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)

```plaintext
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){
  ....

  "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)

```
package u4a2;
import u4a1.Stack;

public class IterativeAckermann {
    // @return Ackermann(n,m)
    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

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

Call by reference: References to the data are passed

```cpp
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;  

myPoint1  
  p1  
  int x1;  
  int y1;  

myPoint2  
  p2  
  int x2;  
  int y2;  

After swap(...)  
myPoint1  
  p1  
  int x1;  
  int y1;  
myPoint2  
  p2  
  int x2;  
  int y2;  
```
Allocation of primitives and objects

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

STACK

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

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

foobar(rex, a);

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

Dog

Animal (name, String, "#@!&")
(sex, char, 'm')
(furcolor, String, "brown")

Animal (name, String, "#@!&")
(sex, char, 'w')

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

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

Animal (name, String, "rex")
(sex, char, 'm')

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

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

Animal (name, String, "rex")
(sex, char, 'm')

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

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

Animal (name, String, "rex")
(sex, char, 'm')

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

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

Animal (name, String, "rex")
(sex, char, 'm')

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

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

Animal (name, String, "rex")
(sex, char, 'm')

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

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

Animal (name, String, "rex")
(sex, char, 'm')

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

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

Animal (name, String, "rex")
(sex, char, 'm')

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

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

Animal (name, String, "rex")
(sex, char, 'm')

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

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

Animal (name, String, "rex")
(sex, char, 'm')

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

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

Animal (name, String, "rex")
(sex, char, 'm')

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

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

Animal (name, String, "rex")
(sex, char, 'm')

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

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

Animal (name, String, "rex")
(sex, char, 'm')
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…

<table>
<thead>
<tr>
<th>String toString(List list)</th>
<th>Already Implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>List add(List list, int value)</td>
<td>Question 1</td>
</tr>
<tr>
<td>int size(List list)</td>
<td></td>
</tr>
<tr>
<td>int sum(List list)</td>
<td></td>
</tr>
<tr>
<td>List sublist(List list, int index) throws ...</td>
<td></td>
</tr>
<tr>
<td>List last(List list)</td>
<td></td>
</tr>
<tr>
<td>int valueAt(List list, int index) throws ...</td>
<td></td>
</tr>
<tr>
<td>int index(List list, int value)</td>
<td></td>
</tr>
<tr>
<td>void append(List list, int value) throws ...</td>
<td></td>
</tr>
<tr>
<td>void concat(List head, List tail) throws ...</td>
<td></td>
</tr>
<tr>
<td>void insertAt(List list, int i, int value) throws ...</td>
<td></td>
</tr>
<tr>
<td>void insertAt(List list, int i, List nl) throws ...</td>
<td></td>
</tr>
<tr>
<td>List remove(List list, int index) throws ...</td>
<td></td>
</tr>
<tr>
<td>List insertSorted(List list, int value) throws ...</td>
<td></td>
</tr>
<tr>
<td>List sort(List list) throws ...</td>
<td></td>
</tr>
</tbody>
</table>
Ex5.Q1 – toString(List list)

```java
public static String toString(List list) {
    if (list == null)
        return "null";
    return list.value + "," + toString(list.next);
}
```

```
myList
  value 76
  next
  value 15
  next
  value 22
  next
  value 3
  next
  value 32
  next
```

```
76,15,22,3,34,null
```

```
u5a1.Lists.toString(myList)
```
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, otherwise the null)
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!
Java Inheritance and Interfaces

Outline

- Person class example
- Inheritance
- Type compatibility
- Polymorphism
- instanceof
- Visibility rules
- Constructor and `super()`
- Final methods and class
- Abstract classes and methods
- Interfaces
- Abstract classes vs. interfaces
Class Person

Attributes

Accessors

Mutators

Getter and Setter Methods

Person

Name
Age
Address
PhoneNumber

toString

getName
getAge
getAddress
getPhoneNumber

setAddress(newAddress)
setPhoneNumber(newPhoneNumbers)
public class Person {
    private String m_name;
    private int m_age;
    private String m_address;
    private String m_phone;

    public Person(String name, int age, 
                   String address, String phone) {
        m_name = name; m_age = age;
        m_address = address; m_phone = phone;
    }

    public String toString() {
        return getName() + " is " + getAge() + 
            "old and lives in " + getAddress();
    }

    public String getName() { return m_name; }
    public int getAge() { return m_age; }
    public String getAddress() { return m_address; }
    public String getPhoneNumber() { return m_phone; }

    .....
Class Student

Person
Name
Age
Address
PhoneNumber

toString()
getName()
getAge()
getAddress()
getPhoneNumber()

setAddress(newAddress)
setPhoneNumber(newPhoneNumber)

Student
Name
Age
Address
PhoneNumber
Legi

toString()
getName()
getAge()
getAddress()
getPhoneNumber()
getLegi()

setAddress(newAddress)
setPhoneNumber(newPhoneNumber)
public class Student extends Person {

    private String m_legi;

    public Student(String name, int age, String address, String phone, String legi) {
        super(name, age, address, phone);
        m_legi = legi;
    }

    public String toString() {
        return getName() + " is " + getAge() + "old, lives in " +
            getAddress() + " and has legi-nr.: " + getLegi();
    }

    public String getLegi() {
        return m_legi;
    }
}
public class Student extends Person {

    private String m_legi;

    public Student(String name, int age,
                    String address, String phone, String legi){
        super(name, age, address, phone);
        m_legi = legi;
    }

    public String toString() {
        return getName() + " is " + getAge() + "old, lives in " +
        getAddress() + " and has legi-nr.: " + getLegi();
    }

    public String getLegi() { return m_legi; }
}
public class Student extends Person {

    private String m_legi;

    public Student(String name, int age, String address, String phone, String legi){
        super(name, age, address, phone);
        m_legi = legi;
    }

    public String toString() {
        return getName() + " is " + getAge() + " old, lives in " +
        getAddress() + " and has legi-nr.: " + getLegi();
    }

    public String getLegi() { return m_legi; }
}

Student
• defines a constructor
• calls the basis class constructor through the usage of super
Inheritance

- Student extends Person
- Student can:
  - Add new fields
    - m_legi
  - Add new methods
    - getLegi()
  - Override existing methods
    - toString()
- Student cannot:
  - Remove fields
  - Remove methods
Why inheritance?

- Better design
- Code reuse
- Code «maintenance»
- Abstraction of the real world
The Object class in Java

- The Object class in Java

  - Is a superclass for all other classes defined in Java's class libraries, as well as for user-defined Java classes.
  - This does not include primitive types (char, int, float, etc.): they are not classes!

- When a class is defined in Java, the inheritance from the Object class is implicit, therefore:

  ```java
  public class MyClass {
      ......
      }
  ```

  is equivalent to:

  ```java
  public class MyClass extends Object {
      ......
      }
  ```
The Object class in Java

Quelle: sun.com
Visibility Rules

- **private members**
  - Private members in the base class are not accessible to the derived class, and also not to anyone else

- **protected members**
  - Protected members are visible to methods in a derived class and also methods in classes in the same package, but not to anyone outside

- **public members**
  - Everyone
Final Methods and Classes

- A derived class
  - Can accept the base class methods
  - Or can override the base class methods

- A method declared as **final** in the base class cannot be overridden by any derived class
- A **final** class cannot be extended!
  - E.g. Integer, Character,...
Abstract Classes

- Abstract method
  - Is a method that all derived classes must implement

- Abstract class
  - A class that has at least one abstract method

- If a class derived from an abstract class fails to override an abstract method, the compiler will detect an error
  - Eclipse: 'Hint-Bulb' provides help!
Interface

The interface in Java is the ultimate abstract class.

A class can implement many interfaces.

A class implements an interface if it provides definitions for all the methods „declared“ in the interface.

So, both abstract classes and interface provide a specification of what subclasses must do.

But....
Abstract Class vs. Interface

Abstract class
- An abstract class can provide complete code, default code, and/or just stubs that have to be overridden
- May declare methods as protected abstract
- A class may extend only one abstract class

Interface
- An interface cannot provide any code, much less default code
- All methods declared are implicitly public abstract
- A class may implement several interfaces

› http://java.sun.com/docs/books/tutorial/java/IandI/index.html
Have Fun!