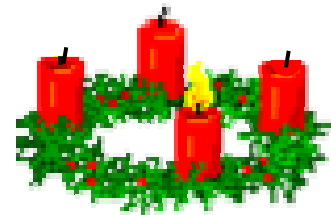


# **Praktische Informatik 1**

## **Variablen, Referenzen, Zuweisungen**


Thomas Röfer

- Variablen
- Zuweisung / Initialisierung
- Konstanten
- Lebenszeit / Sichtbarkeit
- Java Stack / Heap
- Call by Reference / Value
- Musterlösung, Übungsblatt



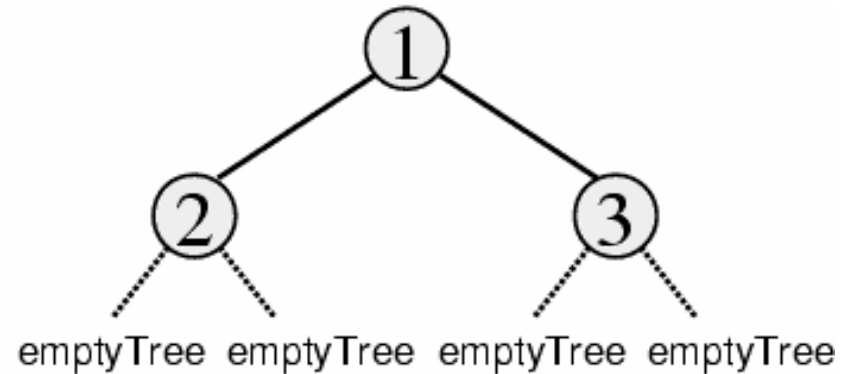
# Java-Stack (Stapel, Keller)

```
class Foo {  
    static int main(int j) {  
        int i = 1;  
        int k = fun1(i, j);  
        return k;  
    }  
  
    static int fun1(int a, int b) {  
        boolean x;  
        return a * b;  
    }  
}
```



## Beispiel: BinTree

```
class BinTree {  
    BinTree left;  
    int val;  
    BinTree right;  
    boolean isEmpty;  
  
    BinTree() {isEmpty = true;}  
  
    BinTree(BinTree l, int v, BinTree r) {  
        left = l;  
        val = v;  
        right = r;  
        isEmpty = false;  
    }  
}
```



```
BinTree b = new BinTree(  
    new BinTree(new BinTree(), 2, new BinTree()),  
    1,  
    new BinTree(new BinTree(), 3, new BinTree())  
);
```

# Java-Heap (Halde)

Stack

M		
boolean	b	true
int	i	42
String	s	
BinTree	t	
M		

Heap

"Hallo"		
BinTree	left	null
int	val	0
BinTree	right	null
boolean	isEmpty	true
BinTree	left	
int	val	17
BinTree	right	null
boolean	isEmpty	false
M		

- `boolean b = true;`
- `int i = 42;`
- `String s = "Hallo";`
- `BinTree t = new BinTree(new BinTree(), 17, null);`

# Java-Heap – Arrays

Stack

M		
int[]	a	
int[][]	b	
M		

Heap

int	length	3
17	42	15
int	length	3
		null
int	length	6
29	0	0
0	0	0
int	length	2
1	24	
M		

- `int[] a = {17, 42, 15};`
- `int[][] b = new int[3][];`
- `b[0] = new int[6];`
- `b[1] = new int[] {1, 24};`
- `b[0][0] = 29;`

# Stack und Heap – Zuweisungen

```
class Value {
    int value;
    Value(int v) {
        value = v;
    }
}
```

Stack

M		
int	i	<del>42</del> 17
int	j	17
Value	v	
Value	w	
M		

- int i = 42, j = 17;
- Value v = new Value(42);
- Value w = new Value(17);
- i = j;
- v = w;
- w.value = 19;

Heap

int	value	42
int	value	<del>17</del> 19
M		

# Stack und Heap – Funktionsaufrufe

- `int i = 42;`
- `Value v = new Value(17)`
- `fun(i, v);`

```
void fun(int j, Value w) {
    w.value = j;
    j = 63;
}
```

Stack

M		
int	i	42
Value	v	
int	j	<del>42</del> 63
Value	w	
Rücksprungadresse		

Heap

int	value	<del>17</del> 42
M		

# Call by Reference / Call by Value

- Call by Value

- Parameter werden als Werte auf dem Stack abgelegt
- Änderungen an ihrem Wert wirken sich nicht auf den Aufrufer aus
- Übergebene Parameter sind Rechtswerte

- Call by Reference

- Referenzen auf die Parameter werden auf dem Stack abgelegt
- Die aufgerufene Funktion arbeitet daher mit den Originalen der Variablen und kann diese ändern
- Übergebene Parameter müssen Linkswerte sein



# Call by Reference / Call by Value

## Beispiel Pascal

- Call by Value
  - Lokale a und b werden vertauscht
  - Keine Wirkung auf Aufrufer
- Call by Reference
  - Übergebene a und b werden vertauscht
  - Wirkung auf Aufrufer

```
PROCEDURE swap(a, b : INTEGER)
  VAR temp : INTEGER;
BEGIN
  temp := a; a := b; b := temp
END
```

```
PROCEDURE swap(VAR a, b : INTEGER)
  VAR temp : INTEGER;
BEGIN
  temp := a; a := b; b := temp
END
```

# Call by Reference / Call by Value

## In Java

- Die Parameterübergabe ist identisch mit Zuweisung, daher gibt es nur „Call by Value“
- Aber
  - Eine Funktion kann Referenzen auf Objekte bzw. Reihungen als Parameter erhalten
  - Diese Referenzen selbst können zwar nicht mit Wirkung auf den Aufrufer verändert werden, wohl aber die Objekte und Reihungen, auf die sie zeigen

## Beispiel: Zahlen austauschen

- `int c = 17;`  
`int d = 42;`  
`swap1(c, d);`
- `Value e = new Value(17);`  
`Value f = new Value(42);`  
`swap2(e, f);`
- `swap3(e, f);`

```
void swap1(int a, int b) {  
    int t = a;  
    a = b;  
    b = t;  
}
```

```
void swap2(Value a, Value b) {  
    Value t = a;  
    a = b;  
    b = t;  
}
```

```
void swap3(Value a, Value b) {  
    int t = a.value;  
    a.value = b.value;  
    b.value = t;  
}
```

# Beispiel: Zahlen austauschen

Stack

M		
int	a	17
int	b	42
M		

swap1

Stack

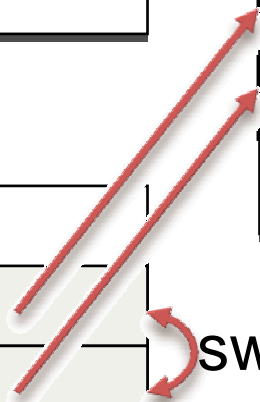
M		
Value	a	
Value	b	
M		

swap2

Heap

int	value	17
int	value	42
M		

swap3



# Musterlösung zu Übungsblatt 4

- Aufgabe 1
  - Aggregation vs. Komposition
  - *has-a*- und *is-a*-Relationen werden vererbt!
- Aufgabe 2
  - Immer noch
    - `while (z > 0) { /* Die Scheife läuft, solange z > 0 */`
  - statt
    - Dazu wird  $i$   $j$ -mal in die Variable  $r$  hinein multipliziert. Die Anzahl der Schleifendurchläufe wird mithilfe der Variablen  $z$  bestimmt, die von  $j$  auf 0 herunter gezählt wird.
  - Testen?

Praktische Informatik I WS 2007/08

## Übungsblatt 4 Musterlösung

### Aufgabe 1 Objektbeziehungen (30%)

a) Erkläre den Unterschied zwischen *has-a* und *is-a* Beziehungen.

Zwei Objektklassen stehen in einer *has-a*-Beziehung zueinander, falls Objekte der einen Klasse Objekte der anderen Klasse enthalten. Bei der *is-a*-Beziehung stehen zwei Objektklassen in einer Subtypbeziehung zueinander. Das bedeutet, dass eine Klasse (Subtype, Subklasse) alle Eigenschaften der anderen Klasse (Obertyp, Oberklasse) besitzt und darüber hinaus noch weitere.

b) Gebt an, welche Beziehungen (z.B. *has-a* oder *is-a*) zwischen einigen der folgenden Klassen von Objekten bestehen: Büro, Möbel, Tisch, Stuhl, Bein, Schreibtisch, Drehstuhl, Begründet cure Wahl.

*has-a*-Beziehungen:

- Ein Büro „hat“ 0-n Möbel.
- Ein Tisch „hat“ 0-n Beine.
- Ein Stuhl „hat“ 0-n Beine.

*is-a*-Beziehungen:

- Ein Tisch „ist“ ein Möbel.
- Ein Stuhl „ist“ ein Möbel.
- Ein Schreibtisch „ist“ ein Tisch.
- Ein Drehstuhl „ist“ ein Stuhl.

Wenn man davon ausgeht, dass es auch noch Möbel ohne Beine gibt, ist dies die minimale Anzahl von Relationen. Sowohl *is-a*- als auch *has-a*-Beziehungen werden vererbt, z.B. hat auch ein Drehstuhl Beine, weil der Stuhl welche hat, zu dem er in einer *is-a*-Relation steht.

c) Visualisiert sie in einem Klassendiagramm.

Siehe Abb. 1

### Aufgabe 2 Der TutorInnen Leid... (70%)

**Funktion 1.** Marla Mysterys Funktion berechnet die  $j$ -te Potenz der Zahl  $i$ ; also  $i^j$ . Dazu wird  $i$   $j$ -mal in die Variable  $r$  hinein multipliziert. Die Anzahl der Schleifendurchläufe wird mithilfe der Variablen  $z$  bestimmt, die von  $j$  auf 0 herunter gezählt wird. Für  $0^0$  wird auch 1 zurückgeliefert, was zumindest identisch mit  $\text{Math.pow}(0, 0)$  ist. Für negative Exponenten liefert die Funktion stets 1 zurück, was angesichts der Tatsache, dass sich die korrekten Ergebnisse ohnehin nicht als Ganzzahl darstellen lassen, nicht überrascht. Sobald der Darstellungsbereich von  $\text{int}$  verlassen wird, sind die Ergebnisse natürlich nicht mehr richtig:

# Übungsblatt 6

- N-Gramm: Folge von  $n$  Zeichen oder Worten

- $n = 1$

testing	testing	one	two	one	two
---------	---------	-----	-----	-----	-----

- $n = 2$

testing	testing	one	two	one	two
---------	---------	-----	-----	-----	-----

- $n = 3$

testing	testing	one	two	one	two
---------	---------	-----	-----	-----	-----

# Übungsblatt 6

- Implementierung
  - 4 Methoden implementieren
  - b) und c) nutzen a)
  - d) nutzt b) und c)
- Tests
  - Jede Methode einzeln testen
  - Nur Tests innerhalb der Eingabespezifikation
  - Daher: Eingabespezifikation angeben, z.B.
    - $i$  muss im Bereich ... liegen
    - Der Text muss mindestens ... Worte haben

Praktische Informatik I WS 2007/08

## Übungsblatt 6

Abgabe: 12.12.07

### Aufgabe 1 Könnt' ich doch so schreiben wie... (100%)

Ein *N-Gramm* ist eine Folge von  $n$  Zeichen oder Wörtern. Betrachtet man einen Text, kann man ihn als eine Folge von sich überlappenden *N-Grammen* ansehen, d.h. mit jedem Wort beginnt ein neues *N-Gramm* und die letzten  $n - 1$  Worte eines *N-Gramms*, das beim  $i$ -ten Wort beginnt, sind identisch mit den ersten  $n - 1$  Worten des *N-Gramms*, das bei Wort Nummer  $i + 1$  anfängt. Diese Repräsentation macht man sich z.B. bei Spracherkennungssystemen zunutze, wo man die Wahrscheinlichkeiten für das Auftreten bestimmter *N-Gramme* (meistens 3-Gramme, d.h.  $n = 3$ ) aus großen Textmengen ermittelt hat und somit auch die Wahrscheinlichkeiten für das Auftreten bestimmter Sätze kennt. So kann man die Erkennungsleistung deutlich verbessern, ohne den gesprochenen Text inhaltlich verstehen zu müssen.

Im Rahmen dieser Übungsaufgabe, die als *uebung06* im Repository abzulegen ist, wollen wir *N-Gramme* nicht zur Texterkennung einsetzen, sondern zur Textgenerierung. Gegeben sei ein Text, der als Reihung von Wörtern (Zeichenketten) vorliegt. Wir beginnen an einer beliebigen Stelle mit dem Index  $i$  in dem Text. Das Wort mit dem Index  $i$  geben wir aus. Bei  $i$  beginnt auch ein *N-Gramm*. Nun suchen wir alle *N-Gramme* aus dem Text, die identisch zu dem bei  $i$  sind, wählen ein zufälliges mit dem Index  $j$  davon aus und setzen unseren Text bei  $j + 1$  fort, d.h. unser neues  $i$  ist  $j + 1$ . Dies wiederholen wir solange, bis wir genug Text ausgegeben haben. Es ist zu erwarten, dass dieser Ansatz, abhängig von der Länge  $n$  der *N-Gramme*, ganz unterschiedliche Ergebnisse liefert. Daher soll  $n$  ein Eingabeparameter des Algorithmus sein.

Zum Einlesen eines Textes findet ihr auf der PI-1-Seite das Archiv *uebung06.zip*, das die Klasse *WebReader* enthält. Diese definiert eine Klassenmethode, mit der man sich einen Text als Reihung von Wörtern liefern lassen kann. Der Text muss in reinem Textformat vorliegen. Als Quelle bietet sich das Projekt Gutenberg an, z.B. kann man *The Time Machine* von H. G. Wells mit `String[] words = WebReader.read("http://www.gutenberg.org/files/35/35.txt");` herunterladen.

Die Aufgabe lässt sich gliedern, indem man folgendes implementiert:

- Eine Methode, die feststellt, ob bei zwei Indizes  $i$  und  $j$  im Text die gleichen *N-Gramme* beginnen, d.h. die Wörter bei  $i \dots i + n - 1$  und  $j \dots j + n - 1$  paarweise gleich sind.
- Eine Methode, die zählt, wie oft das *N-Gramm* bei Index  $i$  im Text vorkommt.
- Eine Methode, die den Index des  $m$ -ten Vorkommens des *N-Gramms* bei Index  $i$  im Text zurückliefert.
- Eine Methode, die einen zufälligen Text generiert. Dazu werden der Ursprungstext als String-Array, die Länge der *N-Gramme*, der Index des ersten Wortes und die maximale Anzahl der zu generierenden Worte benötigt.

Testet methodenweise.

**Tipps:** Eine Zufallszahl im Bereich  $0 \dots m - 1$  generiert man mit `(int)(Math.random() * m)`. Für Tests bietet es sich an, sich auch eine eigene, kurze Reihung mit Worten zu definieren, anstatt den Text aus dem Web zu holen, weil der Zugriff darauf recht langsam ist und man die Häufigkeiten der *N-Gramme* in dem fremden Text nicht kennt.



# Übungsblatt 6

## Textgenerierung mit 2-Grammen

<u>testing</u>	testing	one	<u>two</u>	<u>one</u>	two
----------------	---------	-----	------------	------------	-----

testing	<u>testing</u>	<u>one</u>	two	<u>three</u>
---------	----------------	------------	-----	--------------

testing	testing	one	two	one	two
---------	---------	-----	-----	-----	-----



## Übungsblatt 6

- Was passiert, wenn ein N-Gramm am Ende des Textes vorkommt
  - aber auch anderswo?
  - und nirgendwo sonst?
- Mögliche Erweiterungen (nicht gefragt)
  - Teile des Ursprungstextes ausschließen
    - Hinweise am Anfang und Ende
  - Zeilenumbruch bei der Ausgabe
- WebReader.read() kann auch lokale Texte lesen
  - "file:///C:/Users/myAccount/Desktop/Text.txt"
  - "file:///home/myAccount/Desktop/Text.txt"