Yannick Müller muelleya@student.ethz.ch
yajm.ch

ETH MADE EASY

Computer Science
Basisprüfung - Block 1

January 2019

Introduction to Programming - Lecture

Prof. Thomas R. Gross

Contents

The following was presented in the lecture on 18. September 2018.

1 Einführung in die Programmierung

1.1 Prüfung

24. Januar 2019
ONA7 und ONA25 ab 9:00

1.2 Allgemeines

Laboratory for Software Technology
`https://lst.inf.ethz.ch`
`https://www.video.ethz.ch/lectures/d-infk/2018/autumn/252-0027-00L.html`
Username: gro-18w
Password: Ca3A7Zh

1.3 Warum Programmieren lernen?

- Problem analysieren
- Problem in Teilprobleme zerlegen
- Lösungen finden
- Ergebnisse kombinieren

Zwiespalt:

1. Programmiersprache einfach zu schreiben
2. Programmiersprache einfach zu lesen

EBNF:

- Extended
- Backus
- Naun or Normal
- Form

`https://en.wikipedia.org/wiki/Extended_Backus\OT1\textendashNaur_Form`
Control Forms:

- Sequence
- Decision
- Repetition
- Recursion

EBNF rule
Left Hand Side, Symbol and Right Hand Side

The following was presented in the lecture on 21. September 2018.

2 EBNF

EBNF Beschreibung ist eine Menge von EBNF Regeln
LHS ⇐ RHS
⇐: bedeutet ist definiert als **RHS kann eines der folgenden enthalten:**

- Namen
- Buchstaben (Kann auch eine Zahl sein)
- Kombinationen der vier Kontrollelemente ("control forms")

digit kurisiv gedruckt ist das Gleiche wie <digit>
Wenn etwas krusiv gedruckt ist, ist es definiert als etwas.

2.1 Control forms (zum Kombinieren)

2.1.1 Aufreihung (Squence)

- Aufreihung von links nach rechts

- Reihenfolge ist wichtig
- Muss mit Abstand getrennt werden
- Jeder Name darf nur einmal definiert werden

Beispiel:
buchstabe_1 = *D*
buchstabe_2 = 2
buchstabe_3 = 8
raum = *buchstabe_1* *buchstabe_2* *buchstabe_3*

2.1.2 Auswahl

- Eine Menge von Alternativen (Reihenfolge unwichtig)
- Durch | (eng.: "Stroke") getrennt
- Alternativen folgen den EBNF Regeln

2.1.3 Option

- Element in [und] (eng.: "square bracket")
- Kann gewählt werden, muss aber nicht gewählt werden

2.1.4 Wiederholung (Repetition)

- Der zu wiederholdende Ausdruck steht zwischen { und } (eng.: "curly braces")
- Kann soviel wiederholt werden, wie ich will
- 0 Wiederholungen bedeutet es fehlt und ist auch möglich

2.2 EBNF Beispiel

Definition ganzer Zahlen:
digit $\Leftarrow 0|1|2|3|4|5|6|7|8|9$
integer $\Leftarrow [+|-]$ *digit* $\{$*digit*$\}$

2.3 Überprüfung der LHS

Genaue Übereinstimmung: legal $\Rightarrow$ EBNF Regel erfüllt

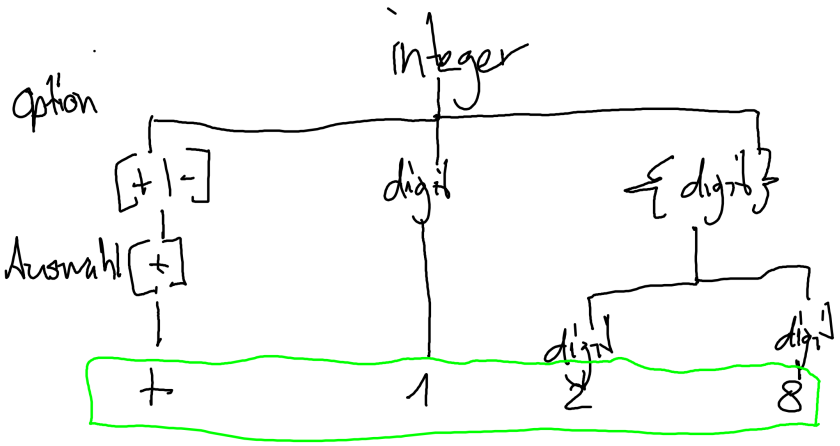
- Buchstaben im Symbol
- Es darf kein Buchstabe im Symbol übrig bleiben

- Es darf kein Element übrig bleiben

Sonst: Symbol nicht legal ⇒ illegal

2.4 Ableitungsbäume

Graphische Darstellung eines Beweises durch eine Tabelle
Oben: Name der EBNF Regel
Unten: Symbol



2.5 Sonderzeichen

Wenn ein Sonderzeichen wie {, }, [,], (,) geschrieben werden muss, wird ein Rahmen herumgezeichnet.

2.6 Äquivalente EBNF Beschreibung

Äquivalent = Gleichwertig
EBNF Beschreibubungen erkennen die selben legalen und illegalen Symbole

2.7 Syntax und Semantik

EBNF beschreibt nur die Syntax

2.8 EBNF Beschreibung *integer_set*

- Mengen von Zahlen
- Anfagn einer Beschreibung, Ende
- Zwischen { und } keine oder mehr Zahlen, durch Komma getrennt

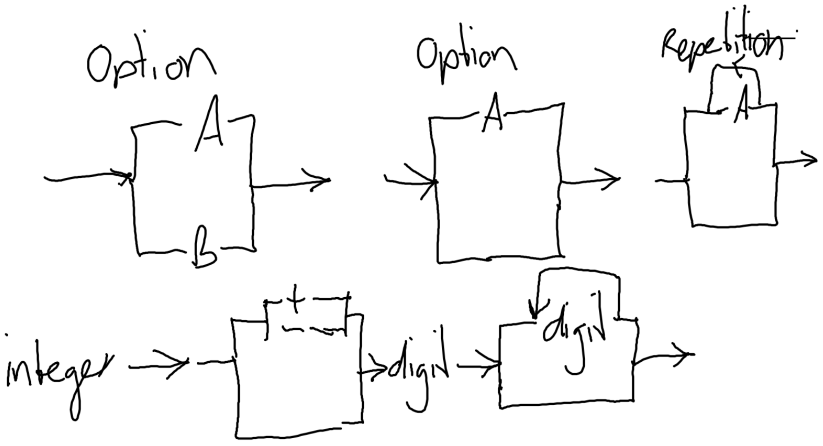
$$integer\_list \Leftarrow integer\{,integer\} \tag{1}$$

$$integer\_set \Leftarrow \boxed{\{}\boxed{[integer\_list]}\boxed{\}} \tag{2}$$

Box bedeutet wir meinen genau dieses Zeichen.

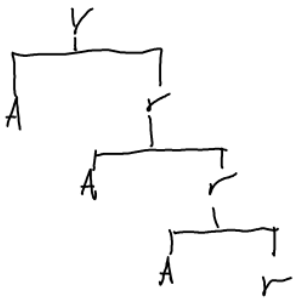
The following was presented in the lecture on 25. September 2018.

2.9 Graphische Darstellung von EBNF Regeln



2.10 Rekursion

Wenn der Name der Links auftrifft auch Rechts auftritt hat man eine rekursive Regel. $r \Leftarrow |Ar$



Das Problem ist, dass nicht jede Rekurison durch Wiederholungen ausgedrückt werden kann. Deshalb wird Rekurison eingeführt. Beispiel: Wenn die Anzahl A gleich von B sein sollte, z.B AAABBB
$balance \Leftarrow |A \quad balance \quad B$

1. Direkte Rekursion:
   $r \Leftarrow A|Ar$

2. Indirekte Rekursion:
   Folge von Regeln $N_1, ..., N_k$ sodass N_2 auf der RHS von N_1, N_3 auf der RHS von R_2, ... und N_1 auf der RHS von N_k erscheint.
   $name1 \Leftarrow A \quad name2$
   $name2 \Leftarrow B \quad name1|C$

3 Einfache Java Programme

3.1 EBNF

Hält die Syntax Regeln von Java Programmen fest

3.1.1 Bezeichner

$bezeichner \Leftarrow letter \quad \{letter|digit\}$

3.2 Einführung

class Name sollte gleich dem .java Namen sein.
Eine Java Methode muss main heissen; das ist der Code den wir ausführen möchten.
Konvention Programmnamen und class beginnt mit grossem Buchstaben.
Konvention method beginnt mit einem Kleinbuchstaben
Java Comments: // Text und /* Text */

The following was presented in the lecture on 28. September 2018.

3.3 Methoden

public static void helper() { } ist ein Beispiel für eine weitere Methode.
Ein guter Programmierer zerlegt ein Programm in verschiedene Methoden und hat so das Problem eigentlich schon gelöst.

3.3.1 static methods

static methods: Methode mit weiteren Eigenschaften
main ist eine static method
main wird automatisch aufgerufen

3.3.2 Wie entstehen Methoden?

Entwickeln des Algorithmus und Auftelung in Teil-Probleme

```
1  public class PrintExample {
2      public static void main(String[] args) {
3          printWarning();
4          System.out.println("Langer Text");
5          printWarning();
6      }
7
8      public static void printWarning() {
9          System.out.println("\n-----------------\n");
10         System.out.println("!!!Warning!!!");
11         System.out.println("\n-----------------\n");
12     }
13 }
```

3.3.3 Ausführung einer Methode

Die Abfolge der Ausführung nennen wir Kontrollfluss (eng.: Control flow)

3.4 Typen und Variablen

Typen (eng.: "types") beschreiben Eigenschaften von Daten
Die Programmiersprache legt fest, wie ein Typ implementiert ist.

3.4.1 Welche Typen kann ein Java Programm verwenden?

Table 1:

Name	Beschreibung	Beispiele
int	ganze Zahlen	42, -3, 0, 926394
double	reelle Zahlen	3.1, -0.25, 9.4e3
char	(einzelne) Buchstaben	'a', 'X', '?', '\n'
boolean	logische Werte	true, false

3.4.2 Ausdrücke (Expressions)

Expression: Ein Wert oder Operanden und Operator, die einen Wert berechnen

$wert \Leftarrow 0|1|2|3|4|5|6|7|8|9$
$operator \Leftarrow +|-|*|/|\%$
$expr \Leftarrow wert \mid expr\ operator\ expr$

Während der Ausführung eines Programms werden die Ausdrücke ausgewertet (eng.: evaluated)

3.4.3 Reele Zahlen (Typ double)

The following was presented in the lecture on 02. October 2018.

3.5 String und Operationen

In Java gibt es automatische Konvertierung wo man zum Beispiel String und Integeres zusammenfügen kann.

3.6 for loop syntax

```
for (initialization; test; update) {
    statement;
}
```

- Initialisierung wird einmal am Anfang ausgeführt und bestimmt, welche Variable für den Foor-Loop verwendet wird.
- Test
- Update

x=1
y=x++: x wird zuerst verwendet und erst dann ausgeführt. Deshalb ist y auch gleich 1.

3.7 Input und System.in

Scanner: erlaubt es Input von unterschiedlichen Quellen zu lesen. Scanner sind in der Bibliothek java.util definiert.

```
import java.util.*;
Scanner console = new Scanner(System.in): //Statt console kann man auch einen anderen
    Namen nehmen.
int alter = console.nextInt();
```

The following was presented in the lecture on 05. October 2018.

3.8 Verschachtelte for-Schleife

Auf Englisch nennt man es Nested Loop.

3.9 Übergabe von Werten

Auf Englisch nennt man es "Value semantics".
Die Werte werden an die Funktion Übergeben und diese behandelt die Werte als neue Werte ausser Sie wurden als Static definiert.

3.10 if-else Anweisung

Um if-else Schleifen zu vereinfachen können boolesche Ausdrücke verwendet werden.

3.11 Boolesche Ausdrúcke

&& is And
|| is Or
! is not
Boolesche Operationen haben sehr kleine Präsedenz.

3.12 Typ boolean

Boolesche Werte können in Boolean abgespeicheret werden. Diese können den Wert true or false annehmen.

3.13 Bedingte short-circuit Auswertung

Java beendet die Auswertung eines booleschen Ausdrucks sobald das Ergebnis feststeht.
Umbedingt Aufpassen. Es kann ungewollte Nebenwirkungen haben, wenn zum Beispiel Variablen noch verändert werden in einer if-Schleife, jedoch der zweite Teil nicht immer ausgeführt wird.
De Morgan's Regeln, das sind Reglen für die Negation von booleschen Ausdrücken und können zum Teil solche Nebenwirkungen verhindern.

The following was presented in the lecture on 09. October 2018.

3.14 Gartenzaun Analogie

Füge eine Anweisung ausserhalb der Schleife hinzu.

3.15 while-Schleifen

while-Schleife: Führe den Schleifenrumpf so lange aus wie der boolesche Ausdruck den Wert true ergibt.

3.16 do/while-Schleife

Führe das do Zuerst aus und mache erst dann den while check.

```
do {
# Statement
} while (# check)
```

3.17 Ergebnis Rückgabe

```
public static type name(paramters) {
statments;
...
return expression;
}
```

type void bedeutet, dass es kein Rückgabewert gibt.
Bei einer void Methode kann man auch ein return schreiben, einfach ohne dass ein Wert zurückgegeben wird.

3.18 Scope (Sichtbarkeitsbereich)

scope: Der Teil eines Programms in dem eine Variable sichtbar ist. Eine Variable die in einer Methode definiert wurde, existiert nur in der Mathode. Definiert in for-Schleife exisitiert nur in for-Schleife.

3.19 String, Math und Random

Ein String ist KEIN array. String methoden:

- indexOf
- length
- substring
- toLowerCase
- toUpperCase

Diese Methoden ändern den String nicht. Man muss einen neuen String zuweisen.

The following was presented in the lecture on 12. Octobre 2018.

3.20 Math Klasse

Enthählt viele Mathematik Methoden.
Die meisten geben double Werte zurück.

3.21 Random Klasse

```
Random rand = new Random();
int randomNumber = rand.nextInt(10);
```

3.22 Arrays

Ein Array erlaubt uns, mehrere Werte des selben Typs zu speichern.

```
type[] name = new type[length];
```

Arrays class hat eine equals Methode mit der man zwei Arrays vergleichen kann.
Sie enthählt auch eine toString Methode für die Ausgabe von Arrays.

The following was presented in the lecture on 16. October 2018.

3.23 Graphische Benutzeroberflächen (GUIs)

```
int width=500, height = 300;
Window window = new Window("First GUI", width, height);
window.open();
window.fillCircle(30,30,20);
for (int i = 0; i < width;i++0 {
    double x = 0.05 * i;
    double y = Math.sin(x);
    window.fillRect(i, y*height/4 + height/2,1,1);
    window.refresh(10);
}
window.waitUntilClosed();
```

3.24 Input / Output Dateien

```
import java.io.*;
import java.util.*;
public static void main ([String[] args) throws FileNotFoundException {
    File file = new File("example.txt"); // Only the handler; does not create a new
        file;
    if (file.exists() && file.length() > 1) {
        Scanner scanner = new Scanner(file);
        int zahl = scanner.nextInt();
```

```
8           PrintStream fileOutput = new PrintStream(file);
9           fileOutput.println(zahl);
10      }
11  }
```

Note 1. Nicht mit dem selben Inupt und Output file arbeiten.

3.25 Javadoc

Kann man benützen um verschiedene Methoden über Klasen und Methoden holen. Sollte man unbedingt vor dem Test anschauen. Da man es verwenden kann während dem Test.

The following was presented in the lecture on 19. October 2018.

4 Klassen und Objekte

Klassen beschreiben einen Typ
Typen beschreiben Eigenschaften von Daten
Ein Typ beschreibt eine Menge oder Kategorie von Daten Werten. Der Begriff Objekt ist der Sammelbegriff für alle Datenwerte, die durch irgend eine Klasse beschireben werden.
Begriff new initialisiert ein neues Objekt.
Strings sind auch Objekte deshalb könnte man auch mit dem new Operator ein neuer String konstruieren:

```
1  String s = new String("Hello World");
```

4.1 Reference Semantics

Eine Variable enthält eine Referenz auf einen Array.
Wenn man aber nun a den Array b zuweist, und nun a[0] verändere ist auch b[0] verändert.
Objekte verwenden auch Reference Semantics.
Wenn man den Array kopieren möchte muss man folgenden Command verwenden.

```
1  int[] src = {1,2,3,4,5};
2  int[] dest = new int[5];
3  System.arraycopy( src, 0, dest, 0, src.length ); // Professor
4  dest = src.clone(); // Eigene Erfahrung
```

4.2 Value Semantics

Die Basistypen verwenden Value Semantics, wo man jede Variable einzeln verändern kann.

4.3 Klassen selber entwickeln

In einer Klasse können wir:

1. Einen namenlosen Service implementieren
2. Eine neue Art von Objekten beschreiben

Klasse ist die Vorlage die Objekte beschreibt.
Ein Objekt ist ein Gebilde das Zustand und Verhalten verbindet (State and behavior)
Objekt-orientieretes Programmieren ist ein Programmiermodel das ein Programm als eine Menge von aufeinander einwirkenden Objekten organisiert.
Objekte sind Exemplare einer Klasse

- Wenn ein Objekt erschaffen wird spricht man von der Instanziierung
- Die Menge aller Objekte einer Klasse bilden einen Typ

4.4 Beispiel Point Objekte

Point Objekte können von anderen Programmen verwendet werden.
Programme die Point Objekte verwenden heissen Klienten (Client Programs) der Point Klasse.

The following was presented in the lecture on 23 .October 2018.

4.5 Attribute

Ein Objekt ist ein Gebilde das Zustand und Verhalten erbindet. Klasse ist die Vorlage die Objekte beschreibt.
Attribute (field): Eine Variable innerhalb eines Objektes die Teil des Objekt Zustandes ist
Ein Exemplar ist ein Neu generiertes Exemplar dieser Klasse.
Die Klasse die ein neues Exemplar erstellt ist ein Klient der Klasse wo das Exemplar programmiert ist.
Mit einer Referenzvariable kann man auf ein Attribut eines Objektes zugreifen.
Für reference variables gelten die reference semantics Regeln.
Wenn man zwei Reference Variablen auf das gleiche Exemplar hat und ein Attribute des Exemplares verändert, ändert sich auch das Attribut für das Exemplar und nicht nur für die reference variable.
Mit “null” kann man eine reference Variablen zurücksetzen.

4.6 Array in Objekten

Initialisierung von Array Objekten:

```
Point[] ziel = new Point[3];
for (int j; j < ziel.length; j++) {
    Point[j]= new Point();
}
```

Initialisierung eines mehrdimensionalen Arrays

```
int [][] x = new int[5][5];
int [][] x = {{1,0,0},{0,1,0},{0,0,1}};
```

The following was presented in the lecture on 26. October 2018.

4.7 Methoden

Methoden in Exemplaren habe keine static Methode. Normale Methoden, die sich in Objekten befinden.

AccessorMethoden Eine Methode die es erlaubt den Zustand eines Objektes anzusehen (bsp. getAdresse();)

MutatorMethoden Eine Methode die den Zustand eines Objektes verändert (bsp. setAdresse();)

Um ein Objekt zu drucken, kann man dafür eine toString Methode im Objekt schreiben.

```
public String toString() {
    return name + Arrays.myarray
}
```

4.8 Konstruktoren

Konstruktor initialisert den Zustand eines neuen Objektes.
Deshalb kann man dann, so Objekte erstellen:

```
Point p = new Point(3, 8); // Initialisierung von irgendwo

public class Point {
    int x;
    int y;

    // Konstruktor
    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }

     // Default Konstruktor
    public Point() {
```

```
15          x = 0;
16          y = 0;
17          // this(0, 0) Besser
18      }
19
20      //Methods
21      public void setLocation(int newX, int newY {
22          // Statements
23      }
24  }
```

Eine Klasse kann mehrere Konstruktoren haben, jedoch muss jeder Konstruktor eine unverwechselbare Liste von Parametern haben. Also unterschiedliche typen.
Wenn wir einen Konstruktor haben, verschwindet der Default Konstruktor. Deshalb ist es empfolen noch einen Default Konstruktor zu kreieren.
Im Konstruktor darf keine Variable deklariert werden.
Konsturktoren sind keine Methoden, deshalb darf man auch nichts returnieren. (Auch kein void)

4.9 Encapsulation

Idee: Verstecken der Details der Implementierung einer Klasse vor den Klienten der Klasse.
Trennung von Verhalten (extern sichtbar) und Zustand (intern).
Encapulation erlaubt Anpassung an Objekten, ohne dass der Klient etwas mitgekriegt und dadurch kaputt geht.

4.9.1 Private Attribute

```
1  private int id;
```

Private Mehoden ond types können nur von der jeweiligen Klasse aufgerufen werden

4.9.2 this Keyword

this.field Expliziter Zugriff auf ein Attribut

this.method() Explizierter Zugriff auf eine Methode

this verweist immer auf den impliziten Parameter einer Methode
this kann auch verwendet werden um einen Konstruktor aufzurufen: this(constructor)

The following was presented in the lecture on 30. October 2018.

4.10 Redundanz in Programmen

Modul: wiederverwendbare Software, in einer Klasse abgelegt.
So wird es aufgerufen:

```
class.method(parameters);
```

4.11 static

Eine Static Methode gehöhrt zur Klasse. Static Methoden sind nicht Teil eines Attributes. Und deshalb exisiteren sie nur einmal, unabhängig wieviele Exempalre erstellt wurden.
Im Englischen wird eine static Variable field genannt.
final bedeutet, dass es nicht mehr verändert werden kann.

The following was presented in the lecture on 02. November 2018.

4.12 Type Case

Eine explizite Umwandlung in einen anderen type wird type cast genannt. Er ist rechtsassoziativ, dass heisst er wandelt nur den Operanden direkt rechts daneben um.

```
(type) expression
double result = (double) 19 / 5;
int result2 = (int) result;
doulbe x = (double) 1 + 1 / 2; // 1.0
double y = 1 + (double) 1 / 2; // 1.5
double y = 1 + 1.0 / 2; // 1.5 The same as above
```

5 Arbeiten mit Objekten und Klassen

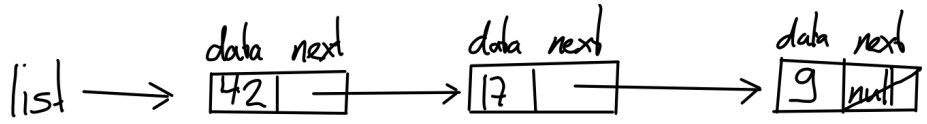
5.1 Kombinationen von Referenzen auf Klassen und Arrays

```
public static void someFct() {
    SomeClass [] ca = new SomeClass[3]

    for (int i=0, i<3; i++ {
    ca[i] = new SomeClass();
    ca[i].one = new OtherClass();
    ca[i].two = new int[5];
    ca[i].three = new SomeClass();
    }

    System.out.println(ca[2].three.two[3]);
}
```

5.2 Rekursive Methoden

1. Basis Fall finden. Wann hört die Rekursion auf? Muss immer gegeben sein.
2. Wie stellen wir sicher, dass wir auf die Basis hinarbeiten.

5.3 Datenstrukturen mit Verknüpfungen



5.3.1 List Node

```
1  class ListNode {
2      int data;
3      ListNode next;
4
5      public ListNode(int data) {
6          this.data = data;
7          this.next = null;
8      }
9
10     public ListNode(int data, ListNode next) {
11         this.data = data;
12         this.next = next;
13     }
14 }
15
16 public class ConstructList {
17     public static void main(String[] args) {
18         ListNode list = new ListNode(0);
19         for (int i=1; i<=10; i++) {
20             list = new ListNode(i,list);
21         }
22
23          ListNode list2 = new ListNode(0);
24          ListNode currentList = list2;
25          ListNode nextList;
26         for (int i=1; i<=10; i++) {
27             nextList = new ListNode(i);
28             currentList.next=nextList;
29             currentList = nextList;
30         }
31         System.out.println(list.data);
32         System.out.println(list2.next.next.next.next.data);
33     }
34 }
```

The following was presented in the exercise on 06. November 2018.

5.4 get-Methode

```
public int get (int index) {
    ListNode current = front;
    for (int i=0; i<index; i++) {
        current = current.next;
    }
    return current.data;
}
```

5.5 add-Methode

```
public void add(int index, int value) {
    if (index==0) {
        front = new ListNode(value, front);
    } else {
        ListNode current = front;
        for (int i=0; i<index-1; i++) {
            current = current.next;
        }
        current.next = new ListNode(value, current.next);
    }
}
```

5.6 Program beenden

```
System.exit(-1);
```

5.7 remove-Methode

```
public void remove (int index) {
    if (index==0) {
        front = front.next
    } else {
        ListNode current = front;
         for (int i=0; i<index-1; i++) {
            current = current.next;
        }
        current.next = current.next.next;
    }
}
```

5.8 addSorted-Methode

```
public void addSorted() {
    if (front==null || value <= front.data) {
```

```
3           front = new ListNode(value, front);
4       } else {
5           ListNode current = front;
6           while (current.next != null && current.next.data < value) {
7               current = current.next;
8           }
9           current.next = new ListNode(value, current.next);
10      }
11  }
```

5.9 Vergleiche Array und Liste

Array: Konstante Zugriffszeit
Liste: Grösse ist flexibel

5.10 Unterschied null und leere Liste

null Liste LinkedIntList list =null;
leere Liste: LinkedIntList list = new LinkedIntList();

The following was presented in the lecture on 09. November 2018.

5.11 Java Programme Visualisieren

`https://cscircles.cemc.uwaterloo.ca/java_visualize/`

The following was presented in the lecture on 13. November 2018. Double Int List

1. Den Wert einer Zahl
2. Verweis auf Vorgänger
3. Verweis auf Nachfolger

5.12 Neue Klasse aus existierenden Klassen

Vererbung (eng.: inheritance) erlaubt uns, eine Klasse als Erweiterung einer anderen Klasse auszudrücken.
Vererbungshierarchie (Inheritance Hierarchy): Eine Menge von Klassen, die durch eine Beziehung verbunden sind, die gemeinsamen Code verwenden.
Vererbung erlaubt es neue Klassen aus existierenden Klassen zu bilden, sodass die neue Klasse die Attribute beziehungsweise das Verhalten der alten Klasse übernimmt.
Eine Klasse kann eine andere erweitern (extend) und Daten und Zustand sowie Verhalten absorbieren.

Die subclass erweitert die superclass.

```
public class subclass extends superclass {
    // First get the Constructor of the superClass
    public subConstructor(int years) {
        super(years); // Called den Construcotr the SuperClass.
    }

    public double getSalary() {
        return super.getSalary() + 5000;
    }

     // Add extended code
    // It is possible to override methods
}
```

Aufpassen bei Konstruktoren. Subclass erben keine Konstruktoren automatisch. Man muss sie explizit aufrufen.

The following was presented in the lecture on 16. November 2018.

5.13 Details offen halten

In der Superclass muss eine Methode nicht endgültig festigelegt werden - eine Subclass kann dann später diese Methode überschreiben.
Selbe Signatur jedoch kann das Verhalten mit override überschrieben werden.

5.14 Protected Attribute

Auf dieses Attribut kann nur innerhalb der Klasse und ihrer Subclasses zugegriffen werden.

5.15 Array mit verschiedenen Referenzvariablen

```
Angestellte [] array = new SuperClass [5];
array[0] = new FristSubclass();
array[1] = new SecondSubclass();
array[2] = new ThirdSubclass();
```

5.16 Klasse Object

```
Object [] array = new Object [5];
array[0] = new Point(5, 3);
array[1] = "Hello World";
array[2] = new Person();
```

```
array[3] = new Point(5, 3);
int len = array[1].length(); // Throws an error
if (array[0] == array[3]) {} // Is always false
```

5.17 Casts

```
public boolean equals(Object o) {
    if (o instanceof Point) {
         Point point = (Point) o; // Type Cast
         return x == point.x && y == point.y;
    } else {
        return false;
    }
}
```

5.18 @Override

Ist sinnvoll um Fehler zu vermeiden und nicht einfach neue Methoden zu erstellen. Falls man etwas falsch implementiert hat.

The following was presented in the lecture on 20. November 2018.

5.19 Polymorphismus

Ein Programm ,welches viele Formen und Gestalten annehmen kann. Es sollte so entwickelt werden, sodass es für unterschiedliche Objekttypen verwendet werden kann und sein Verhalten seinem Typen anpasst.
Gerebte Klasse werden Kursiv geschrieben oder unterstrichen. Man kann Vereberungen auch mit einer Tabelle darstellen, mit allen Methoden und Klassen.
Wenn eine Methode nur in einer Unterklasse definiert ist, kann man mittels cast darauf zugreifen.

```
((Subclass) var).subclassmethod();
```

Casts ändert nicht die Darstellung oder das Verhalten eines Objektes. sondern nur die Menge der Methoden, die aufgerufen werden können.
Verwanldung geht nur nach unten und oben in der Inheritance Hierarchie.

6 Interfaces

Interfeaces geben uns eine ist-ein Beziehung ohne gemeinsamen Code und Zustand aber mit gemeinsamen Verhalten (Methoden) (Seit Java 8.0 gemeinsamen Code erlaubt)

```java
public interface Shape {
    public double area();
    public double perimeter();
}
```

Keine Aussagen über Attribute mögich.

The following was presented in the lecture on 23. November 2018.

6.1 Interface Repetition

Eine Gruppe von Methoden die eine Klasse implementieren kann. Keine Aussagen über Attribute mögich.

```java
public interface Vehicle {
    public void start();
    public void move(double speed);
    public void stop();
}
```

Die Interface Deklaration enthählt abstrakte Methoden: Methoden Header ohne Implementation

```java
public class Circle implements Shape {
    private double radius;

    pubic double area() {
        return Math.PI * radius * radius
    }

    public double perimeter() {
        return 2*Math.PI * radius
    }

}
```

```java
public class Bisycle implements Vehicle {
    pubic void start() {
       ...
    }

    public void move(double s) {
        ...
    }

    public void stop() {
        ...
    }
}
```

Ein Interface kann ein anderes Interface erweitern.

```
public interface Amphicar implements Vehicle, Boat, Property {
    ...
}
```

Interfaces geben uns eine ist-ein Beziehung ohne gemeinsamen Code und Zustand.
Interfaces nützen nicht die Klasse sondern den Klienten.
Jedes Objekt, das das Interface implementiert, kann als Parameter übergeben werden.
Interfaces sind wichtig um grosse Programme zu strukturieren.

6.2 Overloading

Zwei oder mehrere Methoden mit dem selben Namen aber unterschiedlichen Parameter. Zum Beispiel unterschiedliche Typen oder Reihenfolge.

```
void foo(int i, double j) {}
void foo(double i, int j) {}
foo(1,2) // Throws an error! Because it does not know which function it should call.
```

7 Exceptions

Änderung der Ausführungsreihenfolge auf Grund eines aussergewöhnlichen Ereignisses.
Man muss dabei zwischen zwei unterschiedlichen Exceptions unterscheiden. Wenn das Programm damit umgehen kann und es weiter geht (FileNotFoundException, InputNotValid) oder das Programm muss gestoppt werden, da eine Weiterführung unmöglich ist (bsp. OutOfMemoryError, NullPointerException, NoBatteryLeft).
Mit throws kann man eine exception weitergeben.
Der Handler (try – catch) reicht das Problem nicht mehr weiter nach oben, sondern versucht das Problem zu lösen.

```
Scanner rd==null;

while (rd == null) {
    try {
        rd = new Scanner(new FileReader(input.next()));
    } catch (MyException) {
         System.out.println("File is too big!")
    }  catch (IOException ex) { \\IOException is a SuperType of FileNotFoundException
        System.out.println("Cannot open file!")
    }
}
```

Nur der Erste Catch Block wird ausgeführt, alle anderen werden übersprungen.

```
static Scanner getScanner(String n) throws FileNotFoundException, MyException {
    File f = new File(n);
    if (f.exists() && f.length() > 1000) {
        throw new MyException();
```

```
    }
    return new Scanner(f);
}

class MyException extends Exception {
    // Here can be something but not manditory.
}
```

The following was presented in the lecture on 27. November 2018.

8 Generische Programmierung

8.1 Collection

Ein Objekt das eine Ansammlung von Daten speichert.

8.2 ArrayList

Wie kann man ArrayList aufsetzen, dass wir diese Methoden für alle Arten von Objekten nutzen können?
Daraus folgt der Typ der Elemente sollte ein Parameter sein,
Java erlaubt dies mit generic types.

```
ArrayList<Type> name = new ArrayList<Type>();
```

Den Type kann man nun zwischen den Symbolen angeben.

8.3 Typeparameter

```
class MyType<T> {
    T intern;
    String s;

    public MyType() {}

    public MyType(T,i) {
        intern = i;
        s = i.toString();
    }

    public String toString() {
        return s
    }
}

myType<Point> mine = MyType<Point>(new Point(1,2));
```

8.4 Wrapper Klasse

```
ArrayList<int> list = new ArrayList<int>(); // Throws Error! Needs to be a Wrapper
    Class!
ArrayList<Integer> list = new ArrayList<Integer>();
ArrayList<Double> list = new ArrayList<Double>();
```

Das Umwandeln eines Basistyps in den entsprechenden Wrapper Typ wird als “boxing” bezeichnet. Das zurückwandeln nennt man unboxing.

The following was presented in the lecture on 30. November 2018.

8.5 compareTo

```
String a = "alice";
String b = "bob";
if (b.compareTo(a) > 0) {
    System.out.println("The first is bigger than the second.");
}
```

8.6 Collections

```
ArrayList<String> list1 = new ArrayList<String>();
for (int i = 0; i<s.length; i++) {
    list1.add(new String(s[i]));
}
Collections.sort(list1);
```

8.7 Interface Comparable

```
public interface Comparable<E> {
    public int compareTo(E other);
}
```

Eine Klasse kann das Interface Comparable implementieren und so eine natürliche Ordnung für ihre Exemplare definieren.

```
class OwnClass implements Comparable<OwnClass> {
    String item;
    //Constructors
    public int comareTo(Ownclass myobject) {
        String compareItem = myobject.item;
        return item.compareTo(compareItem);
    }
}
```

8.8 Ternary Operator

```
max = (a>b) ? a : b;
```

The following was presented in the lecture on 04. December 2018.

8.9 Collections

Wichtige Java Service Klasse: Collections.

8.10 Mengen (Sets)

Eine Sammlung eindeutiger und einzigartiger Elemente. (Dublikate sind erlaubt). Folgende Operationen können ausgeführt werden: add, remove, contains.
Set kann in folgenden Klassen implementiert werden: HashSet und TreeSet.

8.10.1 HashSet

Bei einem HashTable wird einem String eine ganze Zahl zugeordnet und beim Index dieser ganzen Zahl eingetragen. Dadurch kann man immer in konstanter Zeit den String finden.

8.11 Tree Set

Basiert auf einem binären Baum und Abrufe können in $\mathcal{O}(\log n)$ gemacht werden.

8.12 Linked Hash Set

Speichert Elemente in der Reihenfolge in der sie in die Menge hinzugefügt werden. Dadruch kann man sie in konstanter Zeit wieder abrufen.

```
List<String> list = new ArrayList<String>();
Set<Integer> set1 = new TreeSet<Integer>(); // Da es ein Set ist koennte man es
    nachtraglich zu einem HashSet aendern => Grosse Flexibilitaet
Set<String> set2 = new HashSet<String>(list);
```

8.13 Operationen in Sets

Union addAll

Intersection retainAll

Difference removeAll

8.14 Ordnungsrelationen

HashSet: Elmenete werden in beliebiger Reihenfolge gespeichert
TreeSet: Aufsteigend (nach compareTo) ArrayList: Werden in der Reihenfolge des Hinzufügens gespeichert.

8.15 For each Schleife

Folgender Code geht durch HashSet und TreeSet in derselben Reihenfolge durch.

```
for (type name: collection) {
    //statements;
}
for (String name: myMap.keySet()) {
    String name = myMap.get(name);
}
```

8.16 Map

Hält eine Menge Schlüssel und eine Sammlung von Werten, wobei jeder Schlüssel mit einem Wert assoziiert ist. (Python: Dictionary)

```
Map<String, String> myMap = new HashMap<String, String>(); // Or TreeMap
myMap.put("ETH", "is cool");
myMap.get("ETH");
myMap.remove("ETH");
```

8.17 KeySet

Die Methode keySet liefert die Menge (Set) aller Keys in der Abbildung.

```
myMap.keySet();
```

8.18 Values

Die Methode values liefert eine Ansammlung aller in der Map auftretenden Werte.

```
myMap.values();
```

8.19 Umkehrfunktion der Abbildung

Es ist erlaubt, eine Map von Mengen, oder eine Liste von Listen oder sonst zu definieren. Muss eine Abbildung von Keys auf eine Menge von values sein.

```
Map<Integer, Set<String>> myMap = new HashMap<Integer, Set<String>>();
myMap.put(3, new TreeSet<String>());
myMap.get(3, add("FirstEntry");
myMap.get(3, add("SecondEntry");
myMap.get(3); // [FirstEntry, SecondEntry]
```

Example 1. WordCount

```
array = [allWords];
Map<String, Integer> wordCount= new HastSet<String, Integer>();
for (int i=0; i<array.length; i++) {
    if (w.contains(array[i] {
        int currentCount = wordCount.get(array[i]) + 1;
        wordCount.put(array[i], currentCount;
    } else {
        wordCount.put(array[i], 1;
    }
}
for (String name: wordCount.keySet()) {
    int count = myMap.get(name);
    if (count >= 1000) {
        System.out.println(name + " " + count);
    }
}
```

The following was presented in the lecture on 07. December 2018.

8.20 Iteration

In einer For each Schleife durch ein Set darf das Set nicht verändert werden. Deshalb braucht man einen Iterator. Ein Objekt das einem Klienten erlaubt, die Elmente einer Ansammlung zu besuchen.

```
Set<Double> scores = new HashSet<Double>();
for (double score : scores) {
    System.out.println(score);
}
```

Ein Iterator ist ein Objekt das einem Klienten erlaubt, die Elemente einer Ansammlung zu besuchen. Deshalb kann man an einer beliebigen Position ein Element ansehen oder entfernen. Sie hat eine hasNext und next Methode.

```
Iterator<Integer> itr = scores.iterator();
while (itr.hasNext()) {
    int score = itr.next();
    System.out.println(score);
```

```
5       if (score(<10) {
6           itr.remove();
7       }
8   }
9   System.out.println(scores);
```

Oder mit einem keySet

```
1   Iterator<String> itr = scores.keySet().iterator();
```

9 Abstrakte Datentypen

Spezifikation einer Ansammlung von Daten und der Operationen, die mit diesen Daten ausgeführt werden können. Wir wissen nicht wie der Datentyp implementiert ist, brauchen das jedoch auch nicht zu wissen.
Es ist eine gute Idee die Variablen für Ansammlungen als Variable des ADT Interface Typs zu deklarieren.

9.1 Stack

Eine Ansammlung zu der Elemente hinzugefügt werden können und aus der Elemente entfernt werden können. LIFO: Last in first out.

The following was presented in the lecture on 11. December 2018.

10 Systematisches Programmieren

Um rationale Zahlen darzustellen ist es wichtig die Zahlen richtig darzustellen, dafür ist der gcd hilfreich. Mit einem Zahlenpaar kann man rationale Zahlen genau darstellen. Ein Kommentar kann klären wer für die korrekte Implementierung verantwortlich ist.

10.1 Hoare Logik

Ein Ansatz wie man über ein Programm logische Schlüsse ziehen kann.

1. Vorwärts und rückwärts schliessen
2. Genauere Definition von Aussagen, Vor- und Nachbedingungen

10.1.1 Vorwärts schliessen

- Bestimmt was sich aus den ursprünglichen Annahmen herleiten lässt.

- Sehr praktisch wenn eine Invariante gelten soll.
- Simuliert die Ausführung des Programms
- Leicht zu verstehen, jedoch führt es dazu dass viele Details festgehalten werden müssen.

10.1.2 Rückwärts schliessen

- Bestimmt hinreichende Bedingungen die ein gegebenes Ergebnis garantieren.
- Wenn das Ergebnis erwünscht ist, dann folgt aus den Bedingungen die Korrektheit.
- Ist das Ergebnis unerwünscht, dann reichen die Bedingungen um einen Bug zu generieren.
- Oft von grossem praktischem Nutzen. Man muss verstehen was jede Anweisung zum Erreichen eines bestimmten Zustands beiträgt.
- Es gibt eine neue Sicht auf ein Programm

10.2 Terminologie

Die Annahme, die vor der Ausführung eines Statements gilt, ist die Precondition.
Die Aussage, die nach der Ausführung gilt, ist die Postcondition

10.2.1 If-Statements

Die Precondition für den then-Block und den else-Blocks (eines If-Statements) beinhaltet das Ergebnis des Tests
Die Postcondition nach dem If-Statement ist die Disjunktion der Postconditions des then und -else Blocks

10.3 Notation

Statt die Pre/Postconditions in Kommentaren verwendet man geschweifte Klammern. Dazwischen steht eine logische Aussage.

The following was presented in the lecture on 14. December 2018.

10.4 Hoare Triple

Eine Hoare Triple besteht aus zwei Aussagen und einem Programmsegment

$$\{P\} \quad S \quad \{Q\} \tag{3}$$

$$\text{Precondition Programmsegment Postcondition} \tag{4}$$

Wenn P gilt und man dann S ausführt, dann gilt danach Q.
Andernfalls ist das Hoare Triple ungültig.

10.5 Assert

Eine Aussage in Java kann durch eine assert Statement ausgedrückt werden, Wenn zur Laufzeit expression nicht gültig ist, dann wird ein Laufzeitfehler generiert.

```
assert expression;
```

10.6 Hoare Triple - Zuweisung

Eine Variable wird durch eine andere Variable ersetzt. Q' ist gültig genau dann wenn $P \Rightarrow Q'$

10.7 Hoare Triple - Folgen von Anweisungen

Triple ist gültig wenn es eine Aussage R gibt sodass

1. $\{P\}S1\{R\}$ ist gültig und
2. $\{R\}S2\{Q\}$ ist gültig

10.8 Hoare Triple - If-Statements

Triple ist gültig wenn es eine Aussages Q1, Q2 gibt sodass

1. $\{P \wedge b\}S1\{Q1\}$ ist gültig und
2. $\{P \wedge !b\}S2\{Q2\}$ ist gültig und
3. $Q1 \vee Q2 \Rightarrow Q$

Beispiel

Example 2.

```
{x>0}
if(a>10) {y=2x;}
else {y=a*x;}
{y>0}
```

$\{(x > 0) \wedge (a < 10)\}$
y=2x
$\{y \geq 2\}$

$\{(x > 0) \wedge !(a < 10)\}$

```
y=ax
```
$\{y \geq 10\}$

$(y \geq 2) \vee (y \geq 10) \Rightarrow y > 0$

10.9 Aussagen über Zustände

Wenn gilt $P1 \Rightarrow P2$ dann gilt $P1$ ist stärker und $P2$ ist schwächer als $P1$
Beim Rückwärtsschliessen sollte man immer die schwächst mögliche Precondition suchen. Da man so auch auf die Stärkeren schliessen kann.
Ohne Schleifen und Methoden gibt es für jedes Programmsegement S und jede Postcondition Q eine eindeutige schwächste Precondition. Abgekürzt wp(S,Q)
Die schwächste Precondition ist true.
Wenn wir einer Variable einen Wert zuweisen dann müssen wir eventuell die Variablennamen ändern, sonst kann es Komplikationen geben.
Eine swap Methode kann nützlich sein um Werte zu vertauschen.

The following was presented in the lecture on 18. December 2018.

10.10 Vorbedingung

Es interessiert uns immer die schwächste Vorbedingung. Das macht es leichter die Vorbedingung der Vorbedingung zu zeigen.

10.11 Loop

Example 3. Man muss eine Invariante suchen.

```
// assume: x >=x
y=0; i=0;
// invariant: y = sum(1,i)
while(i != x) {
    // y = sum(1,i) and i != x
    i++;
    // y = sum(1,i-1)
    y+=i;
    // y = sum(1,i-1)+i
}
// i = x and y = sum(1,i)
// assert: y = sum(1,x)
```

Um Aussagen über die Ausführung des Loops zu machen brauchen wir eine Invariante.
Invariante und der Schleifen Test müssen stark genug sein um zu zeigen, dass die Postcondition des Rumpfs auch die Invariante impliziert.
Invariante und der Schleifen Test müssen stark genug sein um die Postcondition der Schleife zu zeigen.

10.12 Hoare Logik - Invariante

Ein Hoare Triple is gültig wenn es eine Invariante I gibt so dass:

1. Invariante gilt zu Beginn $P \Rightarrow I$
2. Nach Ausführen des Rumpfes gilt die Invariante wieder $\{I \wedge B\}S\{I\}$
3. Invariante impliziert Postcondition. $(I \wedge !B) \Rightarrow Q$

Invarianten dürfen weder zu stark noch zu schwach sein.

10.13 Methedologie um Invarianten zu finden

1. Bestimme zuerst die Invariante und lasse die anderen Schritte leiten.
   Was bringt uns jede Iteration näher an das Ziel?
   Was muss nach jeder Iteration gelten?
2. Schreibe einen Rumpf der die Invariante gültig macht
3. Bestimme den Loop Test so, dass Test-ist-false die Postcondition impliziert
4. Schreibe die Initialisierung so, dass dieser Code die Invariante sicher stellt.

10.14 Chiara Problem

Man hat rote, weisse und blaue Kugeln, die in einem Array sortiert werden müssen.
Es ist besser den beliebigen Teil in der Mitte zu haben. Links hat man die roten Kugeln und rechts die Blauen.

The following was presented in the lecture on 21. December 2018.

10.15 Abstract

Bedeutet dass es nur ein abstraktes Interface ist.
Muss von allen subclass implementiert werden.
Dadurch kann man Interfaces implementieren, ohne alle Methoden zu deklarieren, muss aber in subclasses gemacht werden.

10.16 Inner Classes

Innere Klasse ist eine Klasse die innerhalb einer anderen Klasse definiert sind.

Introduction to Programming - Exercise

Prof. Thomas R. Gross
TA: Mickey

Contents

Getting help from Mickey

- What am I trying to do?
- What am I doing?
- What am I expecting to happen?
- What is happening?

- Line number
- Any output/logs

The following was presented in the exercise on 25. September 2018.

1 Introduction

1.1 Organisation

`git-scm.com/book/en/v2`

1.2 Preview

`blog.osteele.com/2008/05/my-git-workflow/`

1.3 Gerade Zahl - Good exam question

$ger\_digit = \Leftarrow 0|2|4|6|8$
$ung\_digit \Leftarrow 1|3|5|7|9$
$all\_digit \Leftarrow ger\_digit|ung\_digit$
$gerade\_zahl \Leftarrow [+|-] \quad \{all\_digit\} \quad ger\_digit$

1.3.1 With recursion

$digit \Leftarrow \quad |all\_digit\ digit$
$gerade\_zahl \Leftarrow [+|-] \quad digit\ ger\_digit$

1.4 EBNF - Graphical

1. Ableitungsbaum
2. Syntax
3. Pathway

1.5 Vereinfachung der EBNF Regel

Simplify: $[A[A[A]]] \Longrightarrow [A][A][A]$

The following was presented in the exercise on 09. October 2018.

1.6 Ternary Operator

```
int i = input > MAX ? MAX : input;
```

1.7 Verzahnungen - Ugly Code

```
import java.io.PrintStream;
import java.util.Scanner;

public class Verzahnung {

    public static void main(String[] args) {
        PrintStream output = new PrintStream(System.out);
        Scanner scanner = new Scanner(System.in);
        String s = scanner.nextLine();
        String t = scanner.nextLine();
        verzahnungen(s, t, output);

        output.close();
        scanner.close();
    }

    static StringBuffer __ = new StringBuffer("");

    public static void verzahnungen(String __, String _, PrintStream $) {
        if (_(__, 0xdeadbeef, _, $) & _(_, 0xcafebabe, __, $))
            $.println(__);
    }

    static boolean _(String $, int O, String _, PrintStream $$) {
        if ((O = $.length()) != 0) {
            __.append($.charAt(0xf % 0b1111));
            verzahnungen($.substring(1, O), _, $$);
            __.setLength(__.length() - 1);
        }
        return O == 0;
    }
}
```

The following was presented in the exercise on 23. October 2018.

1.8 Test Cases

Always write good test cases.

The following was presented in the exercise on 06. November 2018.

public Überall Sichtbar

private Nur in der Klasse und Methoden sichtbar

no modifiers Also from Subclass sichtbar

1.9 Rechner

```
import java.util.Scanner;

public class Rechner {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        parse(scanner);
    }

    public static Expr parse(String input) {
        return parse(new Scanner(input));
    }

    public static Expr parse(Scanner input) {
        if (input.hasNext()) {
            String next = input.next();
            switch (next) {
            case "+":
                return new Addition(parse(input), parse(input));
            case "*":
                return new Multiplication(parse(input), parse(input));
            case "-":
                return new Subtraction(parse(input));
            default:
                return new Value(Integer.parseInt(next));
            }
        }
        return new Value(0);
    }

}

class Expression implements Expr {
    Expr[] kids = new Expression[2];

    @Override
    public Expr[] children() {
        return kids;
    }

    @Override
    public int eval() {
        return 0;
    }

    @Override
    public String toString() {
        return "" + this.eval();
    }

```

```
51 }
52
53 class Addition extends Expression implements BinOp {
54
55     Addition(Expr expr, Expr expr2) {
56         kids[0] = expr;
57         kids[1] = expr2;
58     }
59
60     @Override
61     public int eval() {
62         return kids[0].eval() + kids[1].eval();
63     }
64
65     @Override
66     public Expr left() {
67         return kids[0];
68     }
69
70     @Override
71     public Expr right() {
72         return kids[1];
73     }
74
75 }
76
77 class Multiplication extends Addition implements BinOp {
78     Multiplication(Expr expr, Expr expr2) {
79         super(expr, expr2);
80
81     }
82
83     @Override
84     public int eval() {
85         return kids[0].eval() * kids[1].eval();
86     }
87 }
88
89 class Subtraction extends Expression implements UnaryOp {
90     public Subtraction(Expr parse) {
91         kids[0] = parse;
92     }
93
94     @Override
95     public Expr[] children() {
96         return new Expr[] { kids[0] };
97     }
98
99     @Override
100    public int eval() {
101        return -operand().eval();
102    }
103
104    @Override
105    public Expr operand() {
106        return kids[0];
107    }
108
```

```
109 }
110
111 class Value extends Expression implements Constant {
112     int data;
113
114     public Value(int parseInt) {
115         this.data = parseInt;
116     }
117
118     @Override
119     public int eval() {
120         return val();
121     }
122
123     @Override
124     public int val() {
125         return data;
126     }
127
128     @Override
129     public Expr[] children() {
130         return new Expression[0];
131     }
132 }
```

The following was presented in the exercise on 04. December 2018.

2 Objects

An object is always dynamic.

2.1 Bienen

```
1  import java.io.File;
2  import java.io.FileNotFoundException;
3  import java.io.PrintStream;
4  import java.util.HashMap;
5  import java.util.Map;
6  import java.util.Scanner;
7
8  public class Bienen {
9
10     public static void main(String[] args) throws FileNotFoundException {
11         String dateiName = "bienen.txt";
12         Scanner scanner = new Scanner(new File(dateiName));
13         PrintStream output = new PrintStream(System.out);
14
15         analyze(scanner, output);
16
17         output.close();
18         scanner.close();
```

```
19      }
20
21      public static void analyze(Scanner input, PrintStream output) {
22
23          Scanner item;
24          Person nextPerson;
25          Person maxIncome = new Person ("", "", 0, 0);
26          Person maxPart = new Person ("", "", 1, 0);
27          String maxCountry = "x";
28          Map<String, Integer> countryCount = new HashMap<String, Integer>();
29          countryCount.put(maxCountry, 0);
30
31          while(input.hasNextLine()) {
32              item = new Scanner(input.nextLine());
33              nextPerson = new Person(item.next(), item.next(), item.nextInt(),
                    item.nextInt());
34              if (nextPerson.income > maxIncome.income) {
35                  maxIncome = nextPerson;
36              }
37              if (1.0*nextPerson.special/(nextPerson.income+nextPerson.special) >
                    1.0*maxPart.special/(maxPart.income+maxPart.special)) {
38                  maxPart = nextPerson;
39              }
40              if (countryCount.get(nextPerson.country) == null) {
41                  countryCount.put(nextPerson.country, nextPerson.special);
42              } else {
43                  countryCount.put(nextPerson.country,
                        countryCount.get(nextPerson.country)+nextPerson.special);
44              }
45              if (countryCount.get(nextPerson.country) > countryCount.get(maxCountry)) {
46                  maxCountry = nextPerson.country;
47              }
48          }
49
50          output.println(maxIncome.name+" "+(maxIncome.income+maxIncome.special));
51          output.println(maxPart.name+"
                "+Math.round(100.0*maxPart.special/(maxPart.income+maxPart.special)));
52          output.println(maxCountry+" "+countryCount.get(maxCountry));
53      }
54  }
55
56  class Person {
57      String name;
58      String country;
59      int income;
60      int special;
61
62      public Person(String name, String country, int income, int special) {
63          super();
64          this.name = name;
65          this.country = country;
66          this.income = income;
67          this.special = special;
68      }
69  }
```

The following was presented in the exercise on 11. December 2018.

3 Compiler

CONST Pushes constant value c onto stack

LOAD Loads value of variable v and pushes it onto stack

STORE Pops a value from stack and stores it in variable v

OP Pops two values r and l, computes $l \oplus r$ and pushes the result back onto the stack

FUNC Pops a value x from the stack, computes $f(x)$ and pushes the result back onto the stack.

The following was presented in the exercise on 18. December 2018.

4 Hoare Logic

$\{P\}S\{Q\}$

S Program Code

P Precondition to the program S

Q Postcondition to the program S

4.1 Invariant

1. Invariant holds at the start $P \Rightarrow I$
2. Nach Ausführen des Rumpfes gilt die Invariante wieder $\{I \wedge B\}S\{I\}$
3. Invariante impliziert Postcondition. $(I \wedge !B) \Rightarrow Q$

4.2 Palindrome

```
1  import java.util.List;
2  import java.util.ArrayList;
3  import java.util.Arrays;
4  import java.util.HashSet;
5  import java.util.Set;
6
7  public class Palindrome {
8      public static void main(String[] args) {
9          List<Integer> list = Arrays.asList(0, 1, 2, 3, 3, 2, 1, 0);
10         Set<List<Integer>> set = maxPalindrome(list);
```

```
11          System.out.println(set.toString());
12      }
13
14      public static Set<List<Integer>> maxPalindrome(List<Integer> input) {
15          checkForError(input);
16          Set<List<Integer>> set = new HashSet<List<Integer>>();
17          int n = input.size();
18          for (int i = n; i > 0; i--) {
19              for (int j = 0; j + i <= n; j++) {
20                  List<Integer> list = new ArrayList<Integer>();
21                  for (int q = j; q < j + i; q++) {
22                      list.add(input.get(q));
23                  }
24                  if (checkIfPalindrome(list)) {
25                      set.add(list);
26                      i=0;
27                  }
28              }
29          }
30          return set;
31      }
32
33      private static void checkForError(List<Integer> input) {
34          boolean error=false;
35          if (input==null)
36              error=true;
37          else if (input.isEmpty())
38              error=true;
39          else {
40              for (Integer i:input) {
41                  if (i==null)
42                      error=true;
43              }
44          }
45          if (error) {
46              throw new RuntimeException("invalid list");
47          }
48      }
49
50      private static boolean checkIfPalindrome(List<Integer> list) {
51          int n = list.size();
52          for (int i = 0; i < n / 2; i++) {
53              if (!list.get(i).equals(list.get(n - 1 - i)))
54                  return false;
55          }
56          return true;
57      }
58  }
```