ETH zürich

Informatik II - Übung 05

Katja Wolff
katja.wolff@inf.ethz.ch

Besprechung Übungsblatt 4

L4.A1 – Stack

- Zwei Attribute: buffer und size
- Vergrößern der Kapazität durch `grow()`
  - Benötigt in `push()` wenn `size == buffer.length`
  - Java-Bibliotheksfunktionen möglich:

```
int[] Arrays.copyOf(int[] original, int newLength)
```

L4.A2.a – Ackermann-Funktion

- Rekursive Definition der Ackermann-Funktion:

$$A(0,m) = m+1$$

$$A(n+1,0) = A(n,1)$$

$$A(n+1,m+1) = A(n, A(n+1,m))$$

L4.A2.b – Pseudocode (Beispiel)

```
push n on stack
push m on stack
As long as the stack's size is greater than 1
   pop the uppermost element from stack to m [m]
   pop the uppermost element from stack to n [n]
   if n = 0
      then push m+1 on stack
         [A(0,m)=m+1]

   elseif m = 0
      then push n-1 on stack; push 1 on stack
         [A(n,0)=A(n-1,1)]

   else
         push n-1 on stack
         push n on stack
         push m-1 on stack
         [A(n,m)=A(n-1,A(n,m-1))]
the uppermost element from the stack is the result
```

```
while(stack.size() > 1){
....
```

“Funktionsaufruf”

```
if n == 0 → result = m+1
else if m == 0 → push(n-1), push(1)
else push(n-1), push(n), push(m-1)
```

L4.A3 – Bytecode

```
0:  iload_1
1:  ifne 8
4:  iload_2
5:  iconst_1
6:  iadd
7:  ireturn
8:  iload_2
9:  ifne 21
12: aload_0
13: iload_1
14: iconst_1
15: isub
16: iconst_1
17: invokevirtual #16; //Method A:(II)I
20: ireturn
```

```
21: aload 0
22: iload 1
23: iconst 1
24: isub
25: aload 0
26: iload 1
27: iload 2
28: iconst 1
29: isub
30: invokevirtual #16; //Method A:(II)I
33: invokevirtual #16; //Method A:(II)I
36: ireturn
```

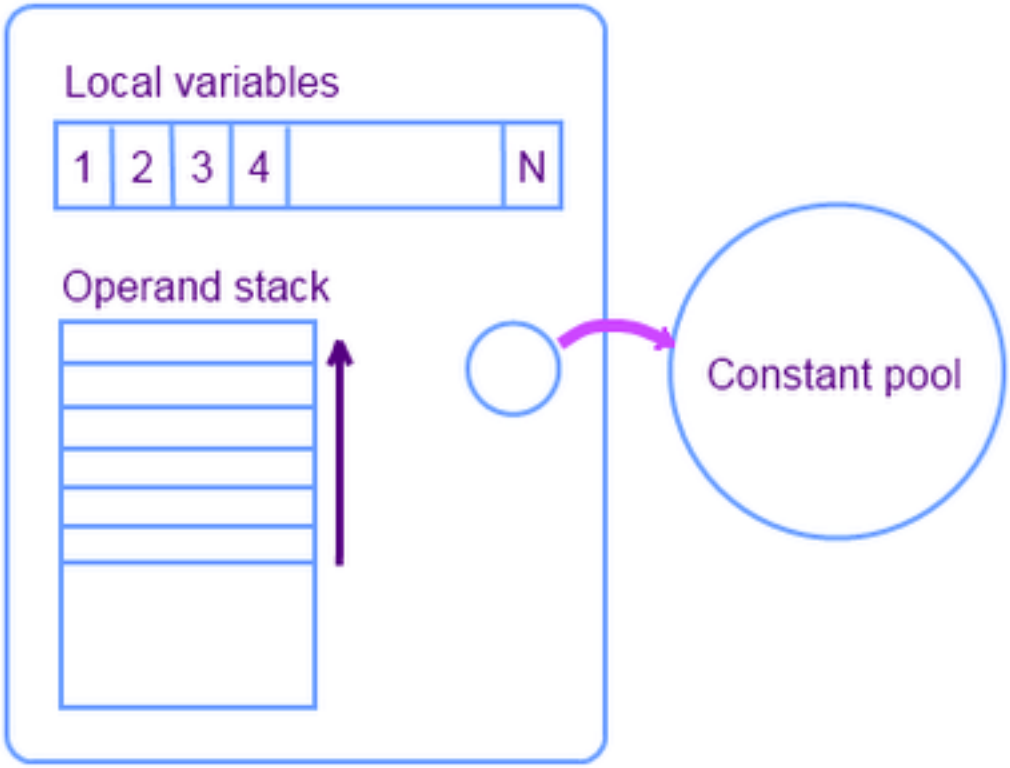
L4.A3 – Bytecode

```
0:  iload_1                                  if(n == 0)
1:  ifne 8
4:  iload_2                                  return m+1
5:  iconst_1
6:  iadd
7:  ireturn
8:  iload_2                                  if(m == 0)
9:  ifne 21
12: aload_0                                  return A(n-1,n)
13: iload_1
14: iconst_1
15: isub
16: iconst_1
17: invokevirtual #16; //Method A:(II)I
20: ireturn
```

```
21: aload 0                                  return A(n-1,A(n,m-1))
22: iload 1
23: iconst 1
24: isub
25: aload 0
26: iload 1
27: iload 2
28: iconst 1
29: isub
30: invokevirtual #16; //Method A:(II)I
33: invokevirtual #16; //Method A:(II)I
36: ireturn
```

Was passiert wirklich beim Funktionsaufruf?

- https://www.artima.com/underthehood/invocationP.html
- http://arhipov.blogspot.ch/2011/01/java-bytecode-fundamentals.html



Call by value
vs.
Call by reference

Call-by-...

- **value**
  - an Funktion übergebenen Daten werden *kopiert*
  - *keine Verbindung* mehr zwischen den Daten beim Aufrufer und den Daten in der Funktion
- **reference**
  - Anstatt Daten zu kopieren werden Referenzen auf die Daten *übergeben*
  - Methodenaufrufe an einem so übergebenen Objekt arbeiten also auf *demselben Objekt*, das auch außerhalb sichtbar ist

Call by value vs. Call by reference

- In C++ ist beides möglich
  - Call by value: Daten werden kopiert und übergeben

```
//C++                 void swap(a, b);
```

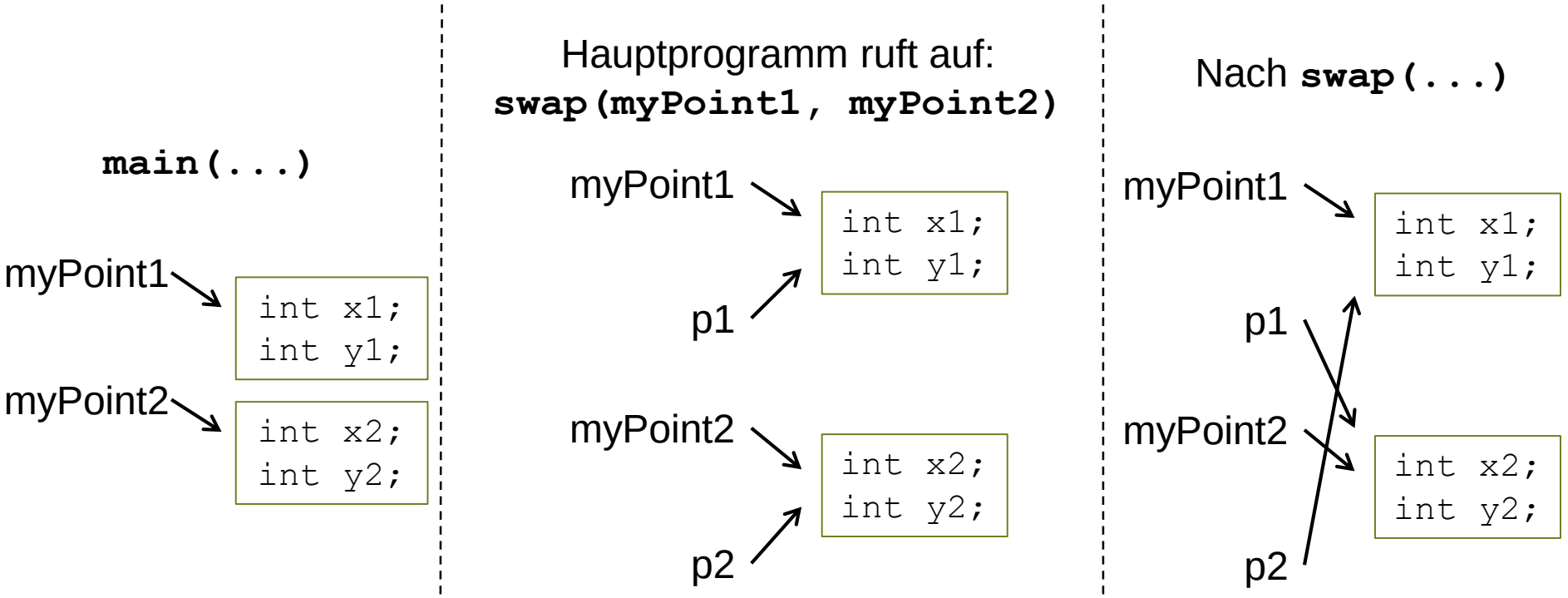
  - Call by reference: Referenz auf die Daten wird übergeben

```
//C++                 void swap(&a, &b);
```

- Java ist IMMER call by value!!
  - Bei der Übergabe einer Referenz auf ein Objekt wird der Adresswert in eine lokale Variable kopiert!
  - Bei Übergabe eines primitiven Typs (`char`, `int`, `float`) wird der Wert in eine lokale Variable kopiert!

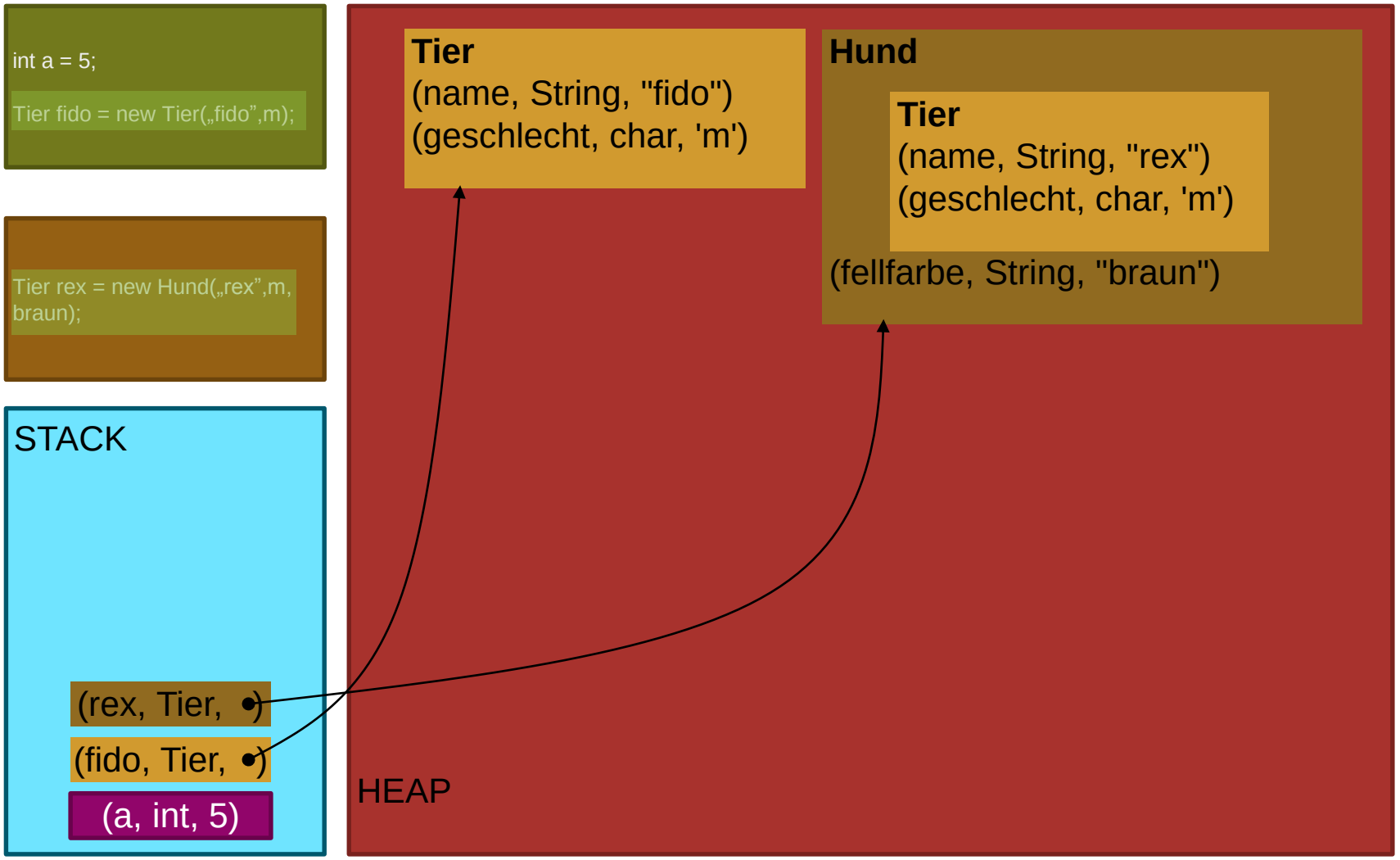
Call by value vs. Call by reference in Java

- Modifizieren von Objekten ist möglich, Vertauschen aber nicht!

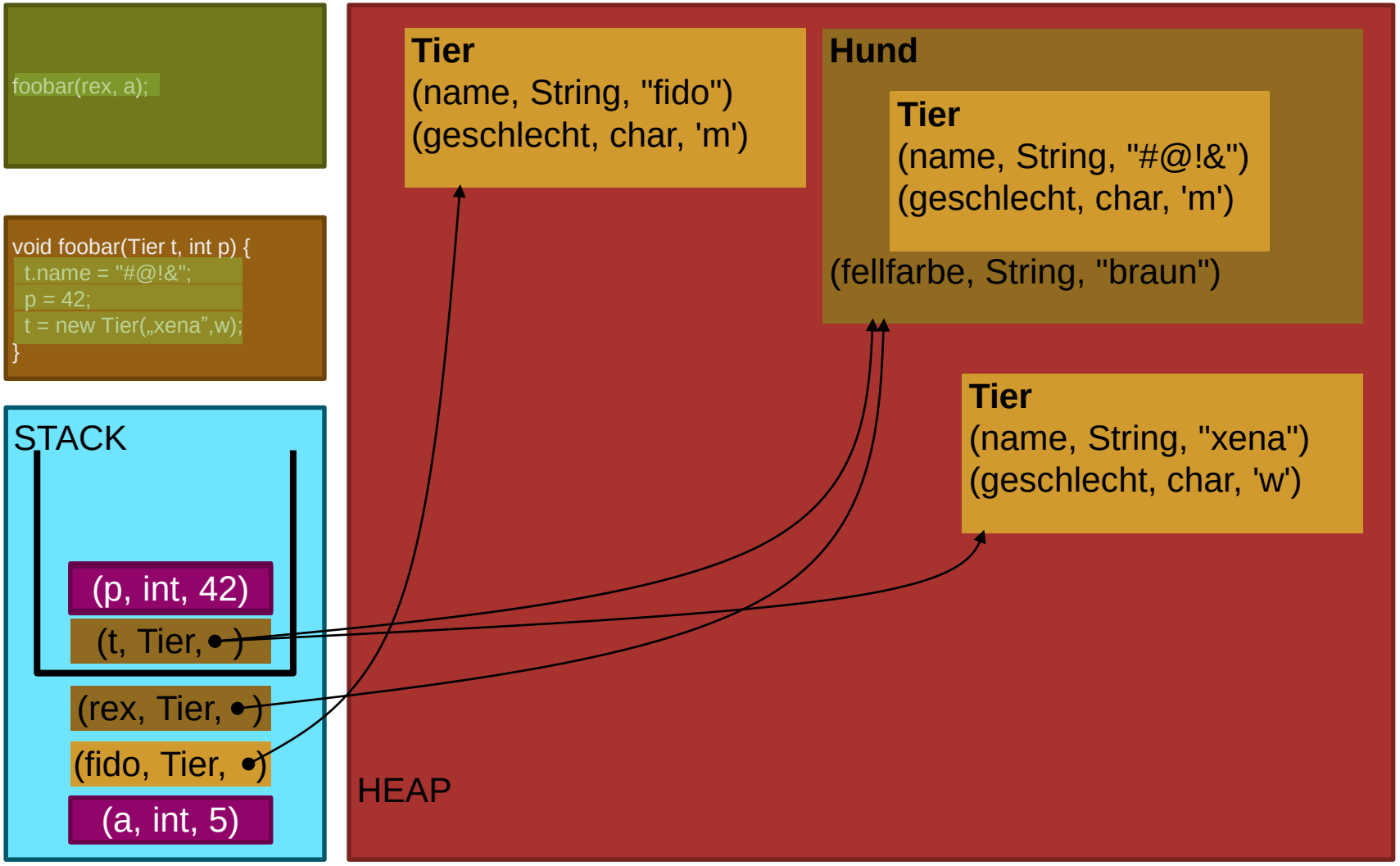


Gute Referenz: *http://www.javaworld.com/javaworld/javaqa/2000-05/03-qa-0526-pass.html*

Allokation primitiver Datentypen und Objekte



Allokation primitiver Datentypen und Objekte



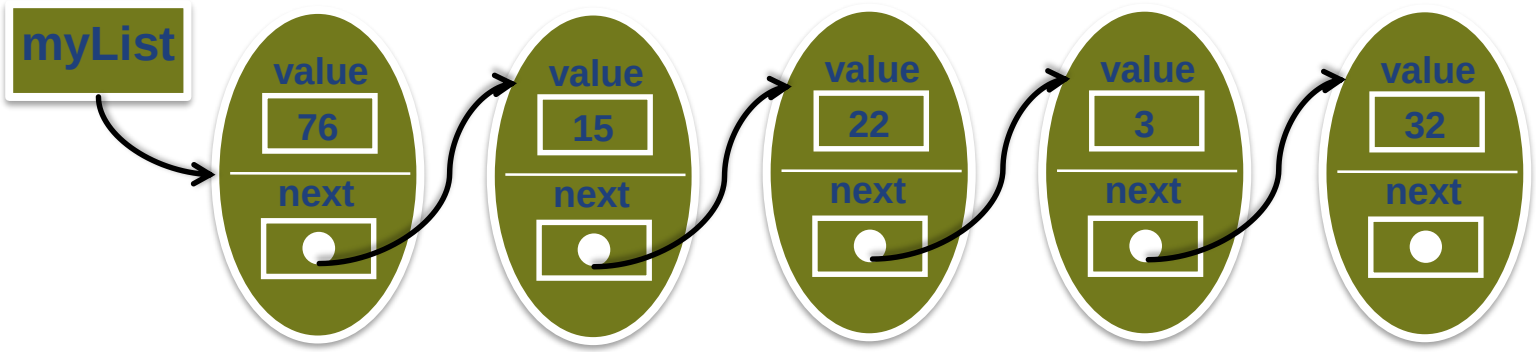
Ausblick
Übungsblatt 5

Übersicht

- Verkettete Listen – Read Only
- Listen modifizieren
- Sortierte Listen
  - Einfügen eines neuen Wertes in eine sortierte Liste
- Dynamischer Stack mit verketteten Listen

Alles per Rekursion!
Keine Schleifen!

Ü5.A1 Beispiel: toString()




```
public static String toString(List list) {
  if (list == null)
    return "null";

  return list.value + "," + toString(list.next);
}
```

u5a1.Lists.toString(myList)

76,15,22,3,34,null

Ü5.A1

- Liste
  - Wert + Referenz auf Nachfolger
  - Nächstes Element ist wieder eine Liste
  - Leere Liste: `null`

- Generell: JavaDoc für exakte Spezifikation lesen

- Nur ReadOnly-Operationen
  - Warum verändert `sublist()` die Liste nicht?
  - Warum verändert `add()` die Liste nicht?

Ü5.A1/A2/A3

Bereits implementiert

```
String toString(List list)
```

Aufgabe 1

```
List add(List list, int value)
int size(List list)
int sum(List list)
List sublist(List list, int index) throws ...
List last(List list)
int valueAt(List list, int index) throws ...
int index(List list, int value) throws ...
```

Aufgabe 2

```
void append(List list, int value) throws ...
void concat(List head, List tail) throws ...
void insertAt(List list, int i, int value) throws ...
void insertAt(List list, int i, List nl) throws ...
List remove(List list, int index) throws ...
```

Aufgabe 3

```
List insertSorted(List list, int value) th...
List sort(List list) throws ...
```

Ü5.A3

- Implementierung von
  - `insertSorted()`
    - Ein Wert in eine sortierte Liste einfügen
  - `sort()`
    - Eine sortierte Liste aus der gegebenen erstellen
    - Tipp: Verwendung von insertSorted()

In 2 Lines of Code möglich

Ü5.A4 – Dynamisch wachsender Stack mit verketteten Listen

- Neuimplementieren der Stack-Methoden mit Hilfe der Liste

- Wo wird beim `push()` das neue Element angehängt?
  - Am Anfang oder am Ende der Liste?
  - Anfang: `add()`, Ende: `append()`

…viel Spass!