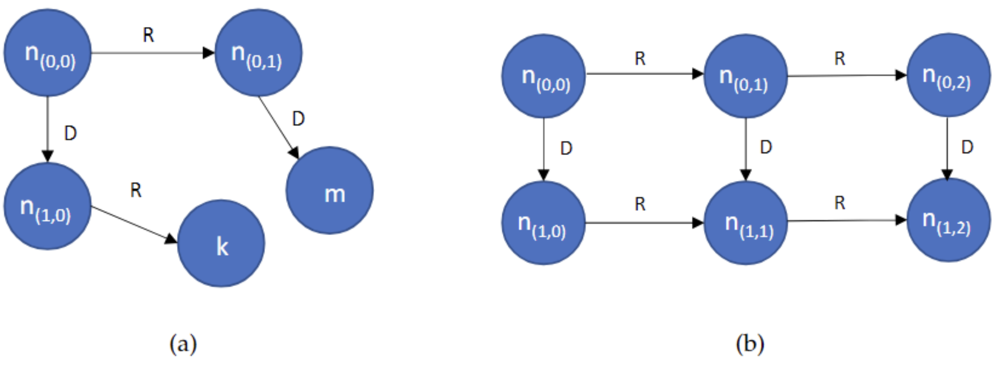


Abbildung 2: Graphen mit quadratischen Gittern als Teilgraphen

Referenzen vs Objekte

Aufgabe 2: Umkehrung

In einem vorherigen Übungsblatt haben Sie eine Linked List für Integers implementiert. In dieser Aufgabe fügen Sie dieser `LinkedIntList` eine weitere Methode hinzu, welche die Liste umkehrt. Eine Liste gilt als umgekehrt, wenn für jedes Paar von Nodes `a` und `b`, für welche zuvor `a == b.next` gegolten hat, in der neuen (umgekehrten) Liste `b == a.next` gilt. Zusätzlich entspricht nach der Umkehrung der erste Node der neuen Liste dem letzten Node der ursprünglichen Liste (und umgekehrt).

Vervollständigen Sie die Methode `reverse()` in der Klasse `LinkedIntList`. Die Methode soll, wie oben definiert, die Liste umkehren. Achten Sie darauf, dass Sie wirklich die Reihenfolge der Nodes selbst umkehren. Es reicht nicht aus, die Reihenfolge der enthaltenen `int`-Werte umzukehren. Es müssen auch in der umgekehrten Liste dieselben Instanzen von `IntNodes` wie in der ursprünglichen Liste verwendet werden. Erstellen Sie also *keine* neuen `IntNodes` mit `new IntNode()`. In der Datei "UmkehrungTest.java" finden Sie einen einfachen Test.

Aufgabe 3: “KI” für das Ratespiel

In Übung 5 implementierten Sie ein Spiel, in welchem der Computer ein Wort auswählt und der Spieler dieses erraten muss. Dort war der Spieler der Benutzer des Programms. In dieser Aufgabe sollen Sie verschiedene “künstliche” Spieler entwickeln. Das heisst, anstelle des Menschen, der über die Konsole Tipps eingibt, werden die Tipps von (mehr oder weniger “intelligenten”) Programmen abgegeben. Ihr Ziel ist es, einen künstlichen Spieler zu entwickeln, der über mehrere Spiele hinweg die Wörter in so wenig Versuchen wie möglich errät.

Die Übungsvorlage enthält bereits den Code für das Ratespiel. Gegenüber Übung 5 ist dieser nun in verschiedene Klassen aufgeteilt. Die drei Hauptklassen sind `RateSpiel`, `Computer` und `Spieler`. Die Klasse `RateSpielApp` enthält eine `main`-Methode, welche das Spiel aufsetzt und durchführt. Durch die Aufteilung ist es möglich, mittels Vererbung Spieler mit unterschiedlichem Verhalten zu schreiben. Die Klasse `Spieler` enthält nämlich nur die Deklarationen der benötigten Methoden, aber keine (sinnvolle) Funktionalität. Subklassen von `Spieler` überschreiben diese Methoden und definieren damit das Verhalten eines Spielers.

Ein konkreter Spieler ist ebenfalls schon in der Vorlage vorhanden: der `KonsolenSpieler`. Dieser besitzt allerdings keine eigene “Intelligenz”, sondern holt sich die Tipps über die Konsole vom Benutzer. Ein `RateSpiel` mit einem `KonsolenSpieler` verhält sich also so wie das Spiel in Übung 5. Starten Sie die `RateSpielApp` und überzeugen Sie sich selbst[1].

Aufgabe 4: Klassenrätsel

In dieser Aufgabe sollen Sie zeigen, dass Sie mit Klassen und Vererbung umgehen können. Im Anhang A finden Sie ein Programm, welches Instanzen von Klassen erstellt und Methoden aufruft. Das Programm macht nichts Sinnvolles und dient nur dem Testen Ihrer Fähigkeiten. In Anhang B befinden sich die verwendeten Klassen, jedoch sind die Klassen noch nicht vollständig. Bei manchen der Klassen fehlt noch die `extends`-Klausel, welche angibt, dass eine Klasse von einer anderen Klasse erbt. Ihre Aufgabe ist es, die nötigen `extends`-Klauseln hinzuzufügen, so dass alles kompiliert und so dass die Ausgabe des Programms von Anhang A am Ende so aussieht wie im Anhang C gezeigt.

Der Code von Anhang A and Anhang B befindet sich in Ihrem `src`-Ordner. Zusätzlich enthält "KlassenTest.java" einen Unit-Test, welcher prüft, ob die Ausgabe des Programms dem Output aus Anhang C entspricht. Beachten Sie, dass Sie für diese Aufgabe **ausschliesslich** `extends`-Klauseln hinzufügen (diese kann es nur an den grauen Boxen aus Anhang B geben), kein anderer Code darf verändert werden.

Tipp: Lösen Sie die Aufgabe zuerst auf Papier, ohne die Hilfe von Eclipse. Sobald Sie herausgefunden haben, welche Klassen von welchen Klassen erben, testen Sie Ihre Lösung in Eclipse. Dies hilft Ihnen, Ihr Wissen über Vererbung zu testen. In der Vergangenheit wurden ähnliche Aufgaben im schriftlichen Teil der Prüfung gestellt.

Kahoot zu Inheritance

https://create.kahoot.it/details/a0c25a8a-38bc-4018-a3eb-03cb1cf8bd6c