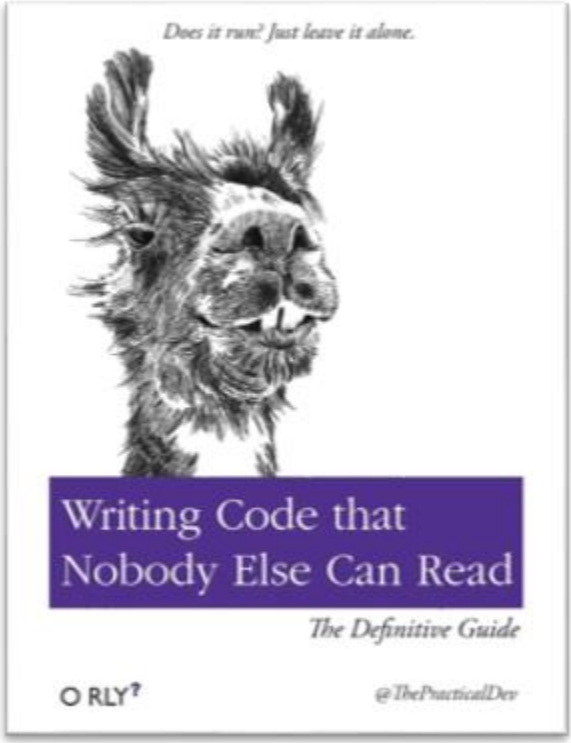
252-0027

Einführung in die Programmierung

Übungen

Woche 13: Inheritance, Collections



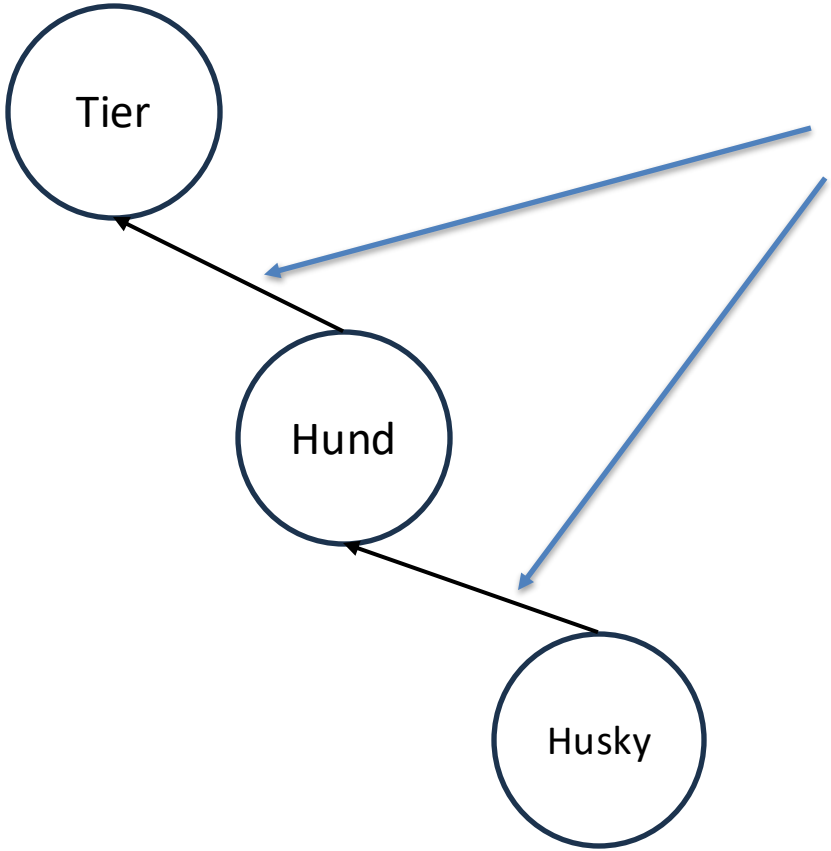
Timo Baumberger
Departement Informatik
ETH Zürich

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

Bonus u12 (IntelliJ)

Konstruktoren



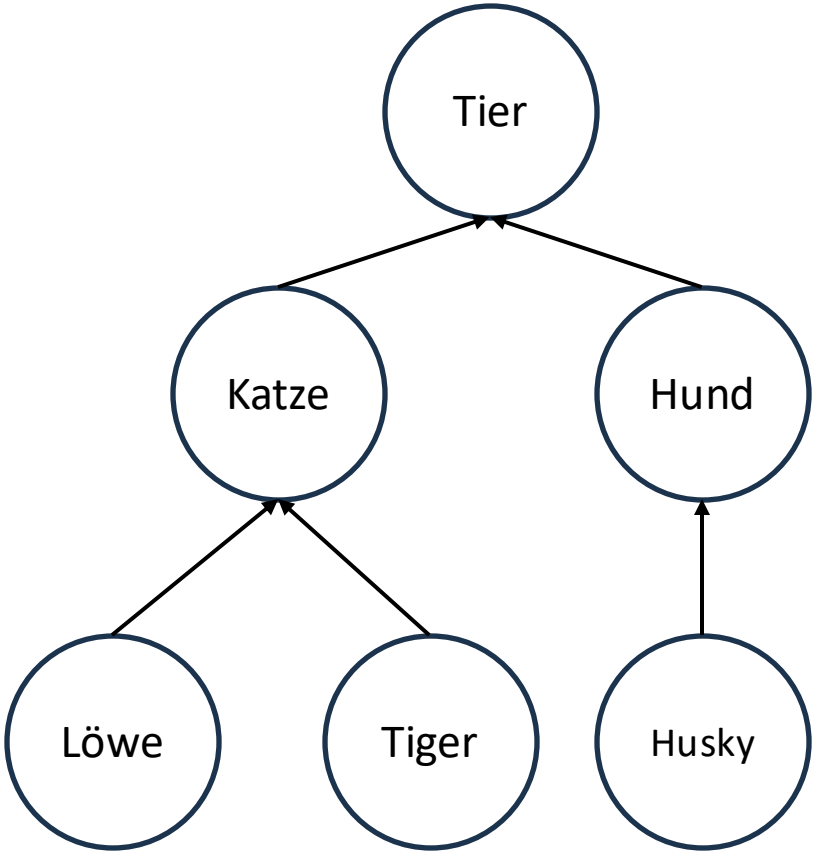
Die Pfeile sind eine “erbt von” - Beziehung

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

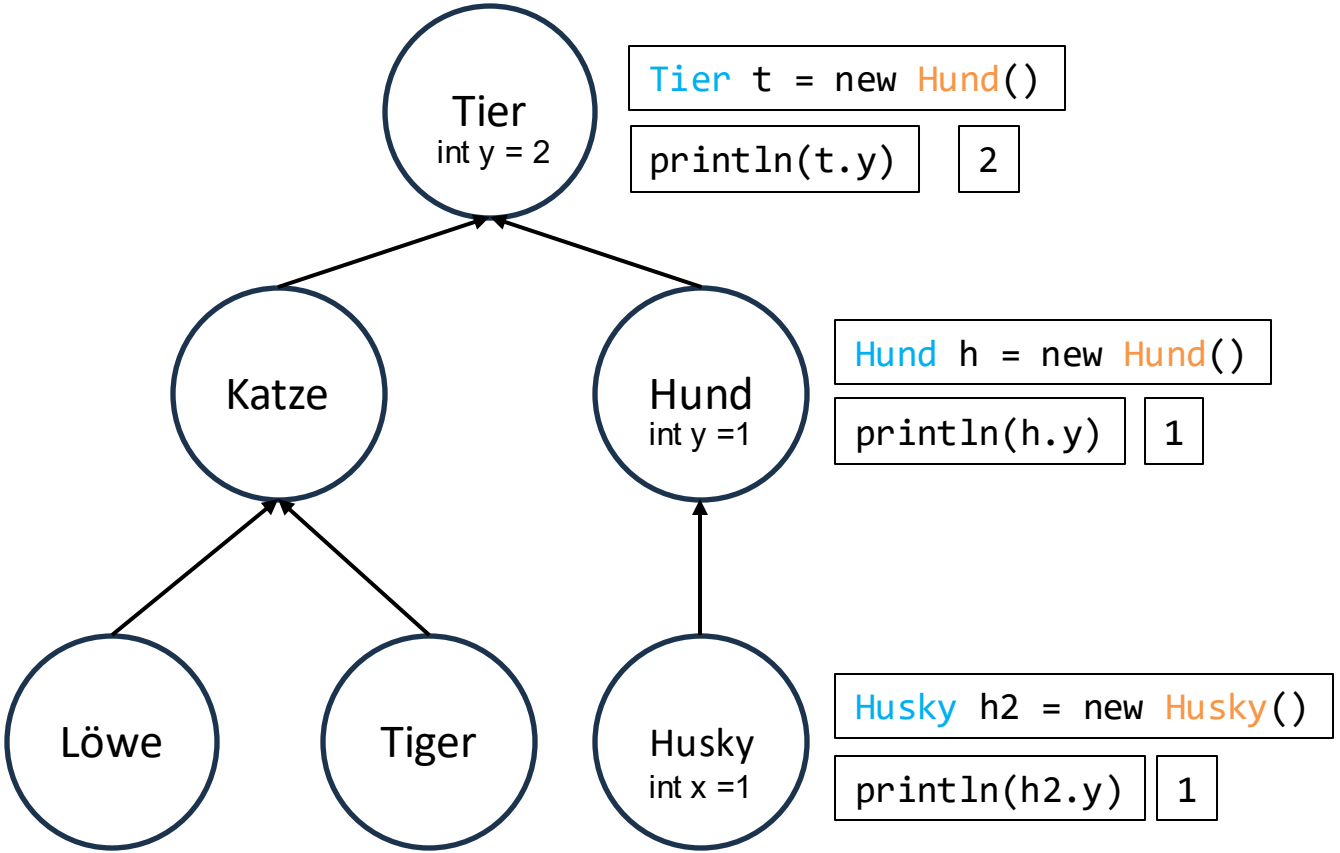
- Ein Husky ist ein Tier und ein Hund.

- Wir wollen sichergehen, dass Attribute richtig vererbt werden

Konstruktoren - Beispiel



Konstruktoren - Beispiel

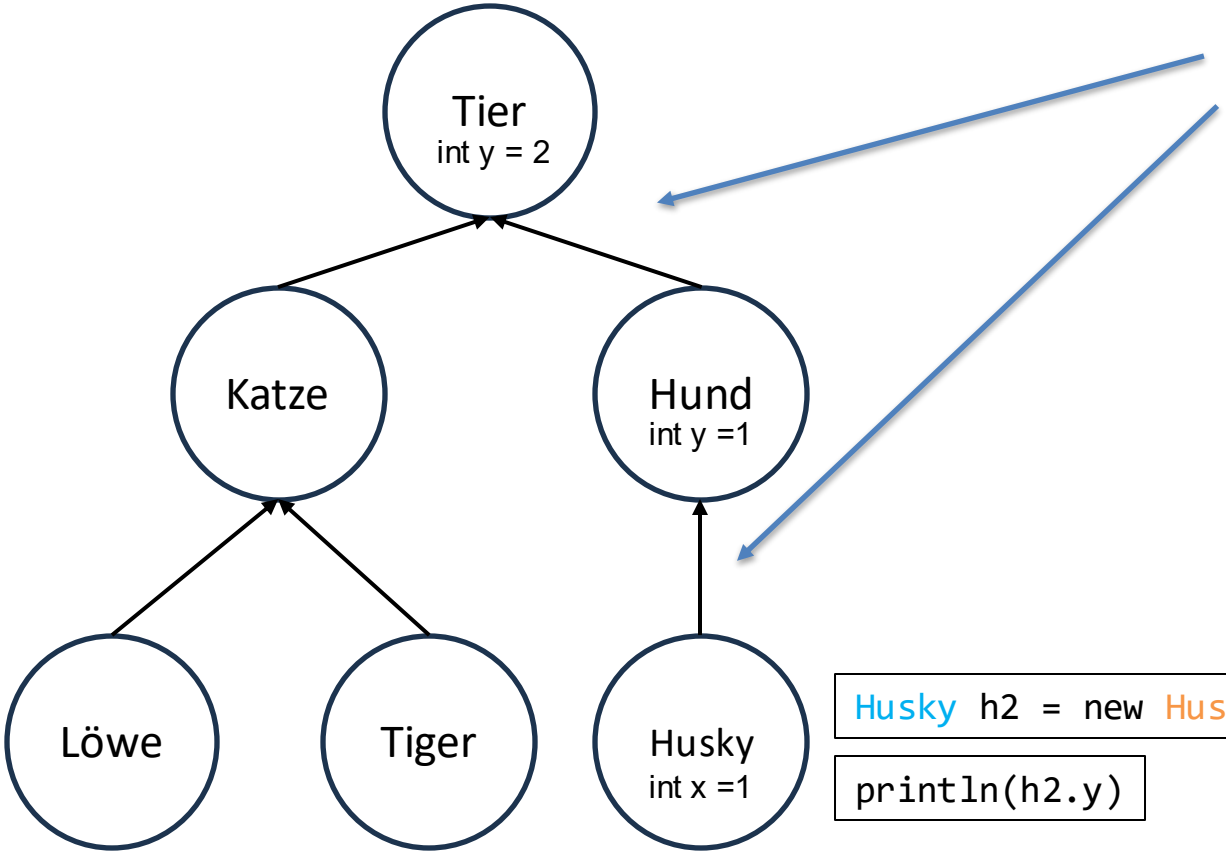



```
public Tier(){
    this.y = 2;
}
```

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

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

Konstruktoren - Beispiel



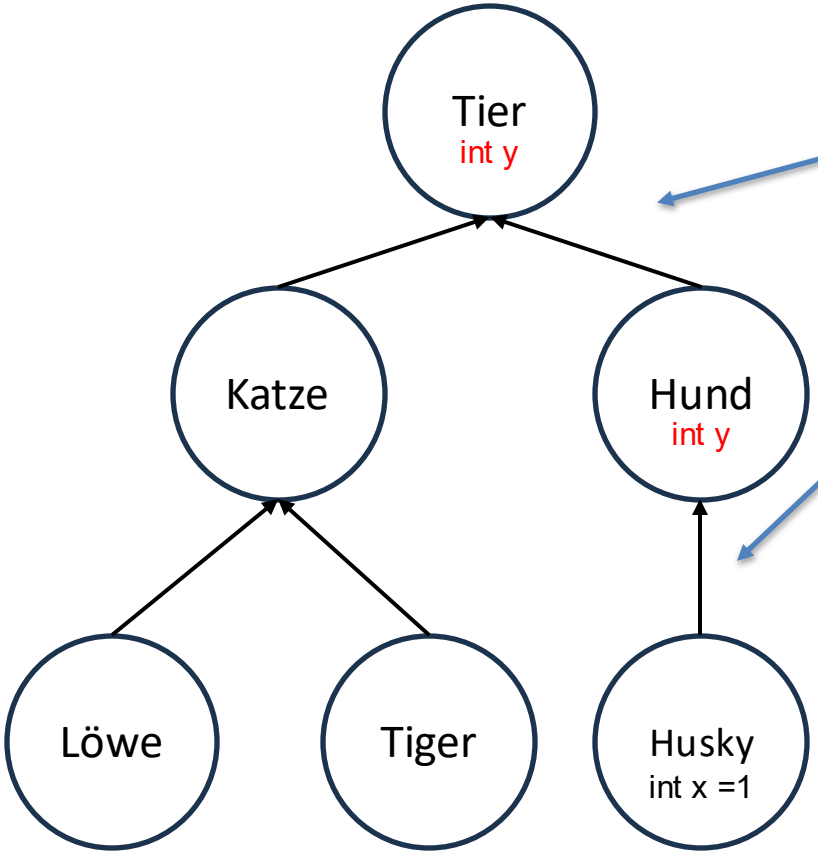
Die Pfeile sind eine “erbt von” - Beziehung

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

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

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

Konstruktoren - Beispiel



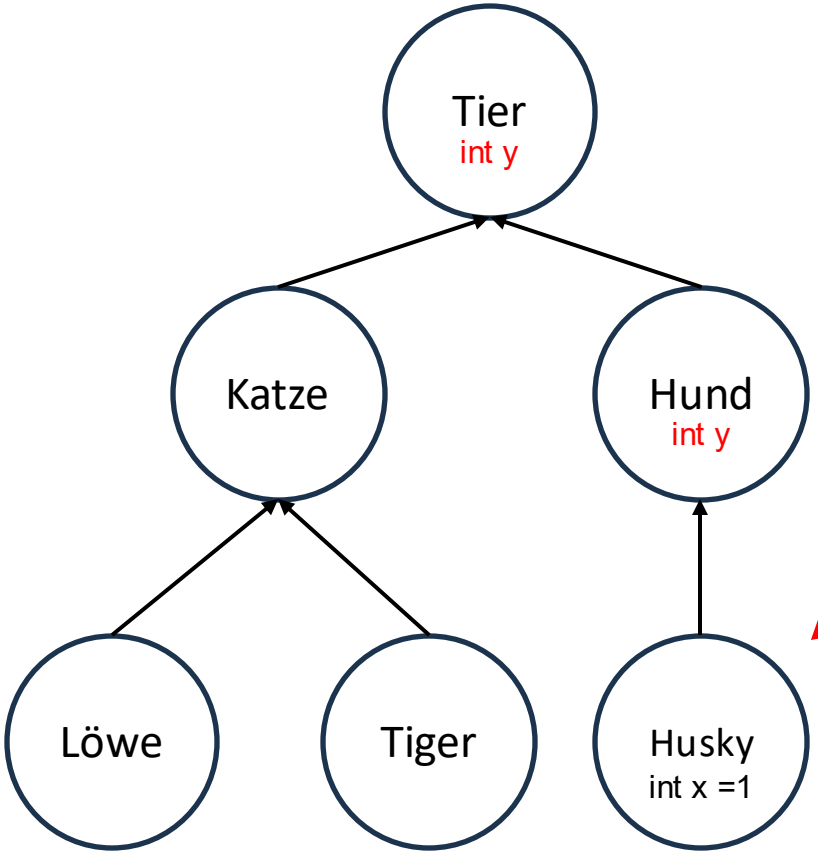
Die Pfeile sind eine “erbt von” - Beziehung

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

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

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

Konstruktoren - Beispiel



Die Pfeile sind eine “erbt von” - Beziehung

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

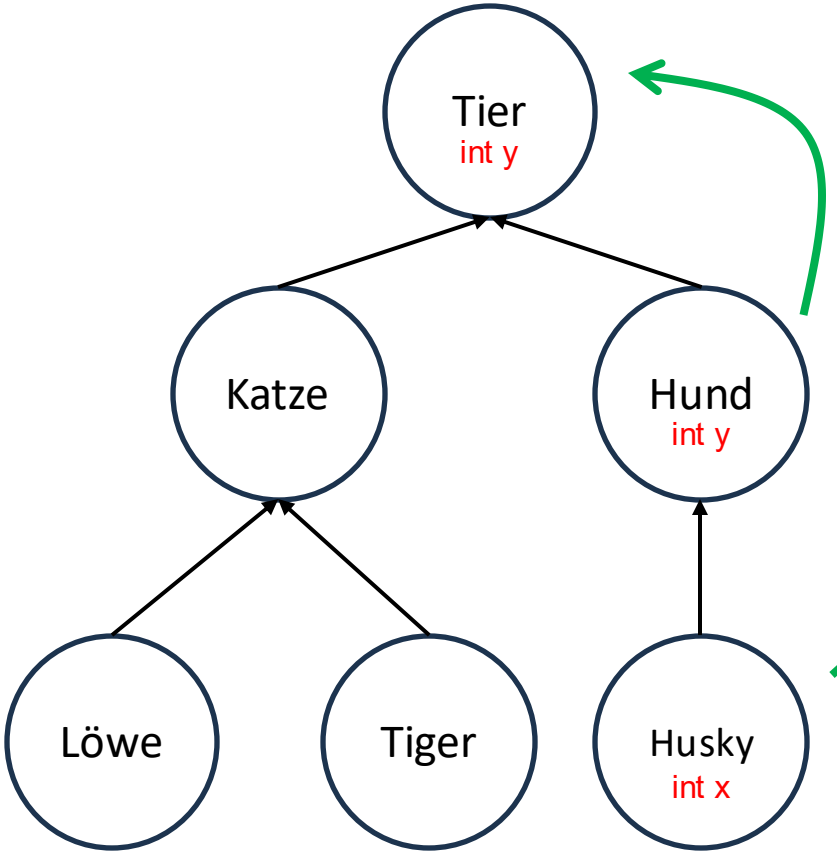
Wie können wir etwas vererben, was nicht gesetzt wurde?

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

Konstruktoren

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

Konstruktoren - Beispiel



super()

super()

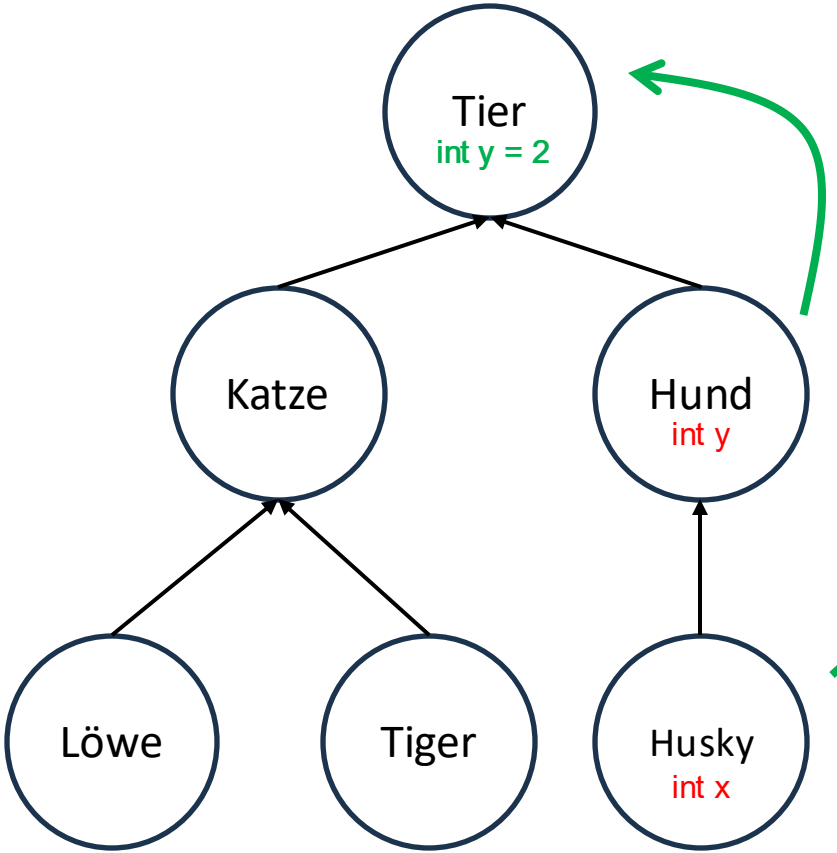
```
Husky h2 = new Husky()
println(h2.y)
```

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

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

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

Konstruktoren - Beispiel



super()

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

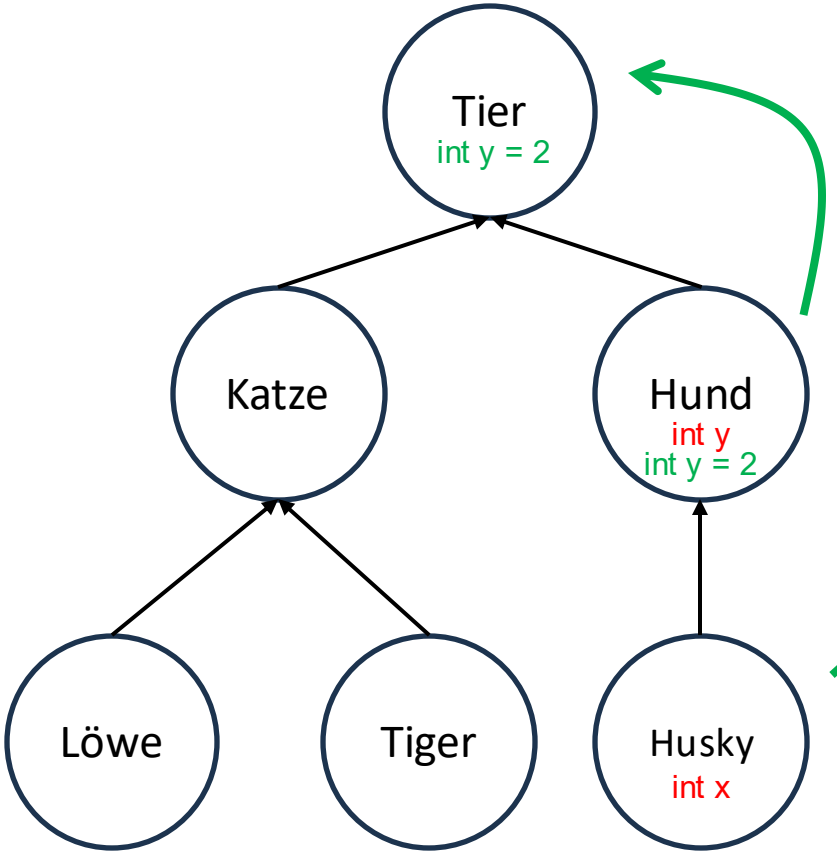
super()

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

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

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

Konstruktoren - Beispiel



super()

super()

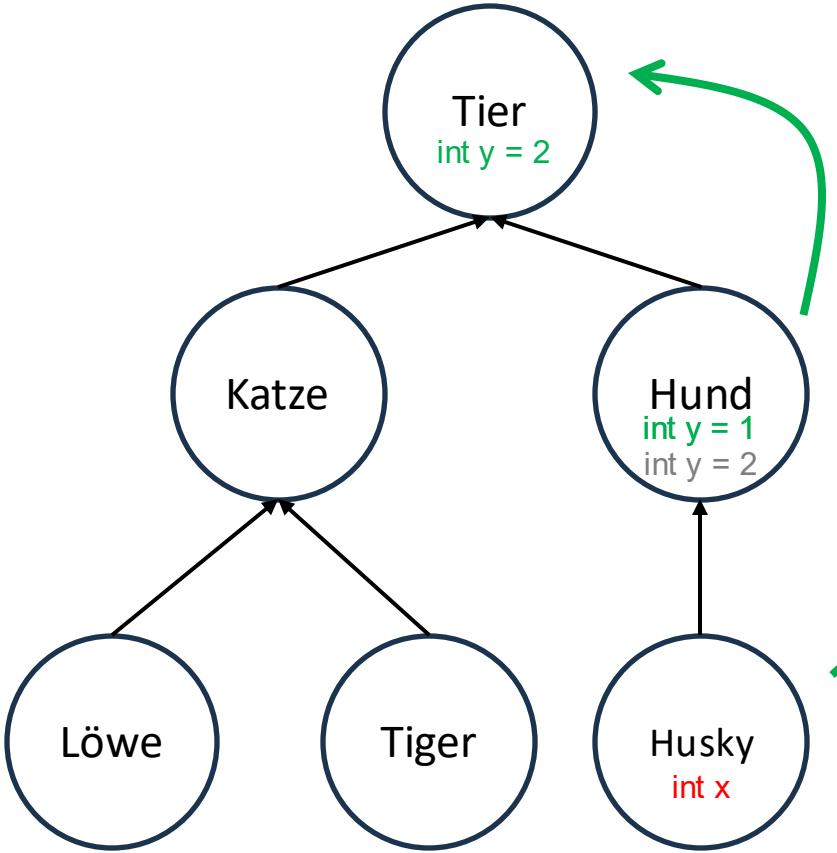
```
Husky h2 = new Husky()
println(h2.y)
```

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

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

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

Konstruktoren - Beispiel



super()

super()

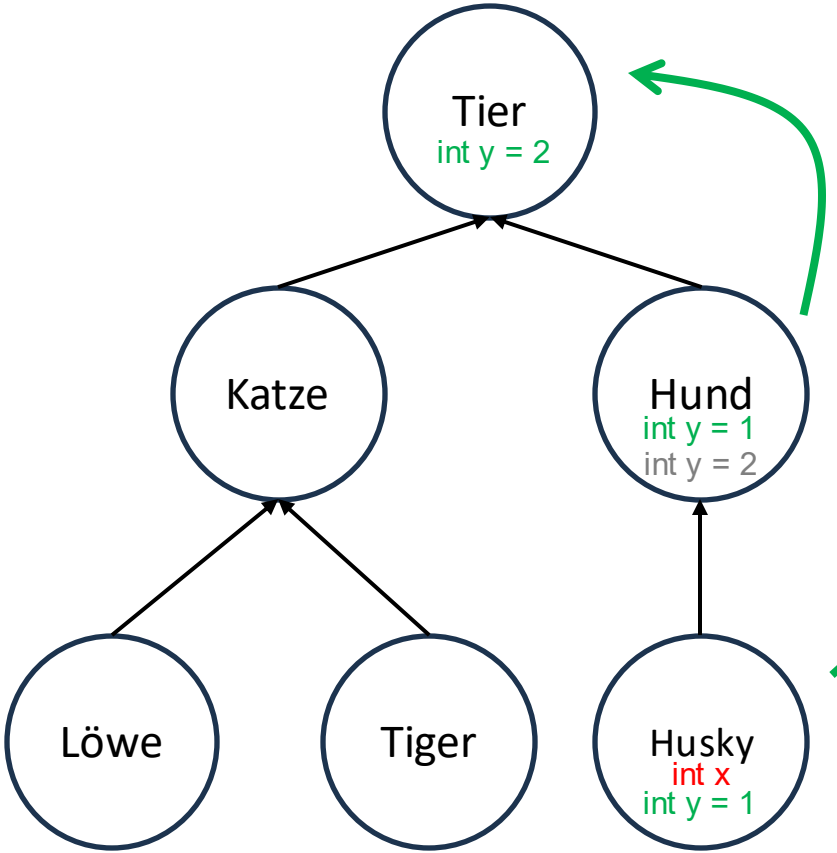
```
public Tier(){
    this.y = 2;
}
```

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

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

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

Konstruktoren - Beispiel



super()

super()

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

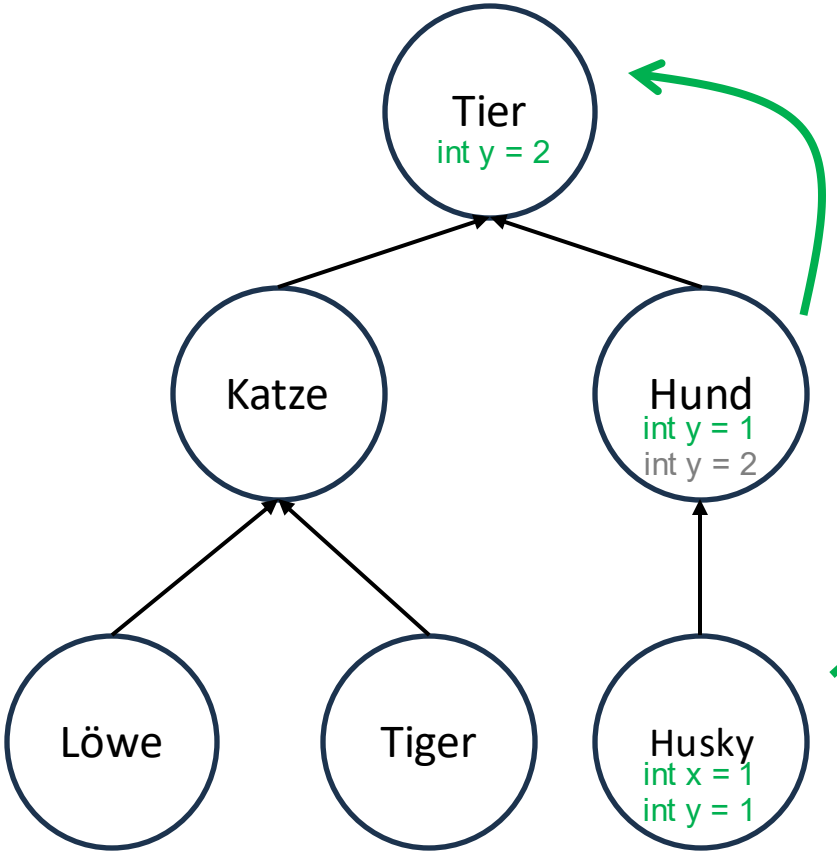
```
println(h2.y)
```

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

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

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

Konstruktoren - Beispiel



super()

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

super()

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

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

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

1

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

Konstruktoren

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

Parametrisierte Konstruktoren

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

Beispiel

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

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

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

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

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

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

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

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

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

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

Interfaces

Interfaces

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

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

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

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

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

```
public class Auto {
}
```

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

Wir können auch mehrere Interfaces implementieren

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

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

Wir können auch mehrere Interfaces implementieren

Interfaces: Intuition

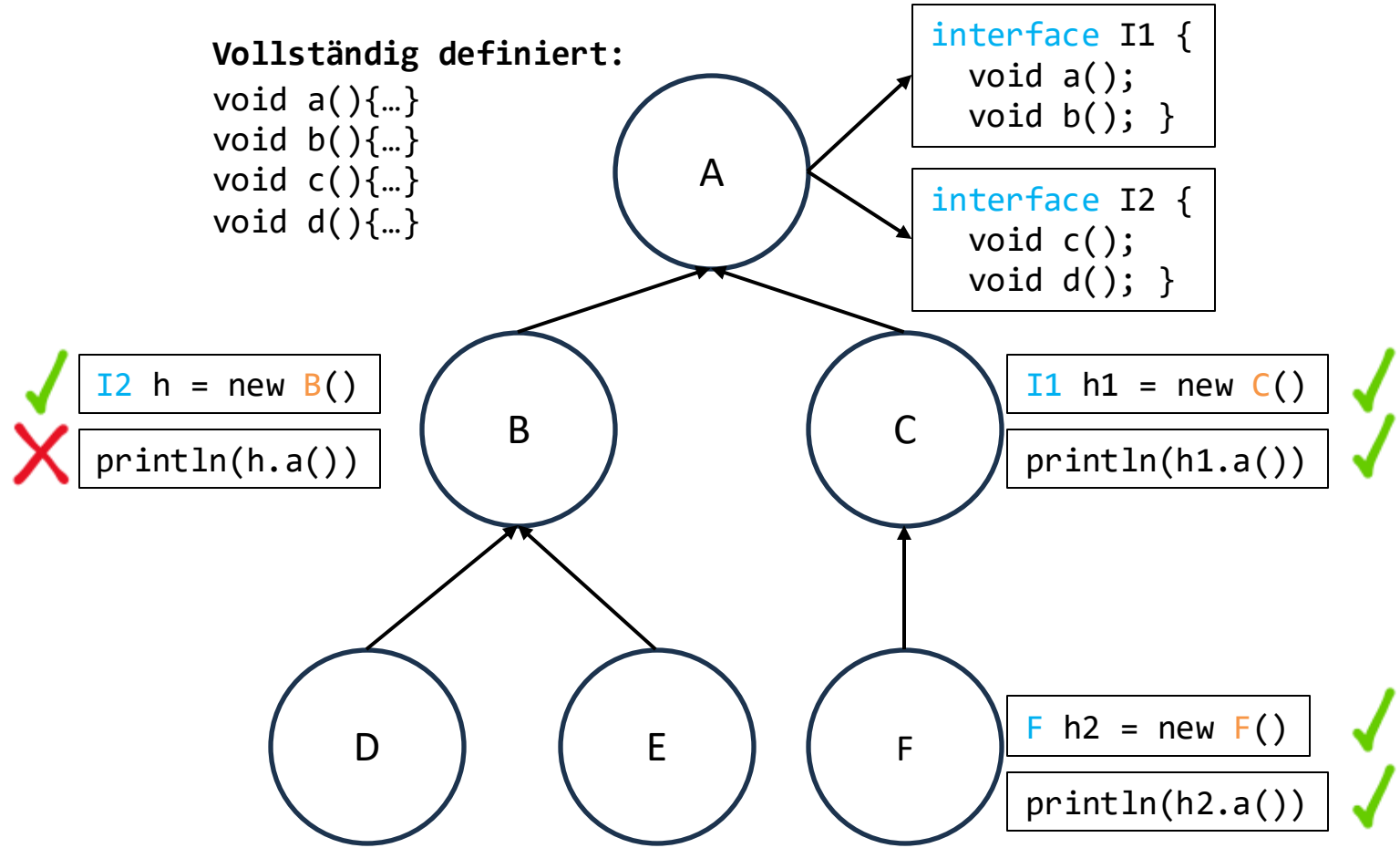
- **Definition:**
  - Ein Interface definiert einheitliche Regeln für Klassen.
- **Grundidee:**
  - Klassen müssen eine vorgegebene Grundstruktur erfüllen.
  - Die Implementierung der Details bleibt der Klasse überlassen.

Interfaces: Aus Sicht des Compilers

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

Vollständig definiert:

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



Dynamic Binding müsste eigentlich funktionieren, oder?

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

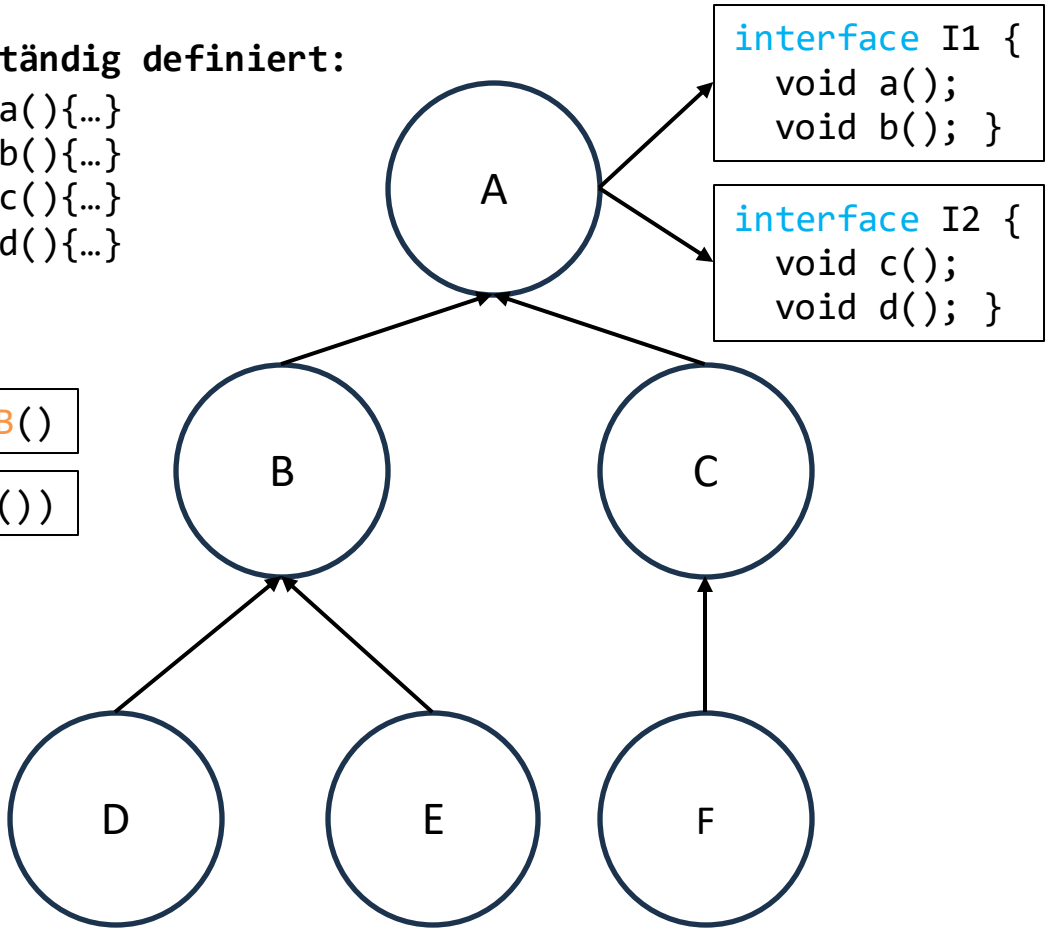
Dies führt zu einem **Compiler-Fehler**!

Vollständig definiert:

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

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

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



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

public class Base {
    int x = 100;

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

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

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

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

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

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


public class S extends T {

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

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

public class X extends Base {
    int x = 600;

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

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

B m1 x=100

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

B m1 x=100

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

Compile-Fehler

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

Compile-Fehler

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

R m1 i=2

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

R m1 x=400

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

S m1 i=3

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

S m1 x=200

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

Compile-Fehler

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

class A1 {
    void setup() {
        System.out.println("Setup A");
    }
}

class B1  extends A1 implements I1 {

    void setup() {
        super.setup();
        System.out.println("Setup B");
    }
    public void value() {
        System.out.println("Value B");
    }
}
```

```
class X1 extends A1 {

    void setup() {
        System.out.println("Setup X");
        value();
    }
    void value() {
        System.out.println("Value X");
    }

}

class Y1 extends X1 {

    void value() {
        System.out.println("Value Y");
    }
    void clear() {
        System.out.println("Clear X");
    }

}
```

```
B1 b = new B1();
Y1 y = new Y1();
b = y;
b.setup();
```

Compile(r)-Error

```
Y1 y = new Y1();
A1 a = y;
B1 b = (B1) a;
b.value();
```

Exception (ClassCastException)

```
Y1 y = new Y1();
A1 a = y;
((Y1)a).value();
```

Value Y

```
Y1 y = new Y1();
((X1)y).setup();
```

Setup X
Value Y

```
Y1 y = new Y1();
((X1)y).clear();
```

Compile(r)-Error

```java
class A {
    void method1() {
        System.out.println("A");
        this.method2(); // dynamic dispatch
    }

    void method2() {
        System.out.println("A");
    }
}

class B extends A {
    @Override
    void method2() {
        System.out.println("B");
        super.method2(); // was ist in diesem Kontext das super keyword?
    }
}

class C extends B {
    @Override
    void method2() {
        System.out.println("C");
        super.method2()
    }
}

public class Test {
    public static void main(String[] args) {
        A obj = new C();
        obj.method1();
    }
}
```

Was ist der Output, wenn das Programm ausgeführt wird?

A
C
B
A

```
System.out.println( (1 % 2) + 3 * ( (double) 9 / 6) + ("5" + 9 % 5 + 4 / 6) );
```

5.5540

```
System.out.println(7 > 6 && (3 + ">" + 2) == "3 > 2" || 4 % 2 == 0  % 0);
```

Exception

Theorie

Loop Detection with Sets

Loop Detection with Sets

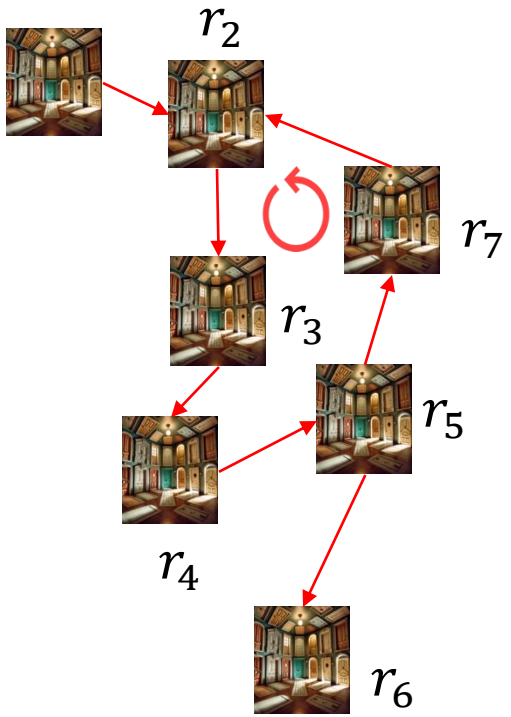
- **HashSet, TreeSet**
- **Für HashSet (oder HashMap) muss immer hashCode() so implementiert, dass gilt:**

Let o_1, o_2 be two different objects.
If o_1.equlas(o_2) == true, then $o1$.hashCode() == $o2$.hashCode()
The other direction does not have to hold (but should to get the best performance)

Zyklen finden

- **Oft:** Annahme, dass keine Zyklen vorkommen.

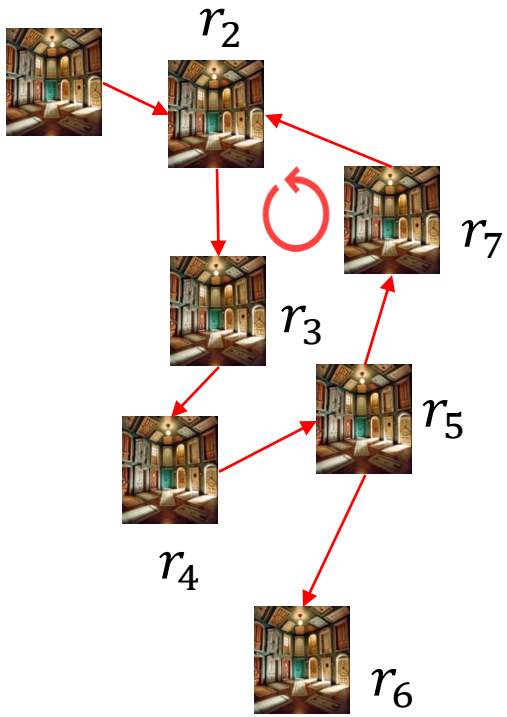
- **Problem:** Was wenn doch Zyklen vorkommen dürfen?



Zyklen finden

- **Option 1:** Modifizieren der Datenstruktur mit einen visited Attribut. *(u08)*

- **Option 2:** Nutzen von Sets um besuchte Nodes zu speichern.



Zyklen finden

- **Wie unterscheiden wir zwei Objekte?**
  - `equals` können wir nicht immer nutzen!

```
public boolean equals(Object o) {
        if(o instanceof Room) {
                o = (Room)o;
                if(this.age == o.age) {
                        return true;
                }
        }
        return false;
}
```

`equals(`  `,`  `)`

`age = 2` `Room@29ca901e`

`age = 2` `Room@5025a98f`

`true` 

Zyklen finden

- **Wie unterscheiden wir zwei Objekte?**
  - Hier können wir Referenzen vergleichen!

equals( , )

age = 2
Room@29ca901e

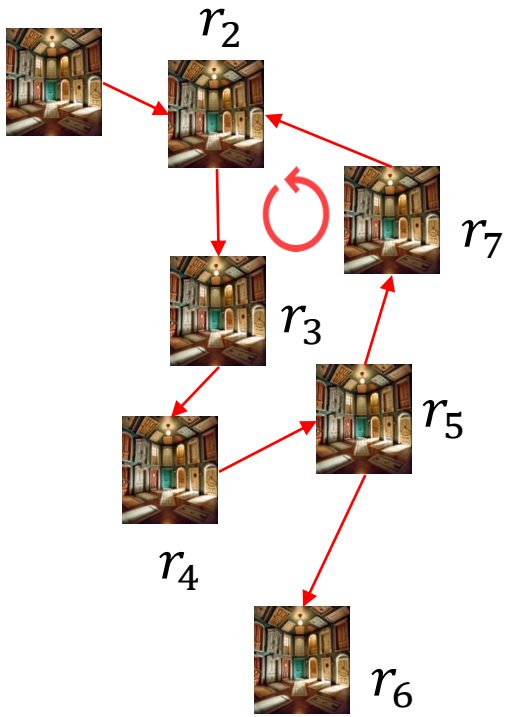
age = 2
Room@5025a98f

false

```
public boolean equals(Object o) {
        if(o instanceof Room) {
                if(this == o) {
                        return true;
                }
        }
        return false;
}
```

Zyklen finden

- **Mit Sets:** Nutzen von `Set<Room>` um alle besuchten Nodes zu speichern.
- **Option 1:** HashSet<Room>
- **Option 2:** TreeSet<Room>

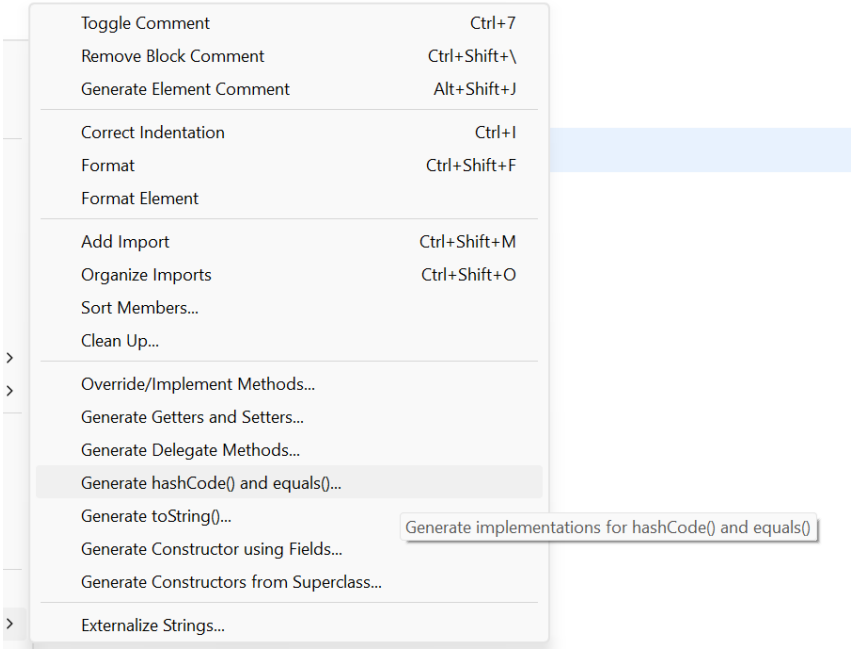


Zyklen finden

- **Option 1:** `HashSet<Room>`
  - Die Methoden `equals` und `hashcode` der `Objekt`-Klasse überschreiben.
- **Option 2:** `TreeSet<Room>`
  - `Comparable`-Interface implementieren und `compareTo`-Methode überschreiben.

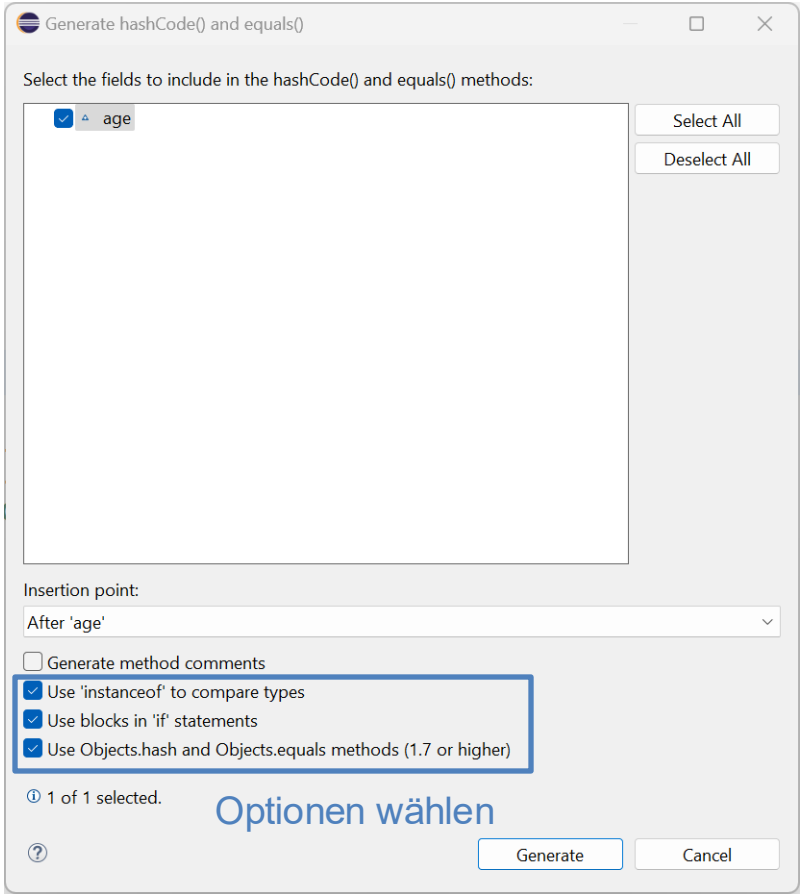
Zyklen finden: HashSet

- **Option 1:** `HashSet<Room>`
  - **In Eclipse:** Rechtsclick -> Source -> Generate hashcode() und equals()...



Zyklen finden: HashSet

- **Option 1:** `HashSet<Room>`
  - **In Eclipse:** Rechtsclick -> Source -> Generate hashcode() und equals()...



Zyklen finden: HashSet

- **Option 1:** `HashSet<Room>`

```java
@Override
public int hashCode() {
    return Objects.hash(age);
}
```

Wir benutzen Operationen und Methoden von Superklassen um `hashCode` zu implementieren.

```java
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (!(obj instanceof Room)) {
        return false;
    }
    Room other = (Room) obj;
    return age == other.age;
}
```

Zyklen finden: HashSet

- **Option 1:** `HashSet<Room>`

```java
@Override
public int hashCode() {
    return Objects.hash(age);
}

@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (!(obj instanceof Room)) {
        return false;
    }
    Room other = (Room) obj;
    return age == other.age;
}
```

Genau gleiche Objekte sollten auch immer equals sein.

Zyklen finden: HashSet

- **Option 1:** `HashSet<Room>`

Objekte von unterschiedlichem Typ sind nie equals.

```
@Override
public int hashCode() {
    return Objects.hash(age);
}

@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (!(obj instanceof Room)) {
        return false;
    }
    Room other = (Room) obj;
    return age == other.age;
}
```

Zyklen finden: HashSet

- **Option 1:** `HashSet<Room>`

```java
@Override
public int hashCode() {
    return Objects.hash(age);
}

@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (!(obj instanceof Room)) {
        return false;
    }
    Room other = (Room) obj;
    return age == other.age;
}
```

Casten damit der Compiler Zugriff auf die Attribute erlaubt.

Zyklen finden: HashSet

- **Option 1:** `HashSet<Room>`

```
@Override
public int hashCode() {
    return Objects.hash(age);
}

@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (!(obj instanceof Room)) {
        return false;
    }
    Room other = (Room) obj;
    return age == other.age;
}
```

Attribute vergleichen.

Zyklen finden: TreeSet

- **Option 2:** `TreeSet<Room>`

Damit TreeSet funktioniert muss Room das Comparable-Interface implementieren.

```
public class Room implements Comparable<Room>{

    int age;

    @Override
    public int compareTo(Room o) {
        // TODO Auto-generated method stub
        return 0;
    }

}
```

Zyklen finden: TreeSet

- **Option 2:** `TreeSet<Room>`

```
public class Room implements Comparable<Room>{

    int age;

    @Override
    public int compareTo(Room o) {
        // TODO Auto-generated method stub
        return 0;
    }

}
```

Jetzt müssen wir nur noch die `compareTo`-Methode sinnvoll implementieren.

Zyklen finden: TreeSet

- **Option 2:** `TreeSet<Room>`
  - `compareTo` gibt 0 zurück, falls die Objekte `equals` sind.
  - `compareTo` gibt 1 zurück, falls `this`-Objekt „grösser“ als das Parameter-Objekt ist.
  - `compareTo` gibt -1 zurück, falls `this`-Objekt „kleiner“ als das Parameter-Objekt ist.

Zyklen finden: TreeSet

- **Option 2:** `TreeSet<Room>`

```
@Override
public int compareTo(Room o) {
    if(this.equals(o)) {
        return 0;
    } else if(this.age > o.age) {
        return 1;
    } else {
        return -1;
    }
}
```

Zyklen finden: TreeSet

- **Option 2:** TreeSet<Room>
  - Wenn Comparable implementiert ist, dann können wir mit `Collections.sort` sortieren.
  - Wenn Comparable implementiert ist, dann können wir mittels `PriorityQueue<Room>` einen Heap erstellen.
  - Mit `Collections.reverseOrder()` können wir von aufsteigender Ordnung zu absteigender Ordnung wechseln.

Zyklen finden: TreeSet

- **Option 2:** TreeSet<Room>
  - Wenn Comparable implementiert ist, dann können wir mit `Collections.sort` sortieren.

```
public void sortList(List<Room> rooms) {
    Collections.sort(rooms);
}
```

Zyklen finden: TreeSet

- **Option 2:** TreeSet<Room>
  - Wenn Comparable implementiert ist, dann können wir mittels `PriorityQueue<Room>` einen Heap erstellen.

```
public PriorityQueue<Room> priorityQueueFromList(List<Room> rooms) {
    PriorityQueue<Room> pQueue = new PriorityQueue<Room>(rooms);
    return pQueue;
}
```

Zyklen finden: TreeSet

- **Option 2:** TreeSet<Room>
  - Mit `Collections.reverseOrder()` können wir von aufsteigender Ordnung zu absteigender Ordnung wechseln.

```
public void sortListReversed(List<Room> rooms) {
    Collections.sort(rooms, Collections.reverseOrder());
}
```

Maps

Maps

- **Option 1:** HashMap
  - Die Methoden `equals` und `hashcode` der `Objekt`-Klasse überschreiben.
- **Option 2:** TreeMap
  - `Comparable`-Interface implementieren und `compareTo`-Methode überschreiben.

Vorbesprechung

Aufgabe 1: Loop-Invariante

Gegeben ist die Methode `findLargestSmaller(int[] al, int value)`, die in einem *sortierten nicht-leeren* Array von `int`-Werten den grössten Wert findet, der strikt kleiner als `value` ist. `value` muss strikt grösser als `al[0]` sein.

Was ist die Invariante für den Loop in der Methode `findLargestSmaller`? Wenn die Elemente `a[0]` .... `a[K]` des Arrays a sortiert sind, dann können Sie das mit `Sorted(a, 0, K)` abkürzen.

```
static int findLargestSmaller(int[] al, int value) {
    // Precondition: Sorted(al, 0, al.length-1) && al.length >= 1 && value > al[0]
    int candidate  = al[0];
    int next = 1;

    // Loop Invariante:
    while (next < al.length && value > al[next] ) {
        candidate = al[next];
        next++;
    }

    // Postcondition: Sorted(al, 0, al.length-1) && al.length >= 1 &&
        ((next < al.length && value <= al[next] && candidate == al[next-1]) ||
        (next == al.length && candidate == al[al.length-1] && value > candidate))

    return candidate;
}
```

Aufgabe 2: Maps

1. Implementieren Sie eine Methode `U11Map.arrayToMap(String[] A)`, die einen Array `A` von Strings als Parameter akzeptiert und eine Map `M` von String zu Integer zurückgibt. Jeder String, der in `A` auftritt, soll in `M` auf die Zahl `0` abgebildet werden. Sie können davon ausgehen, dass `A` nicht `null` ist und dass kein String der empty (leere) String ist.

   Zum Beispiel gibt `arrayToMap` für den Array `[one, two, three, one]` die Map `{one=0, two=0, three=0}` zurück.

2. Implementieren Sie eine Methode `U11Map.arrayToMapOne(String[] A)`, die einen Array `A` von Strings als Parameter akzeptiert und eine Map `M` von String zu Integer zurückgibt. Jeder String, der in `A` auftritt, soll in `M`, falls der String nur einmal vorkommt, auf die Zahl `0` abgebildet werden und, falls der String mehrfach vorkommt, auf die Zahl `1` abgebildet werden. Sie können davon ausgehen, dass `A` nicht `null` ist und dass kein String der empty (leere) String ist.

   Zum Beispiel gibt `arrayToMapOne` für `[one, two, three, one]` die Map `{one=1, two=0, three=0}` zurück und für `[one, two, three, one, one, four, two]` die Map `{four=0, one=1, two=1, three=0}`.

Aufgabe 3: Generische Listen

In dieser Aufgabe implementieren Sie eine generische verkettete Liste. Anhang A zeigt eine generische Version eines Interfaces für Listen. Vervollständigen Sie die Klasse `MyListImpl`, sodass die Klasse das `MyList` Interface implementiert. Dem Interface wurden zwei neue Methoden hinzugefügt. Die Methode `MyListNode getNode(int index)` gibt den Knoten zurück, welcher den Wert der Liste an Position `index` speichert. `MyListeNode` ist selber ein Interface (siehe Anhang B) mit Methoden, welche jeweils den gespeicherten Wert des Knoten, den nächsten Knoten, und ob es einen nächsten Knoten gibt zurückgeben. Damit überprüfen wir, dass `MyListImpl` eine verkettete Liste implementiert. Die Methode `Iterator<T> iterator()` gibt einen Iterator für die Datenstruktur zurück. Implementieren Sie einen neuen Iterator, das heisst eine Klasse, welche das `Iterator` Interface implementiert, und geben Sie nicht den Iterator einer anderen Datenstruktur zurück (zum Beispiel den Iterator einer ArrayList). Dies können wir in den Tests der Korrektur testen. Die `void remove()` Methode vom Iterator wird nicht benötigt. Die Methode `void addAll(MyList<T> other)` sollte eine konstante Laufzeit haben. Ein paar Tests finden Sie in `MyListTest`.

Aufgabe 4: Notenauswertung

Die Klasse `Service` stellt verschiedene Analysen für Prüfungsergebnisse von S Studierenden zur Verfügung. Die Liste von Ergebnissen besteht aus S Einträgen, also jeweils ein Eintrag pro Student/in. Jeder Eintrag besteht aus einer Zeile und enthält (in dieser Reihenfolge):

1. die Immatrikulationsnummer des Studierenden (ein identifizierender positiver `int`-Wert)
2. drei Noten (drei reelle Zahlen im Bereich von 1.0 bis 6.0, getrennt durch Leerzeichen)

Die drei Noten gehören zu den Fächern *Fach 1*, *Fach 2* und *Fach 3*. Zusätzliche Leerzeilen und -zeichen sollen ignoriert werden. Eine Beispiel für eine Liste für 3 Studierende ist:

```
111111004  5.0  5.0  6.0
111111005  3.75 3.0  4.0
111111006  4.5  2.25 4.0
```

Ihre Aufgabe ist es nun, die `Service`-Klasse und ihre Analysen zu implementieren. Die `Service`-Klasse hat einen Konstruktor, welcher alle Prüfungsergebnisse aus einem `Scanner` auslesen und damit das `Service`-Objekt initialisieren soll. Das Objekt soll so initialisiert werden, dass die vorgegebenen Methoden ihre Analysen durchführen können. Sie dürfen dabei Attribute und zusätzliche Methoden frei bestimmen.

Aufgabe 4: Notenauswertung

a) Implementieren Sie nun die Methode `critical()`, welche die zwei Argumente `bound1` und `bound2` erwartet. Die Methode sucht alle "kritischen" Fälle und gibt eine Liste dieser Studierenden zurück. Ein Student darf maximal einmal in der Liste vorkommen. Die zurückgegebene Liste besteht aus den Immatrikulationsnummern dieser Studierenden (in beliebiger Reihenfolge).

   Ein/e Student/in gilt als kritisch, wenn die Note in *Fach 1* $\leq$ `bound1` und die Summe der Noten für *Fach 2* und *Fach 3* kleiner als `bound2` ist.

   Für das obige Beispiel gäbe `critical(4, 8)` eine Liste mit dem Element `111111005` zurück.

b) Implementieren Sie nun die Methode `top()`, welche die Studierenden mit den besten Ergebnissen zurückgeben soll. Der Parameter `limit` bestimmt die maximale Anzahl der zurückzugebenden Studierenden. Falls weniger Ergebnisse als `limit` existieren, sollen einfach alle gefundenen zurückgegeben werden.

   Der Rückgabewert der Methode ist wieder eine Liste der Immatrikulationsnummern. Ein Student darf maximal einmal in der Liste vorkommen. Diese Liste soll absteigend nach der Leistung sortiert sein (der/die Student/in mit dem besten Ergebnis zuerst). Dabei gilt, dass ein Ergebnis A besser ist als ein Ergebnis B, wenn die Summe aller Noten von A grösser ist als die Summe der Noten von B. Sind die Summen gleich, sind die Ergebnisse gleich gut (und die Reihenfolge in der Liste somit egal).

   Für das obige Beispiel gäbe `top(2)` entweder die Liste `[111111004, 111111006]` oder die Liste `[111111004, 111111005]` zurück (beide wären richtig).

Aufgabe 5: Interpreter

In der letzten Übung implementierten Sie einen Evaluator für mathematische Ausdrücke. In dieser Aufgabe erweitern Sie ihn so, dass er statt einzelnen Ausdrücken einfache Programme mit mehreren Anweisungen auswertet. Das nennt man auch *interpretieren* und ein solches Programm entsprechend *Interpreter*.

```
digit  ← 0 | 1 | ... | 9
char   ← A | B | ... | Z | a | b | ... | z
num    ← digit { digit } [ . digit { digit } ]
var    ← char { char }
func   ← char { char } (
op     ← + | - | * | / | ^
atom   ← num | var
term   ← ( expr ) | func expr ) | atom
expr   ← term [ op term ]
stmt   ← var = expr ;
prog   ← { stmt }
```

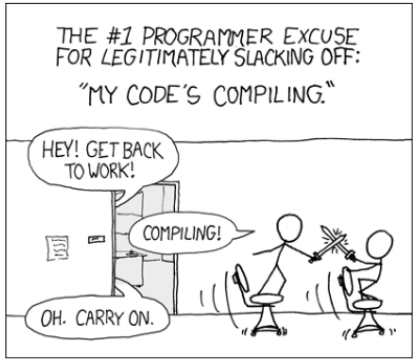
Abbildung 1: **EBNF-Beschreibung** von *prog*

```
alpha = i * ((2*PI) * (1 / 6.05));
size = (0.25 * cos(t/2)) + 0.75;

x = cos(alpha + (0.3 * t)) * size;
y = sin((1.5 * alpha) + t) * size;

r = (cos(alpha + (2 * t)) + 1) / 2;
g = (sin(alpha + (2 * t)) + 1) / 2;
b = (cos(alpha +  (PI/2)) + 1) / 2;
```

Abbildung 2: Beispiel-Programm

Aufgabe 6: Compiler

Das Interpretieren von Quellcode ist ineffizient und langsam. Deshalb werden Java-Programme auch zuerst *kompiliert*, bevor sie ausgeführt werden. Kompilieren heisst, den Quellcode in eine Form zu übersetzen, die vom Computer direkt(er) ausgeführt werden kann. In dieser Übung schreiben Sie einen einfachen Compiler, der den Quellcode von Programmen von Aufgabe 5 in eine Serie von Instruktionen übersetzt, die effizient ausgeführt werden können.



xkcd: Compiling by Randall Munroe (CC BY-NC 2.5)

Die Programmiersprache in Aufgabe 5 hat eine rekursive Struktur: Ausdrücke können mehrere Ausdrücke enthalten, welche wiederum mehrere Ausdrücke enthalten können, usw. Um eine solche Struktur in eine lineare Folge von Instruktionen umzuwandeln, verwenden wir einen *Operanden-Stack*. Dies ist ein `Stack` (wie Sie Ihn in der Vorlesung gesehen haben), der Zwischenresultate von Berechnungen speichert. Instruktionen können Werte auf den Stack "pushen" oder Werte ab dem Stack "poppen" und verwenden. Es gibt folgende Arten von Instruktionen:

Aufgabe 6: Compiler

Programmteil	Instruktionen
`b`	`LOAD b`
`1`	`CONST 1`
`b + 1`	`LOAD b CONST 1 OP +`
`(b + 1)`	`LOAD b CONST 1 OP +`
`2`	`CONST 2`
`c`	`LOAD c`
`2 * c`	`CONST 2 LOAD c OP *`
`sin(2 * c)`	`CONST 2 LOAD c OP * FUNC sin`
`(b + 1) / sin(2 * c)`	`LOAD b CONST 1 OP + CONST 2 LOAD c OP * FUNC sin OP /`
`a = (b + 1) / sin(2 * c);`	`LOAD b CONST 1 OP + CONST 2 LOAD c OP * FUNC sin OP / STORE a`

Tabelle 1: Kompilieren der Anweisung `a = (b + 1) / sin(2 * c);`

`CONST` c	Pusht den konstanten Wert c auf den Stack
`LOAD` v	Lädt den Wert der Variable v und pusht ihn auf den Stack
`STORE` v	Poppt einen Wert vom Stack und speichert ihn in der Variable v
`OP` $\oplus$	Poppt zwei Werte r und l vom Stack (zuerst r, dann l) berechnet $l \oplus r$ und pusht das Resultat zurück auf den Stack
`FUNC` f	Poppt einen Wert x vom Stack, berechnet $f(x)$ und pusht das Resultat zurück auf den Stack

Unten sehen Sie ein kleines Programm, das aus solchen Instruktionen besteht. Es lädt zuerst den Wert der Variable x und dann einen konstanten Wert 2 auf den Stack. Die nächste Instruktion holt sich die beiden Werte vom Stack, multipliziert sie und pusht das Resultat zurück. Die letzte Instruktion schliesslich holt diesen Wert vom Stack und speichert ihn zurück in die Variable x:

```
LOAD x
CONST 2
OP *
STORE x
```

Aufgabe 7: EBNF-Baum

Achtung: Diese Aufgabe gibt Bonuspunkte (siehe "Leistungskontrolle" im `www.vvz.ethz.ch`). Die Aufgabe muss eigenhändig und alleine gelöst werden. Die Abgabe erfolgt wie gewohnt per Push in Ihr Git-Repository auf dem ETH-Server. Verbindlich ist der letzte Push vor dem Abgabetermin. Auch wenn Sie vor der Deadline committen, aber nach der Deadline pushen, gilt dies als eine zu späte Abgabe. Bitte lesen Sie zusätzlich die allgemeinen Regeln.

Nachbesprechung

Aufgabe 1: Loop-Invariante

Gegeben den Pre- und Postcondition formulieren Sie eine Loop-Invariante in der Datei "LoopInvariante.txt" für die folgenden Programme.

1. Um die Loop-Invariante einfacher schreiben zu können, dürfen Sie `contains(arr, c)` benutzen. Hier sagt uns `contains(arr, c)`, ob der `Char c` im Array `arr` enhalten ist. Ebenfalls können Sie `subarray(arr, i)` benutzen, welches eine Kopie vom Array `arr` von Index `0` bis und mit `i` darstellt. Alternativ könnte man auch formale Notation benutzen, in dem man mit Quantoren arbeitet.

```
void erase(char[] arr, char c) {
  // Precondition: arr != null && c != 'x'
  int i = 0;

  // Loop-Invariante:
  while (i != arr.length) {
    if (arr[i] == c) {
      arr[i] = 'x';
    }

    i++;
  }

  // Postcondition: !contains(arr, c)
}
```

Aufgabe 1: Loop-Invariante

2.
```
public int compute(String s, char c) {
    int x;
    int i;

    x = 0;
    i = -1;

    // Loop-Invariante:
    while (x < s.length() && i < 0) {
      if (s.charAt(x) == c) {
        i = x;
      }
      x = x + 1;
    }

    // Postcondition:
    //   (0 <= i && i < s.length() && s.charAt(i) == c) || count(s, c) == 0
    return i;
  }
```

Die Methode `count(String s, char c)` gibt zurück wie oft der Character `c` im String `s` vorkommt. Schreiben Sie die Loop Invariante in die Datei "LoopInvariante.txt". **Achtung:** Die Aufgabe ist schwerer als es zuerst scheint. Überprüfen Sie Ihre Lösung sorgfältig.

Aufgabe 2: Cyclic List

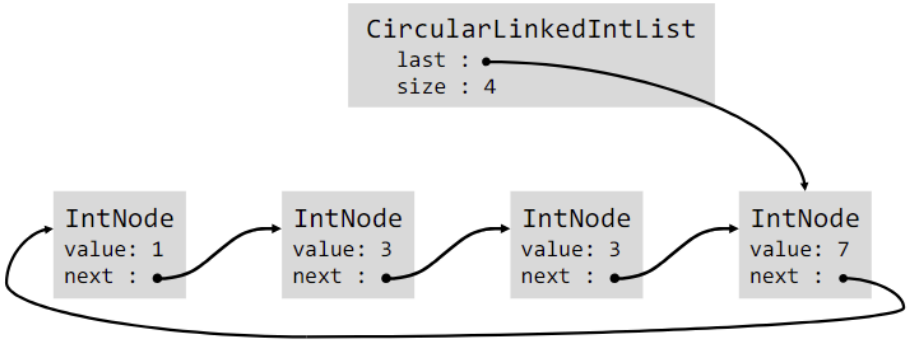


Abbildung 1: Zyklische Liste mit Werten 1, 3, 3, 7.

Aufgabe 3: Expression Evaluator

In dieser und in folgenden Übungen werden Sie eine Reihe von Programmen schreiben, welche andere Programme interpretieren, kompilieren oder (in kompilierter Form) ausführen. Die Programmiersprachen definieren wir selber.

Als Einstieg schreiben Sie ein Programm, welches mathematische Ausdrücke (*expressions*) auswertet. Die Ausdrücke bestehen aus Zahlen, Variablen, Operatoren wie + oder − und einfachen Funktionen wie sin() oder cos(). Die genaue Syntax für diese Ausdrücke finden Sie als EBNF-Beschreibung in Abbildung 2.

```
digit  ⇐ 0 | 1 | ... | 9
char   ⇐ A | B | ... | Z | a | b | ... | z
num    ⇐ digit { digit } [ . digit { digit } ]
var    ⇐ char { char }
func   ⇐ char { char } (
op     ⇐ + | - | * | / | ^
open   ⇐ (
close  ⇐ )

atom   ⇐ num | var
term   ⇐ open expr close | func expr close | atom
expr   ⇐ term [ op term ]
```

Abbildung 2: **EBNF-Beschreibung** von *expr*

Ein Programm, das Ausdrücke auswertet, muss natürlich entscheiden, ob eine gegebene Zeichenkette überhaupt ein gültiger Ausdruck ist[1]. Das nennt man *parsen* und ein solches Programm heisst *Parser*. Aus einer EBNF-Beschreibung wie dieser kann man einfach einen Parser erstellen[2]:

Aufgabe 4: Contact Tracing

In dieser Aufgabe implementieren Sie eine Contact-Tracing-Applikation, welche es ermöglichen soll, Kontakte während eines Virus-Ausbruches nachzuverfolgen. Ihre Implementierung soll zunächst Begegnungen zwischen verschiedenen `Person`-Instanzen anonym protokollieren, so dass bei einem positivem Test die Benachrichtigung aller Personen möglich ist, die direkt oder indirekt mit einer positiv getesteten Person in Kontakt standen.

Anonyme Begegnungen. Um Anonymität zu gewährleisten, dürfen zwei Personen A und B bei einer Begegnung lediglich anonyme Integer-IDs austauschen, ohne dabei die Identität der jeweils anderen Person aufzudecken. Beide Personen speichern hierbei sowohl die eigene ID als auch die ID der anderen Person. Bei der positiven Testung von A kann dann mithilfe der anonymen IDs, die A genutzt hat, festgestellt werden, ob B einer dieser IDs begegnet ist. Um zu vermeiden, dass wiederkehrende IDs die Identifikation einer Person über mehrere Begegnungen hinweg ermöglichen, benutzt jede Person für jede Begegnung frische IDs, welche über eine zentrale Klasse `ContactTracer` vergeben werden. Frisch bedeutet hierbei, dass eine ID zuvor noch nie bei einer Begegnung verwendet wurde.

Direkte und indirekte Kontakte. Nachdem eine Reihe an Begegnungen protokolliert wurden, wird eine oder mehrere Personen positiv getestet. Mit dem erfassten Netzwerk aus Begegnungen soll Ihre Applikation dann zwei verschiedene Arten an Kontaktpersonen bestimmten:

- Als *direkte Kontakte* gelten alle Personen, die eine Begegnung mit einer positiv getesteten Person hatten.
- Als *indirekte Kontakte* hingegen gelten alle Personen, die zwar selbst keine Begegnung mit einer positiv getesteten Person hatten, jedoch Kontakt mit mindestens einer anderen Person, welche als direkter Kontakt gilt, hatten. Indirekte Kontakte mit mehr als einer Zwischenperson müssen Sie dabei nicht berücksichtigen.

Sie dürfen dabei annehmen, dass zunächst alle Begegnungen erfasst werden und erst dann Personen positiv getestet werden. Nach der ersten positiven Testung finden keine weiteren Begegnungen mehr statt.

Programmieraufgaben

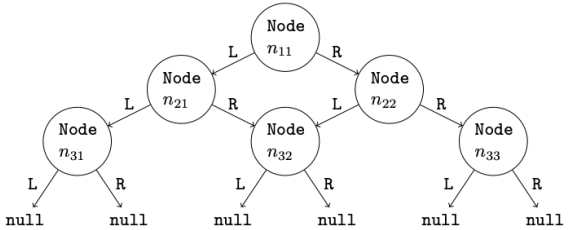
- **Sehr wichtig**
- **Nutzt die Lernphase um Programmieraufgaben zu lösen**
- **Konzentriert euch vor allem auf Prüfungsaufgaben (viele Übungsaufgaben sind alte Prüfungsaufgaben)**

Kahoot

https://create.kahoot.it/details/e0e39832-94c1-420b-bb2b-27f5b20da007

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).

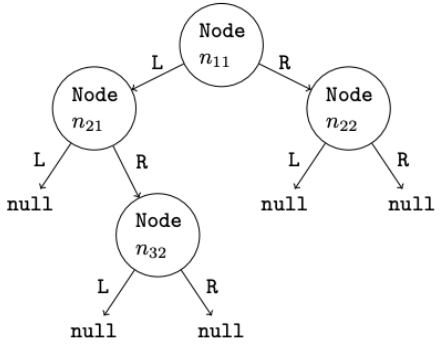
Das Ziel dieser Aufgabe ist, für ein `Node`-Objekt zu entscheiden, ob der durch das `Node`-Objekt definierte Graph einer Pyramide entspricht. Zum Beispiel entspricht der folgende Graph einer Pyramide.

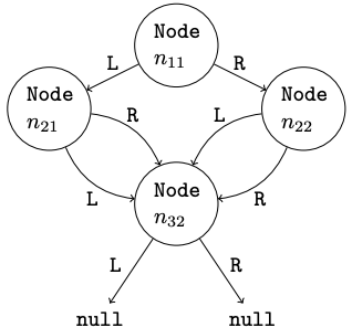


Beachten Sie, dass der rechte Knoten von n_{21} gleich ist wie der linke Knoten von n_{22} (das heisst die Node-Objekte sind gleich!). Ein Graph (wie oben repräsentiert) definiert eine Pyramide genau dann, wenn folgende Bedingungen gelten:

- 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.

```java
public static boolean isPyramid(Node node) {  9 usages  zelleraa *
    Queue<Node> q = new LinkedList<>();
    List<Node> seen = new ArrayList<>(); // or use a set
    int level = 1;
    q.add(node);
    while (!q.isEmpty()) {
        int size = q.size();
        if (size != level) return false;
        for (int i = 0; i < size; i++) {
            Node current = q.poll();
            if (current == null) continue;
            if (seen.contains(current)) return false;
            seen.add(current);
            if (!checkMe(current)) return false;
            if (i == 0) {
                q.add(current.getLeft());
            }
            q.add(current.getRight());
        }
        level++;
    }
    return true;
}
```

```java
public static boolean checkMe(Node node) {  1 usage  new *
    if (node.getLeft() == null || node.getRight() == null) return node.getLeft() == null && node.getRight() == null;
    return node.getLeft().getRight() == node.getRight().getLeft();
}
```