Informatik II (D-ITET)

Tutorial 5

TA: Anwar Hithnawi, E-mail: hithnawi@inf.ethz.ch
Distributed Systems Group, ETH Zürich
Outlook

- Exercise 4: Solution discussion
- More Java insights (Call by Value and Call by Reference)
- Exercise 5: Overview (Lists)
Ex4.Q1 – Stack

- **Noteworthy**
  - Two attributes: buffer and size
  - Capacity $\leftrightarrow$ buffer.length (Array indices from 0 to length-1)
  - Empty $\leftrightarrow$ size == 0
  - Size $\leftrightarrow$ index of first free space at the top

```java
void push(int value) {
    ... buffer[size++] = value;
}
```

- **grow**
  - Conditions of Grow in push: size() == capacity()
  - Java-library functions (search and use)
    ```java
    int[] Arrays.copyOf(int[] original, int newLength)
    ```

- **JavaDoc**
  - How is it documented! Was it helpful?
Ex4.Q2 – Ackermann-Function

Recursion definition of the Ackermann Function

\[
A(0, m) = m + 1 \\
A(n + 1, 0) = A(n, 1) \\
A(n + 1, m + 1) = A(n, A(n + 1, m))
\]
Ex4.Q2.b – Pseudocode (Example)

```
while(stack.size() > 1) {
    ...  // Function call
    if n == 0 → result = m+1
    else if m == 0 → push(n-1), push(1)
    else push(n-1), push(n), push(m-1)
}
the uppermost element from the stack is the result
```
package u4a2;
import u4a1.Stack;
public class IterativeAckermann {
    public int A(int n, int m) {
        Stack stack = new Stack(10);
        stack.push(n);
        stack.push(m);
        while(stack.size() != 1) {
            System.out.println(stack);
            final int cm = stack.pop();
            final int cn = stack.pop();
            if (cn == 0) {
                stack.push(cm+1);
            } else if (cm == 0) {
                stack.push(cn - 1);
                stack.push(1);
            } else {
                stack.push(cn-1);
                stack.push(cn);
                stack.push(cm - 1);
            }
        }
        System.out.println(stack);
        return stack.pop();
    }
}
Ex4.Q3 – Bytecode

- SourceCode-Bytecode, assignment clear?
- Order of parameters / return, clear?

```java
/**
 * Recursive implementation of the Ackermann function.
 */
public class RecursiveAckermann {
    /**
     * Recursive implementation of the Ackermann function.
     *
     * @param n parameter n
     * @param m parameter m
     * @return Ackermann(n,m)
     */
    public int A(int n, int m)
    {
        if (n == 0) return m + 1;
        if (m == 0) return A(n-1, 1);
        return A(n-1, A(n, m-1));
    }
}
```

```java
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
31: invokevirtual
32: ireturn
```
Outlook

- Exercise 4: Solution discussion
- More Java insights (Call by Value and Call by Reference)
- Exercise 5: Overview (Lists)
Two Types of variables in Java

- Primitive Variable: holds the value of primitive data type
  - E.g. `byte`, `int`, `float`, `char`

- Reference Variable: holds a reference to an Object
  - E.g. Arrays, Strings, Classes
Call-by-...

- Call by value
  - The method receives a copy of the variable's
  - No connection between the data in the caller and the data in the function
    - Changes affect only the local copy and anything into a function call is unchanged in the caller’s scope when the function returns

- Call by reference
  - Instead of copying the data an implicit reference to a variable is used as an argument, rather than a copy of its value.
  - Method calls on such an object work on the same instance of the object, so that changes are visible as well outside the method
Call by value vs. Call by reference

In C++ both are possible

Call by value: Data is transferred and copied

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

Call by reference: References to the data are passed

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

Java is **ALWAYS** called by value!!

- This means, that when passing a reference type the address value is copied at local variable!
- In case of transferring from a primitive types the value would be copied in local copy
Call by value vs. Call by reference in Java

- What does this mean for us?
- Is the modification of the given values possible?
  - How / Why not?
- Is exchanging the given values possible?
  - How/ Why not?
Call by value vs. Call by reference in Java

- **Modification** is possible, but **Interchanging** is not!

```
main(...)

myPoint1
  int x1;
  int y1;

myPoint2
  int x2;
  int y2;

Main programm is called nevertheless
swap(myPoint1, myPoint2)

myPoint1
  int x1;
  int y1;

  p1

myPoint2
  int x2;
  int y2;

  p2

After swap(...)

myPoint1
  int x1;
  int y1;

  p1

myPoint2
  int x2;
  int y2;

  p2

Allocation of primitives and objects

```java
int a = 5;
Animal fido = new Animal("fido", m);
Animal rex = new Animal("rex", m, brown);
Animal fido = new Animal("fido", String, "fido")
             (sex, char, 'm')
Animal rex = new Animal("rex", String, "rex")
             (sex, char, 'm')
             (furcolor, String, "brown")
```

STACK

- (rex, Animal, )
- (fido, Animal, )
- (a, int, 5)

HEAP
Allocation of primitives and objects

```java
void foobar(Animal t, int p)
{
    t.name = "#@!&";
    p = 42;
    t = new Animal("xena", 'w');
}
```

```java
foobar(rex, a);
```

```java
Animal (name, String, "fido")
(sex, char, 'm')
```

```java
Animal (name, String, "#@!&")
(sex, char, 'm')
```

```java
Dog (furcolor, String, "brown")
```

```java
Animal (name, String, "xena")
(sex, char, 'w')
```

```java
Animal (name, String, "fido")
(sex, char, 'm')
```

```java
Animal (name, String, "#@!&")
(sex, char, 'm')
```

```java
Animal (name, String, "xena")
(sex, char, 'w')
```

```java
t.name = "#@!&";
p = 42;
t = new Animal("xena", 'w');
```
Outlook

- Exercise 4: Solution discussion
- More Java insights (Call by Value and Call by Reference)
- Exercise 5: Overview (Lists)
Ex5 – Linked Lists

- Exercises sheet 5 mainly cover the concept of recursive definition of lists

- Linked Lists
  - Dynamic Data Structure
  - Singly-linked, Doubly-linked,...

- Fields in non empty lists
  - Values + References

  For singly-linked lists:
  - Value + pointer to the next element in the list
  - The next element is in turn a list
  - Fields of the empty lists (null):
    Value: (no access: throw an Exception)
    Next: null
Ex5.Q1/Q2/Q3

- Static, complete recursive methods
  - It should not generate new lists but rather the passed list should be changed
  - Solve it recursively, without using loops!

- Test-Cases to verify the implementation
Ex5.Q1/Q2/Q3 – public static...

String toString(List list) **Already Implemented**

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 ...

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 ...

List insertSorted(List list, int value) throws ...
List sort(List list) throws ...
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
Ex5.Q1

- Implementation of
  - add
    - Add a value to the front of the list
  - size
    - Calculate the length of the list
  - sum
    - Sum the values in the list
  - last
    - End of list (last node before the zero, otherwise the zero)
Ex5.Q1

- Implementation of
  - sublist
    - "Rest list" From a given index
  - valueAt
    - Return the value of a given index in the list
  - index
    - Index of the first node with a given value

- Tipp: Consider Helper functions (code reusability!)
  - E.g. nodeAt
    - Similar usability in sublist and valueAt
    - You use when manipulating the list as well…
  - Must also be recursively implemented!
Ex5.Q2

- Implementation of
  - append
    - Attach a value at the end of list
  - concat
    - Attach a list to the back of another list
  - insertAt
    - Insert an element to list after certain index
  - remove
    - Delete a value in the list at certain position
Ex5.Q3

- Implementation of
  - `insertSorted`
    - Embed a value in a sorted list
  - `sort`
    - A sorted list of a given input
Ex5.Q4 – Stack with list implementation

- Lists (Implementation) is invisible from the outside

- All the stack-operations can be directly translated to operation on lists
  - Empty list: is the only special case, that needs to be respected by the Class Stack!
Have Fun!