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)
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
  ```cpp
  void swap(a, b);
  ```

- **Call by reference**: References to the data are passed
  ```cpp
  void swap(&a, &b);
  ```

Java is **ALWYAS** 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!

```java
main(...) {
    int x1;
    int y1;
    myPoint1

    int x2;
    int y2;
    myPoint2
}

swap(myPoint1, myPoint2)

After swap(...) {
    int x1;
    int y1;
    myPoint1

    int x2;
    int y2;
    myPoint2
}
```

Allocation of primitives and objects

```
int a = 5;
Tier fido = new Tier("fido", m);

Tier rex = new Hund("rex", m, braun);

Tier (name, String, "fido")
    (geschlecht, char, 'm')

Hund (name, String, "rex")
    (fellfarbe, String, "braun")

HEAP

STACK

(a, int, 5)
(fido, Tier, ⬤)
(rex, Tier, ⬤)
```
void foobar(Tier t, int p) {
    t.name = "#@!&";
    p = 42;
    t = new Tier("xena", w);
}

Allocation of primitives and objects

STACK

HEAP

void foobar(Tier t, int p) {
    t.name = "#@!&";
    p = 42;
    t = new Tier("xena", w);
}

Tier
(name, String, "fido")
(geschlecht, char, 'm')

Hund
Tier
(name, String, "#@!&")
(geschlecht, char, 'm')
(fellfarbe, String, "braun")

Tier
(name, String, "xena")
(geschlecht, char, 'w')

Tier
(name, String, "fido")
(geschlecht, char, 'm')

Informatik II - Übung 05
Let us explore the difference with help of Debugger...
Outlook

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

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

- Handled with references ('Pointer')

- Class list.List (recursive definition)
  - Either an empty list or a chain of integer with a list
  - Function definition: List := null | int List
  - here: null is the empty list
  → null is to be expected as parameter!
Ex5 – Linked 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)**

**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 ...**

**Already Implemented**

**Question 1**

**Question 2**

**Question 3**
public static String toString(List list) {
    if (list == null)
        return "null";

    return list.value + "," + toString(list.next);
}
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!
Happy Holidays!