

252-0027

Einführung in die Programmierung

Übungen

Woche 12: Interfaces, Exception

Vorname Nachname

Departement Informatik

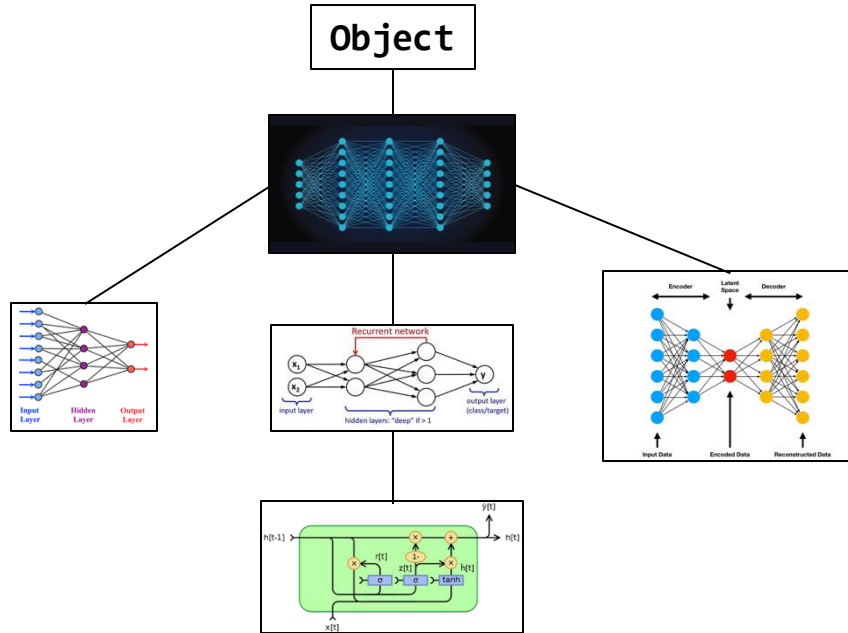
ETH Zürich

Organisatorisches

- Mein Name: *Vorname Nachname*
- Bei Fragen: *ETH - Mail*
 - Mails bitte mit «[EProg25]» 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

Interfaces

Klassen: Neural Network

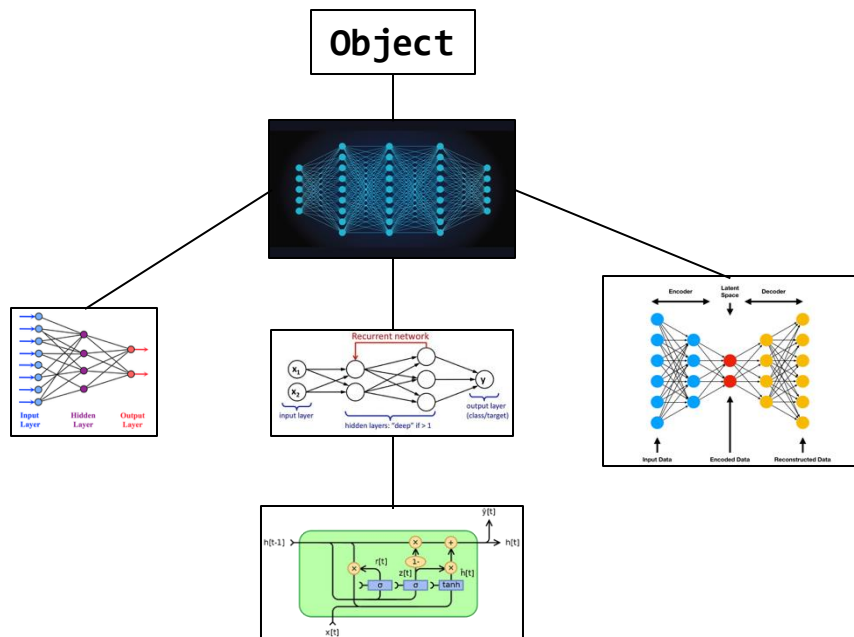


Was macht neuronale Netzwerke aus?

- Nodes, Layers, Weights, etc.
- **train-Methode:** Mit dieser Methode können wir das neuronale Netzwerk trainieren.
- **predict-Methode:** Mit dieser Methode können wir gegeben Inputs einen Output generieren.

Dafür eignet sich ein Interface!

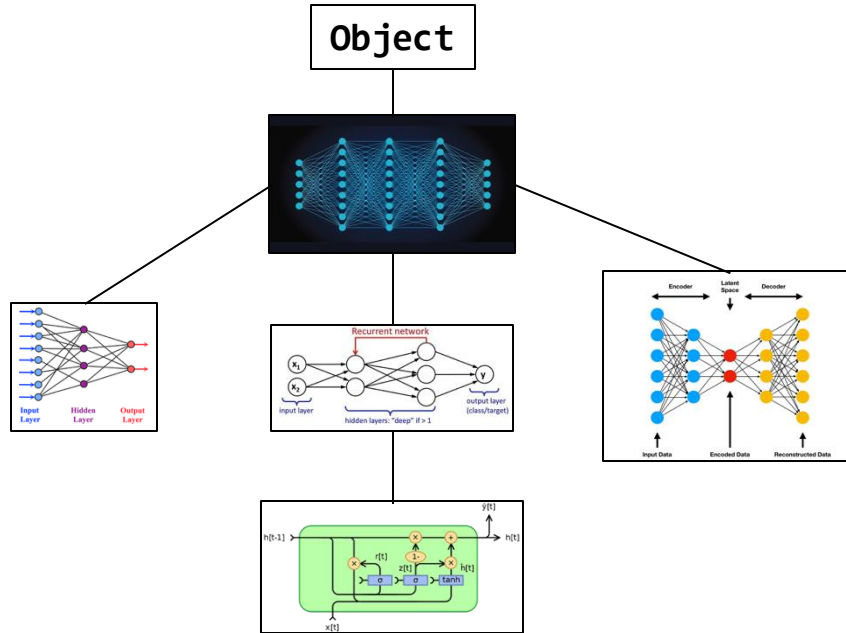
Interfaces: Neural Network



Interfaces:

- Definieren was für ein **Verhalten** eine Klasse haben muss, damit sie das Interface implementiert.
- **Wie** diese Methoden implementiert werden, ist **nicht** Teil des Interfaces.
- Deshalb enthalten Interfaces auch **keine Attribute ausser Konstanten**.
- Jedes Attribut ist public, static und final.

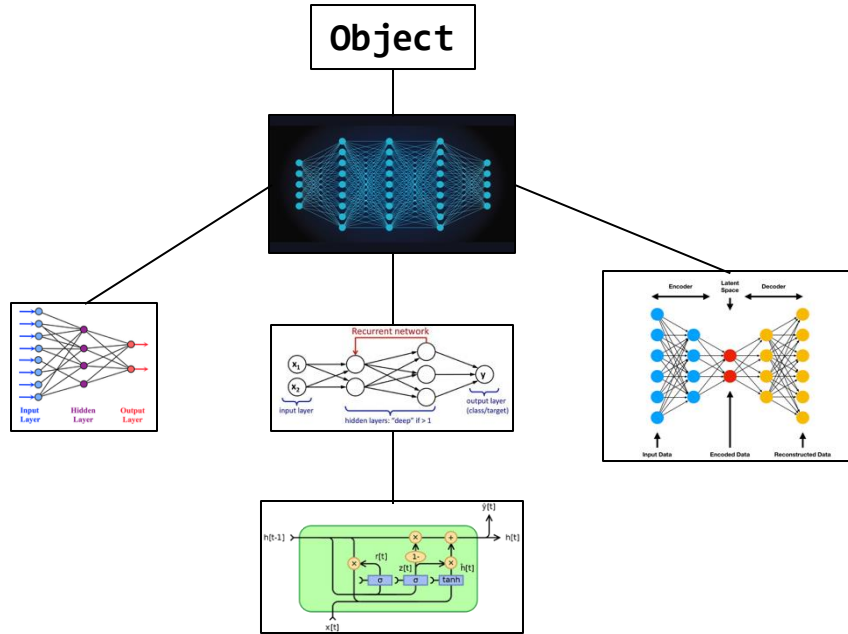
Interfaces: Neural Network



Was macht neuronale Netzwerke aus?

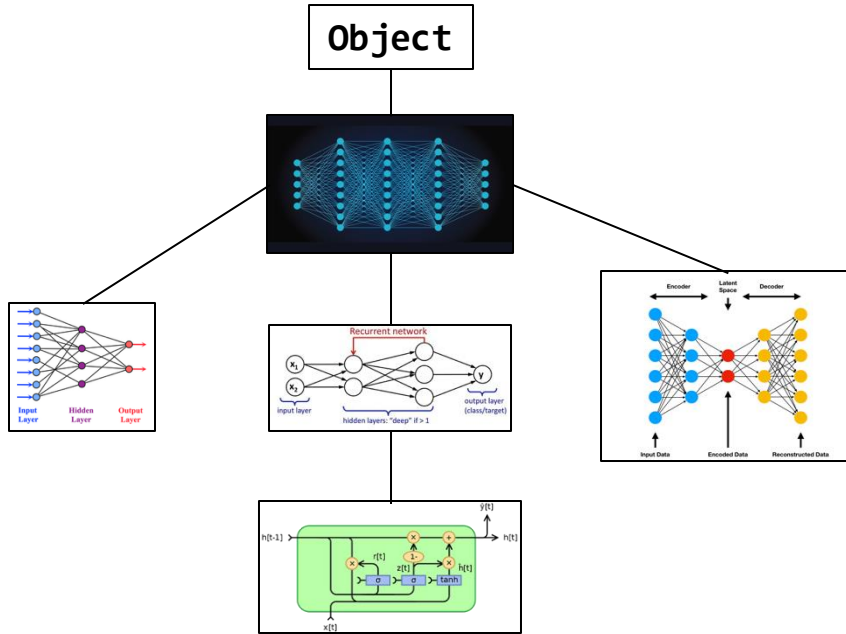
- ~~Nodes, Layers, Weights, etc.~~ **Nicht Teil des Interface.**
- **train-Methode:** Mit dieser Methode können wir das neuronale Netzwerk trainieren.
- **predict-Methode:** Mit dieser Methode können wir gegeben Inputs einen Output generieren.

Interfaces: Neural Network



```
public interface Neural {  
    public void train();  
    public void predict(int[] inputs);  
}
```

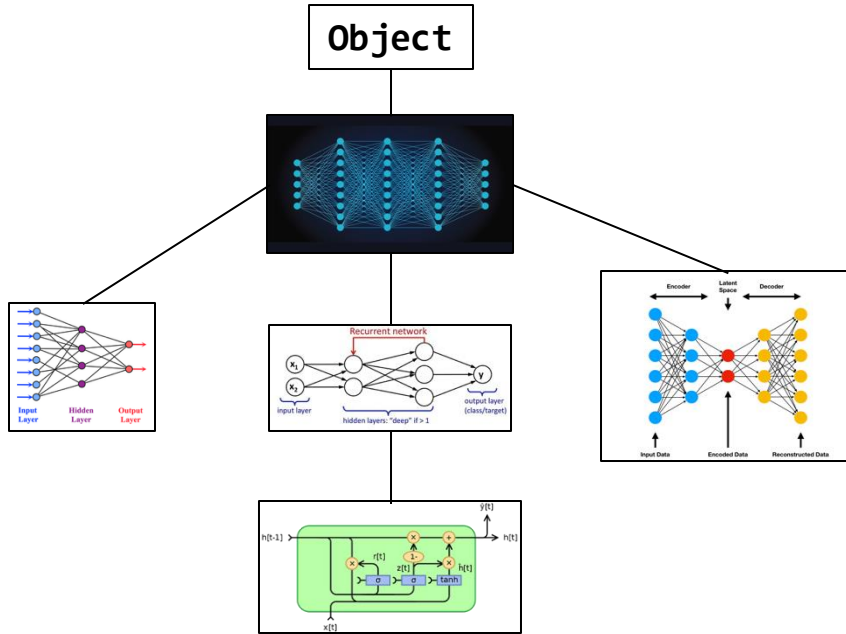
Interfaces: Neural Network



Darf man das?

```
private interface Neural {  
    public void train();  
    public void predict(int[] inputs);  
}
```

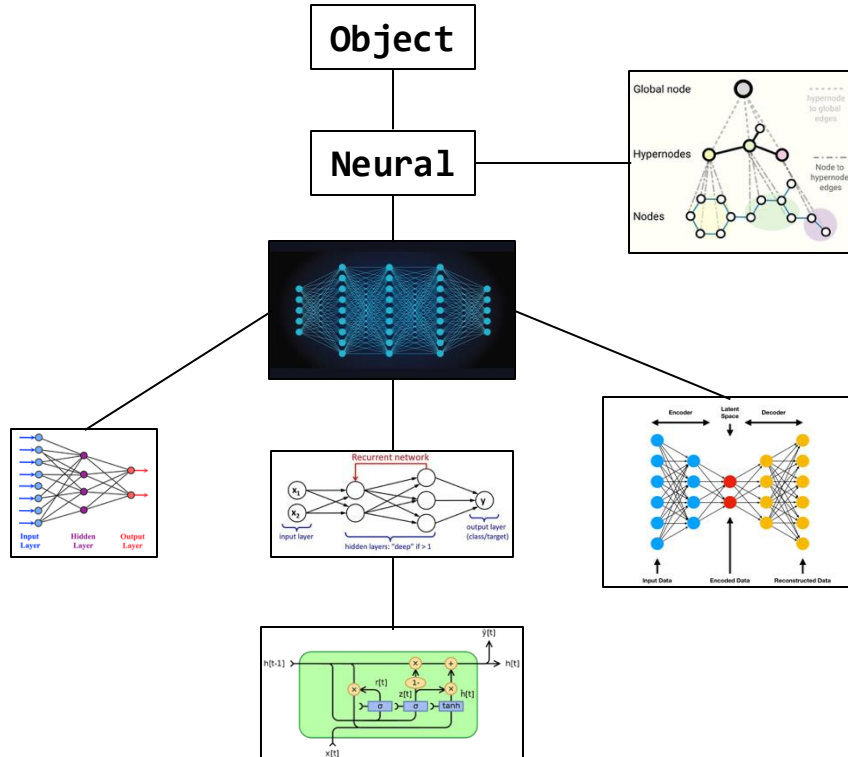

Interfaces: Neural Network



Interfaces müssen einen public oder default modifier haben!

```
private interface Neural {  
    public void train();  
    public void predict(int[] inputs);  
}
```

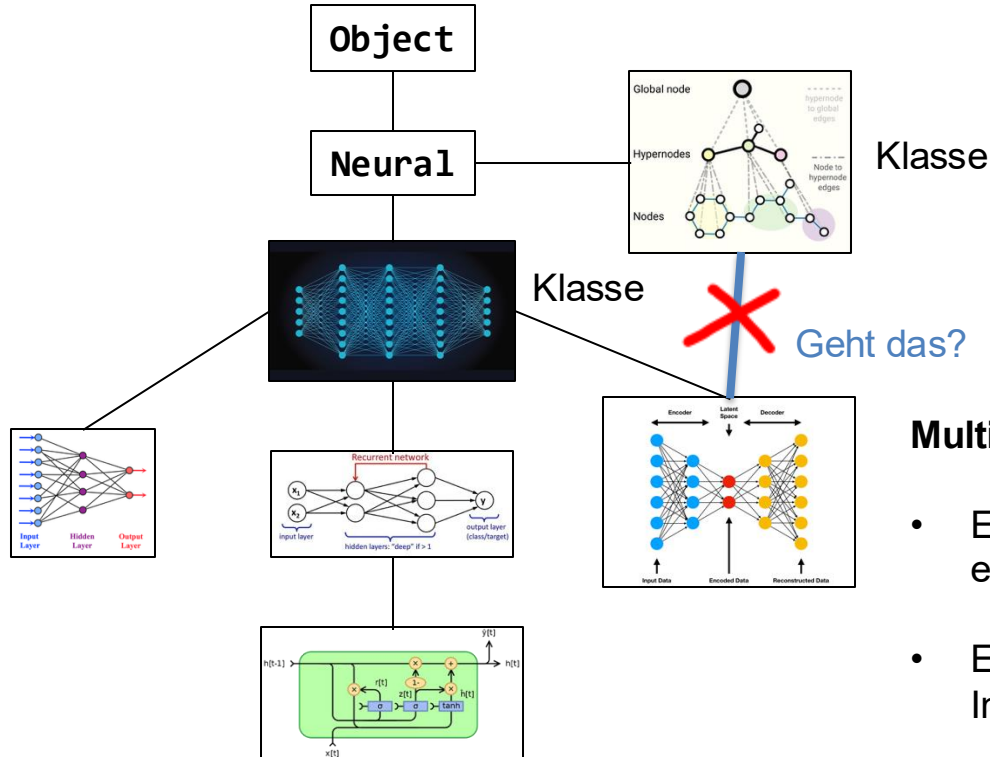
Interfaces: Neural Network



Interfaces:

- Definieren was für ein **Verhalten** eine Klasse haben muss, damit sie das Interface implementiert.
- Definieren wie Klassen einen **Typ**.

Interfaces: Neural Network



Multi-Inheritance:

- Eine Klasse kann höchstens von einer anderen Klasse erben.
- Eine Klasse kann aber mehrere Interfaces implementieren.

Interfaces: ArrayList

Module java.base

Package java.util

Class ArrayList<E>

java.lang.Object

java.util.AbstractCollection<E>

java.util.AbstractList<E>

java.util.ArrayList<E>

Type Parameters:

E - the type of elements in this list

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess, SequencedCollection<E>

ArrayList:

- **Implementiert:** Serializable, Cloneable, Iterable, Collection, List, ...
- **Extends:** AbstractList

Compiler-Fehler vs Exceptions

Zuerst Compile-Fehler,...

- Syntax überprüfen
 - Keine Klammern, Semikolons vergessen?
 - Existiert eine Methode mit diesem Namen und dieser Signatur?
- Typenkompatibilität überprüfen
 - Compiler-Brille (später mehr)
 - `int i = 4.5;`
 - **Präzisionsverlust:** Der Compiler beschwert sich, explizites Casting erforderlich.
 - `String s = (String) new Integer(42);`
 - **Kein Vererbungsverhältnis:** Ein Cast zu Laufzeit würde nie funktionieren.

... dann Exceptions

- Wir überprüfen das Programm auf Logikfehler
 - **ArithmeticException**: Tritt auf bei fehlerhaften mathematischen Operationen (z.B. Division durch Null).
 - **NullPointerException**: Entsteht, wenn auf ein *null*-Objekt zugegriffen wird.
 - **ClassCastException**: Wird geworfen bei einem ungültigen Cast zwischen inkompatiblen Objekten.
 - **ArrayIndexOutOfBoundsException**: Passiert, wenn auf einen ungültigen Index eines Arrays zugegriffen wird.
 - **NumberFormatException**: Tritt auf, wenn versucht wird, einen String in eine Zahl umzuwandeln, der kein korrektes Zahlenformat hat.

Exceptions können sehr spezifisch sein...



```
1 // Eigene Exception
2 class OverheatedException extends Exception {
3     public OverheatedException(String message) {
4         super(message);
5     }
6 }
7
8 class Machine {
9     private int temperature;
10
11     public Machine(int temperature) {
12         this.temperature = temperature;
13     }
14
15     public void checkTemperature() throws OverheatedException {
16         if (temperature > 100) {
17             throw new OverheatedException("Machine is overheated!");
18         }
19     }
20 }
```

Beispiel 1



```
1 public class MyClass {  
2     public static void main(String[] args){  
3         int a = 0;  
4         int b = 5;  
5         System.out.println((a > 0) && (b / a > 1));  
6     }  
7 }
```

Beispiel 1 – Das sieht der Compiler

```
1 public class MyClass {  
2     public static void main(String[] args){  
3         int a = int;  
4         int b = int;  
5         System.out.println((int > int) && (int / int > int));  
6     }  
7 }
```

Beispiel 1



```
1 public class MyClass {  
2     public static void main(String[] args){  
3         int a = 0;  
4         int b = 5;  
5         System.out.println((a > 0) && (b / a > 1));  
6     }  
7 }
```

A red bracket is drawn under the expression `(a > 0)` in line 5, with the word **False** written in red text below it.

False

Beispiel 1 – Ohne Short-Circuiting



```
1 public class MyClass {  
2     public static void main(String[] args){  
3         int a = 0;  
4         int b = 5;  
5         System.out.println((b != 0) && (b / a > 1));  
6     }  
7 }
```

True Exception

Aufgabe 1



```
1 int x = 10
2 double y = 5.0;
3 int z = x + y;
```

Compile-Fehler



Exception



Aufgabe 2



```
1 int x = 10;  
2 int y = 5;  
3 double result = (double x + y;
```

Compile-Fehler



Exception



Aufgabe 3



```
1 String num = "123";  
2 int number = (int) num;  
3 double result = 2 * number;
```

Compile-Fehler



Exception



Aufgabe 4



```
1 double d = 10.9;  
2 int x = (int) d;  
3 System.out.println("x is: " + x);
```

Compile-Fehler

☐

Exception

☐

Weder, noch!

Aufgabe 5



```
1 int x = 0;
2 int y = 5;
3 Integer res;
4
5 for (int i = x; i < y; i++) {
6     res = res / (x - y);
7     y--;
8 }
```

Compile-Fehler



Exception



Aufgabe 6



```
1 double x = 10.5;  
2 int y = (int) (x * "2");  
3 int z = x
```

Compile-Fehler



Exception



Aufgabe 7



```
1 Integer a = null;  
2 int b = 5;  
3 int result = a + b;
```

Compile-Fehler



Exception



Aufgabe 8



```
1 int a = 5;  
2 double b = 2.0;  
3 double result = a / b + a * (b - 1);
```

Compile-Fehler

☐

Exception

☐

Weder, noch!

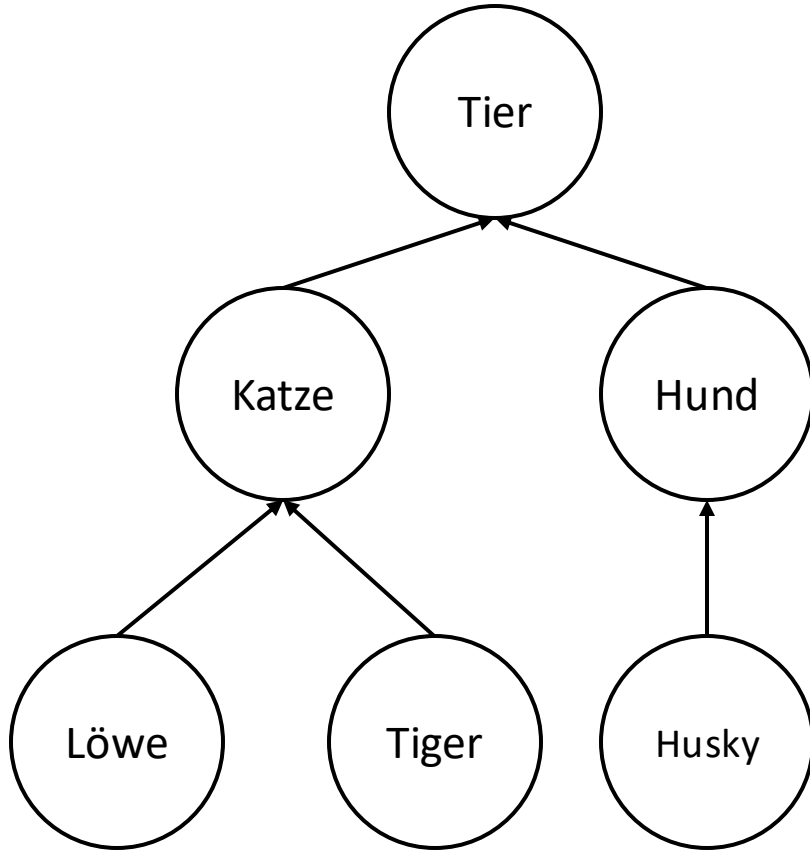
instanceof

konkretesObjekt instanceof Klasse

konkretesObjekt instanceof Klasse

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

instanceof



true oder false?

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

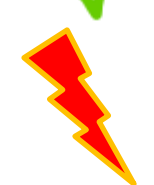
```
hd instanceof Husky
```

```
hd instanceof Tier
```

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

```
h2 instanceof Katze
```

Statischer Typ
Dynamischer Typ



Rename-Funktion in IntelliJ

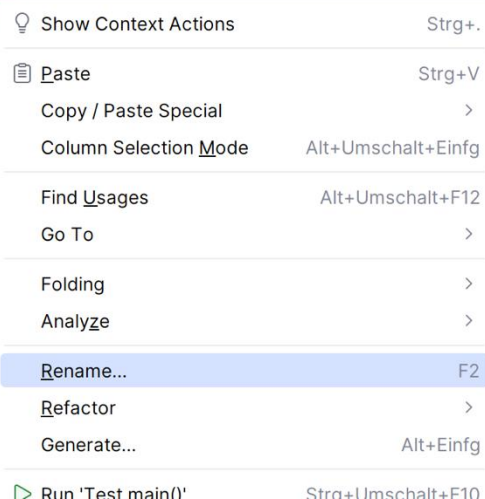
Was kann Rename?

- Die Rename-Funktion in IntelliJ erlaubt es uns Klassen, Methode, Attribute, etc. umzubenennen.
- IntelliJ kann dabei auch nach Verwendungen des Namens suchen. Werden solche Verwendungen gefunden, können die Änderungen, die Sie z.B. am Methodennamen vornehmen, auch auf diese angewendet werden.

Wie nutze ich «Rename»?

```
public static void main(String[] args) {  
    beispielMethode();  
}
```

```
public static void beispielMethode()  
{  
    System.out.println("Hallo!");  
}
```



- Rechtsklick auf den Namen der Methode (oder Klasse)
- „Rename“ auswählen
- Den Namen verändern
- Mit Enter bestätigen
- Danach sind auch alle Aufrufe der Methode geändert

Vorbesprechung

Aufgabe 1: Cyclic List

Bisher haben Sie einfach verkettete Listen gesehen. Zusätzlich wurde ein `IntList`-Interface eingeführt (siehe Anhang), welches die Methoden der Liste abstrahiert.

- a) In dieser Aufgabe üben Sie den Umgang mit Interfaces. Die Klasse `LinkedList` hat alle Methoden, welche vom Interface `IntList` gefordert werden. Implementieren Sie dann eine Methode `ListUtil.addMin(IntList x)`, welche der Liste `x` die kleinste Zahl anhängt, welche in `x` enthalten ist. Implementieren Sie zuletzt die Methode `ListUtil.addMinImpl(LinkedList x)`, welche ebenfalls der Liste `x` die kleinste Zahl anhängt, welche in `x` enthalten ist, aber dafür die Methode `ListUtil.addMin` verwenden soll. Sie dürfen für beide Methoden annehmen, dass die übergebene Liste mindestens ein Element enthält.
- b) In dieser Aufgabe implementieren Sie eine neue Variante einer Liste, die zyklische Liste, welche ebenfalls das `IntList`-Interface implementiert. Zyklische Listen sind ähnlich zu einfach verketteten Listen mit dem Unterschied, dass das `next`-Feld der letzten Node der Liste, falls es einen letzten Knoten gibt, auf den ersten Node der Liste zeigt. Die Knoten der Liste bilden also einen Zyklus. Zusätzlich hat die Liste kein Feld für den ersten Knoten der Liste, da dies unnötig ist. Das Feld `last`, das auf den letzten Knoten zeigt, ist nach wie vor vorhanden. Abbildung 1 zeigt eine solche zyklische Listen mit den Elementen 1, 3, 3, 7. Implementieren Sie die zyklische Liste in der Datei `"CircularLinkedList.java"`. Einige Tests für die Liste finden Sie in `IntListTest`.

Aufgabe 1: Cyclic List

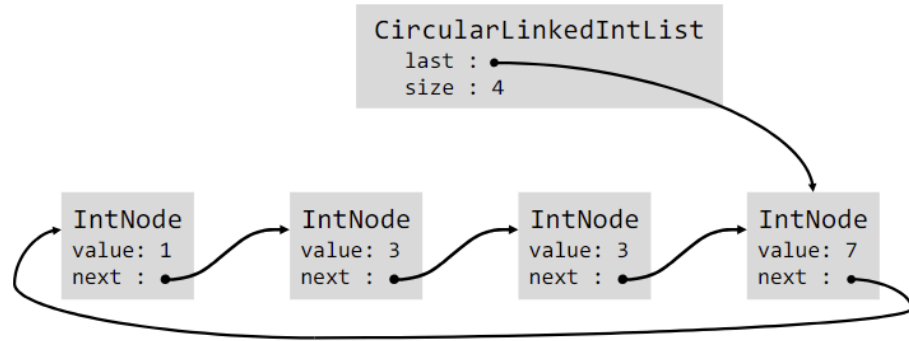


Abbildung 1: Zyklische Liste mit Werten 1, 3, 3, 7.

Aufgabe 2: 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` enthalten 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 2: 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 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   $\leftarrow$  0 | 1 | ... | 9
char    $\leftarrow$  A | B | ... | Z | a | b | ... | z
num     $\leftarrow$  digit { digit } [ . digit { digit } ]
var      $\leftarrow$  char { char }
func     $\leftarrow$  char { char } (
op       $\leftarrow$  + | - | * | / | ^
open     $\leftarrow$  (
close    $\leftarrow$  )

atom     $\leftarrow$  num | var
term     $\leftarrow$  open expr close | func expr close | atom
expr     $\leftarrow$  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 ist¹. Das nennt man *parsen* und ein solches Programm heisst *Parser*. Aus einer EBNF-Beschreibung wie dieser kann man einfach einen Parser erstellen²:

Aufgabe 3: Expression Evaluator

```
/* checks if the next tokens form a valid term */
void parseTerm(...) {
    if(next token is a "open") {
        consume "open" token
        // check if the next tokens are a valid expr:
        parseExpr(...);
        check whether next token is a "close" & consume
    }
    else if(next token is a "func") {
        consume "func" token
        // check if the next tokens are a valid expr:
        parseExpr(...);
        check whether next token is a "close" & consume
    }
    else {
        // check if the tokens are a valid atom:
        parseAtom(...);
    }
}
```

Abbildung 3: Parser-Methode für *term*

```
/* evaluate the next tokens as a term */
double evalTerm(...) {
    if(next token is a "open") {
        consume "open" token
        double val = evalExpr(...);
        check whether next token is a "close" & consume
        return val;
    }
    else if(next token is a "func") {
        consume "func" token
        double arg = evalExpr(...);
        check whether next token is a "close" & consume
        double result = apply function to arg
        return result;
    }
    else {
        return evalAtom(...);
    }
}
```

Abbildung 4: Evaluator-Methode für *term*

- Regeln werden zu Methoden.
- Alternativen werden zu if-Anweisungen.
- Regeln auf der RHS werden zu Methodenaufrufen.

Aufgabe 3: Expression Evaluator

Man unterscheidet dabei zwischen zwei Arten von Regeln: *Parser-Regeln* und *Tokenizer-Regeln*. Zuerst teilt ein *Tokenizer* die Zeichenkette aufgrund der Tokenizer-Regeln in eine Reihe von Tokens auf. In unserer EBNF-Beschreibung sind die Tokenizer-Regeln rot dargestellt. Die grauen Regeln werden zwar intern vom Tokenizer verwendet, aber erzeugen keine eigenen Tokens. Zum Beispiel erzeugt die Zeichenkette "sin(1 + x) * 3.14" die folgende Reihe von Tokens:

```
func : sin(  num : 1  op : +  var : x  close : )  op : *  num : 3.14
```

Danach entscheidet der Parser aufgrund der Parser-Regeln (oben in Schwarz dargestellt), ob eine solche Reihe von Tokens einen gültigen Ausdruck darstellt. Abbildung 3 zeigt, wie die Parser-Methode für *term* aussehen könnte.

- a) In der Übungsvorlage finden Sie eine Tokenizer-Implementation, eine Vorlage für den ExprParser und eine EvaluatorApp mit einer main()-Methode. Diese parst die vom Benutzer eingegebenen Zeichenketten und gibt an, ob sie gültige Ausdrücke sind. Wenn der Benutzer "exit" eingibt, terminiert das Programm. Ihre Aufgabe ist es, den ExprParser zu schreiben.

Erstellen Sie in der schon vorgegebenen parse(String)-Methode eine Tokenizer-Instanz. Die Methoden des Tokenizers sind denen der Scanner-Klasse nachempfunden. Sie können also die hasNext*()-Methoden verwenden, um zu prüfen, welche Art von Token als nächstes kommt, und die next*()-Methoden, um Tokens zu "konsumieren". Schreiben Sie die nötigen parse*()-Methoden, eine für jede Parser-Regel. Die erste Ihrer parse*()-Methoden rufen Sie von parse(String) aus auf. Diese Methoden sollen eine EvaluationException mit einer sinnvollen Fehlermeldung werfen, falls die Zeichenkette kein gültiger Ausdruck ist. Falls z.B. nach "(" und einer expr das Token "10" statt ")" folgt, könnte die Fehlermeldung lauten:

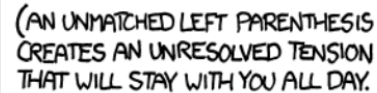
```
Syntax error: unexpected token '10', expected ')'
```

Aufgabe 3: Expression Evaluator

- b) Um aus dem `ExprParser` einen `ExprEvaluator` zu machen, kann man die Methoden so ändern, dass sie im selben Zug das Resultat berechnen. Jede Methode überprüft dann nicht nur, ob die nächsten Tokens der Regel entsprechen, sondern gibt auch gleich den Wert des entsprechenden Ausdruck-Teils zurück. Dies sehen Sie in Abbildung 4.

Benennen Sie die Klasse und die Methoden um³, so dass sie die neue Funktionalität widerspiegeln. Nun können Sie entscheiden: Erstens, welche Funktionen sind erlaubt? Für Aufgabe ?? sollten Sie mindestens `sin()`, `cos()` und `tan()` unterstützen, aber auch andere Funktionen wie `abs()` oder `log()` könnten später Spass machen⁴. Zweitens können Sie entscheiden, wie Sie mit Variablen umgehen. Sie sollten mindestens eine "x"-Variable unterstützen, und wir empfehlen, dass Sie den Wert dafür dem `ExprEvaluator`-Konstruktor übergeben. Sie sollten eine Exception werfen, falls unbekannte Funktionen oder Variablen in einem Ausdruck vorkommen.

Am Schluss sollte die `EvaluatorApp` das Resultat der eingegebenen Ausdrücke ausgegeben, statt nur zu sagen, ob sie gültig sind. Wenn Sie wollen, können Sie dem Benutzer auch die Möglichkeit geben, Werte für Variablen zu definieren.



(AN UNMATCHED LEFT PARENTHESIS
CREATES AN UNRESOLVED TENSION
THAT WILL STAY WITH YOU ALL DAY.

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

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 bestimmen:

- 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.

Aufgabe 4: Contact Tracing

Benachrichtigungen. Da nicht alle Personen gleichermassen gefährdet sind, soll Ihre Applikation die Benachrichtigung der Kontaktpersonen vom Alter, der Art des Kontaktes, sowie dem Testergebnis der jeweiligen Kontaktperson abhängig machen. Dabei soll eine der drei Warnstufen *Keine Benachrichtigung*, *Low-Risk-Benachrichtigung* oder *High-Risk-Benachrichtigung* ausgesprochen werden. Zu Beginn haben alle Personen die Standard-Warnstufe *Keine Benachrichtigung* und gelten als negativ getestet. Davon ausgehend sollen nach jedem registrierten positiven Test die zugehörigen Kontaktpersonen wie folgt benachrichtigen werden:

Testergebnis der Kontaktperson	Alter der Kontaktperson	Direkter Kontakt	Indirekter Kontakt
Positiv	-	Keine Benachr.	Keine Benachr.
Negativ	≤ 60 Jahre alt	High-Risk	Keine Benachr.
Negativ	> 60 Jahre alt	High-Risk	Low-Risk

Eine negativ getestete Person, die höchstens 60 Jahre alt ist und die nur in indirektem Kontakt zu einer positiven Person stand, soll beispielsweise keine Benachrichtigung erhalten (Reihe 2). Eine negativ getestete Person über 60 Jahre hingegen soll als indirekter Kontakt eine Low-Risk-Benachrichtigung erhalten (Reihe 3).

Wenn mehrere Personen positiv getestet werden, soll Ihre Applikation immer die höchste geltende Warnstufe für die anderen, negativ getesteten Personen berechnen. Dabei ist die Ordnung der Warnstufen wie folgt definiert: *Keine Benachrichtigung* < *Low-Risk Benachrichtigung* < *High-Risk Benachrichtigung*. Positiv getestete Personen hingegen sollen immer die Warnstufe *Keine Benachrichtigung* erhalten. Im Allgemeinen dürfen Sie zudem annehmen, dass eine Person, die einmal positiv getestet wurde, für den Rest der Laufzeit Ihrer Applikation als positiv getestet gilt.

Aufgabe 4: Contact Tracing

Implementierung. Erweitern Sie den vorgegebenen Code für die Klasse `ContactTracer` und das Interface `Person` wie folgt, um die Contact-Tracing-Applikation umzusetzen:

Implementieren Sie das Interface `Person` mit den folgenden public Methoden:

- `Person.getUsedIds()`. Diese Methode gibt die Liste aller IDs zurück (`List<Integer>`), die für diese Person als frische ID verwendet wurden, um eine Begegnung zu protokollieren. Nach Hinzufügen einer ID in diese Liste muss dieselbe ID in die jeweilige `Person.getSeenIds()`-Liste des Gegenübers eingetragen sein.
- `Person.getSeenIds()`. Diese Methode gibt die Liste aller IDs zurück (`List<Integer>`), die diese Person als die frische ID des jeweiligen Gegenübers bei einer Begegnung protokolliert hat. Nach Hinzufügen einer ID in diese Liste muss dieselbe ID in die jeweilige `Person.getUsedIds()`-Liste des Gegenübers eingetragen sein.
- `Person.getNotification()`. Diese Methode gibt den aktuellen Benachrichtigungsstatus der Person zurück. Der Rückgabewert soll vom Enum-Typ `NotificationType` sein, welcher vorgegeben ist und die drei möglichen Warnstufen modelliert. `NotificationType` ist im Interface `Person` definiert und enthält die drei Werte `NoNotification` (keine Benachrichtigung), `LowRiskNotification` (Low-Risk-Benachrichtigung) und `HighRiskNotification` (High-Risk-Benachrichtigung).
- `Person.setTestsPositively()`. Diese Methode wird aufgerufen, um eine Person als positiv getestet zu markieren. Nach dem Aufrufen dieser Methode sollen automatisch alle Kontakte von A benachrichtigt worden sein und die entsprechenden Warnstufe per `Person.getNotification()` zurückgeben.

Aufgabe 4: Contact Tracing

Implementieren Sie zusätzlich die Klasse `ContactTracer`, welche die folgenden public Methoden besitzt:

- `ContactTracer.registerEncounter(Person p1, Person p2)`. Mit dieser Methode wird eine (beidseitige) Begegnung zwischen `Person`-Objekten `p1` und `p2` protokolliert, indem die beiden Personen anonyme IDs austauschen. Die ausgetauschten IDs müssen dabei unterschiedlich sein. Eine Begegnung zwischen `p1` und `p2` ist beidseitig und muss somit auch als Begegnung zwischen `p2` und `p1` gewertet werden.
- `ContactTracer.createPerson(int age)`. Diese Methode gibt ein `Person`-Objekt zurück. Das Alter der Person ist durch den `age` Parameter bestimmt.

Alle `Person`-Objekte werden von der Methode `ContactTracer.createPerson(int age)` erstellt. Der `ContactTracer` wird über den parameterfreien Konstruktor `ContactTracer()` instanziiert. Sie dürfen annehmen, dass nie mehr als 1024 Begegnungen zwischen Personen protokolliert werden.

Implementieren Sie auf Basis dieser Vorlage eine Lösung für das Contact-Tracing-Problem. Tests finden Sie in der Datei `"ContactTracerTest.java"`. Die Datei `"ContactTracerGradingTest.java"` enthält die Tests, welche wir bei der Prüfung für die Korrektur verwendet haben. Wir empfehlen, diese Tests erst zu verwenden, wenn Sie denken, dass Ihre Lösung korrekt ist, damit Sie sehen können, wie Sie bei einer Prüfung abgeschnitten hätten.

Nachbesprechung

Aufgabe 1: 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: 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 im Array `arr` von Index 0 bis Index `i` (exklusiv). Alternativ könnte man auch formale Notation benutzen, in dem man mit Quantoren arbeitet. Zum Beispiel, falls `m == min(arr, i)`, dann könnten Sie äquivalent Folgendes schreiben

$$\forall 0 \leq j < i \ (arr[j] \leq m)$$

```
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 2: Loop- Invarianten

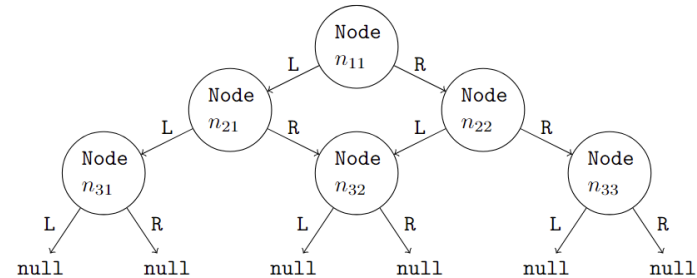
```
2. String append(String str1, String str2) {  
    // Precondition: str1 != null && str2 != null  
    String s1 = str1;  
    String s2 = str2;  
  
    // Loop-Invariante:  
    while (!s2.equals("")) {  
        s1 = s1 + s2.charAt(0);  
        s2 = s2.substring(1);  
    }  
  
    // Postcondition: 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 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 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 Eingabedateien 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:

$$\text{Tarif_}n_l_1_p_1 \dots l_n_p_n$$