Ex4.Q1 – Stack

- Noteworthy
  - Two attributes: buffer length and size
    - capacity ↔ buffer.length (Array indices from 0 to length-1)
    - empty ↔ size == 0
    - size ↔ 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 copy)
    ```java
    int[] Arrays.copyOf(int[] original, int newLength)
    ```

- JavaDoc
  - How it is documented!
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)

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

- 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
Ex4.Q3 – Bytecode

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

```
return A(n-1, A(n, m-1))
```

```
21: aload 0
22: iload 1
23: icont 1
24: isub
25: aload 0
26: iload 1
27: iload 2
28: icont 1
29: isub
30: invokevirtual
31: invokevirtual
32: invokevirtual
33: invokevirtual
34: invokevirtual
35: invokevirtual
36: ireturn
```
more Java insights…
Data Types

- Primitive Types
  - E.g. `byte`, `int`, `float`, `char`

- Reference-Type
  - 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

- Call by reference
  - Instead of copying the data, you assign a reference to it
  - Method calls of a referenced object work on the same object which is visible from outside.
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 **ALWAYS** call by value!!

- This means, that when passing reference type, the address value is copied a 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 program is called nevertheless:
```
swap(myPoint1, myPoint2)
```

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 Dog("rex", 'm', "brown");
```

Diagram:
- **Stack**:
  - (a, int, 5)
  - (fido, Animal, •)
  - (rex, Animal, •)

- **Heap**:
  - Animal (name, String, "fido")
    (gender, char, 'm')
  - Dog
    Animal (name, String, "rex")
    (gender, char, 'm')
    (color, String, "brown")
void foobar(Animal t, int p) {
    t.name = "#@!&;"
    p = 42;
    t = new Animal("xena", 'w');
}

int a = 5;
Animal fido = new Animal("fido", 'm');

Animal (name, String, "fido")
(gender, char, 'm')
Dog
Animal (name, String, "#@!&")
(gender, char, 'm')
(color, String, "brown")
Animal (name, String, "xena")
(gender, char, 'w')
Hints to Ex05
Ex5 – Lists

- Sheet 5 mainly cover the concept of recursive definition of lists

- Handled with references (‘Pointers’)

- Class list.List (recursive definition)
  - *Either an empty list or a chain of integers*
  - Function definition: $\text{List} := \text{null} \mid \text{int List}$
  - here: null is the empty list
  - $\text{null}$ 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 – Linked Lists

vector

move down to make room

linked list

[Max Fomitchev]
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
### Question 1
- `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 ...

### Question 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 ...

### Question 3
- `List insertSorted(List list, int value)` throws ...
- `List sort(List list)` throws ...
Ex5.Q1 – toString(List list)

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

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

Informatik II - Übung 05
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`
    - "Sublist" 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

- Tip: 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!