Informatik II (D-ITET)

Tutorial 6

TA: Hông-Ân Cao, E-mail: hong-an.cao@inf.ethz.ch
Distributed Systems Group, ETH Zürich
Outlook

- Exercise 5 solution discussion
- Exercise 6 hints
Solution Ex5

Variables & Methods

- `beginWithLowerCase`
- `areVeryDescriptiveAnd`
- `UpperCaseSeparated`
- **Not**: `aMethodWhichHasAVeryLongName()`
- **Classes with capital letters**: `class MyClass{ ... }`
- **C++ notation**: attributes start with `m` (mValue, mNext), **not the case with temporary and passed parameters** (int tmp)

Please, comment more!
Solution Ex5.Q1

```java
public static List add(List list, int value) {
    return new List(value, list);
}

public static int size(List list) {
    if (list == null) { return 0; }
    return size(list.next) + 1;
}

public static int sum(List list) {
    if (list == null) { return 0; }
    return list.value + sum(list.next);
}

public static List last(List list) {
    if (list == null) { return null; }
    if (list.next == null) { return list; }
    return last(list.next);
}

public static List sublist(List list, int index) throws IndexOutOfBoundsException {
    if (list == null || index < 0) { throw new IndexOutOfBoundsException(); }
    if (index == 0) { return list; }
    return sublist(list.next, index-1);
}

public static int valueAt(List list, int index) throws IndexOutOfBoundsException {
    if (list == null || index < 0) { throw new IndexOutOfBoundsException(); }
    if (index == 0) { return list.value; }
    return valueAt(list.next, index-1);
}

public static int index(List list, int value) throws NoSuchElementException {
    if (list == null) { throw new NoSuchElementException(); }
    if (list.value == value) { return 0; }
    return 1 + index(list.next, value);
}
```
public static void append(List list, int value) throws IllegalArgumentException {
    if (list == null) { throw new IllegalArgumentException(); }
    u5a1.Lists.last(list).next = new List(value, null);
}

public static void concat(List head, List tail) throws IllegalArgumentException {
    if (head == null) { throw new IllegalArgumentException(); }
    u5a1.Lists.last(head).next = tail;
}

public static void insertAt(List list, int index, int value) throws IndexOutOfBoundsException {
    if (list == null || index < 0) { throw new IndexOutOfBoundsException(); }
    if (index == 0) {
        list.next = new List(value, list.next);
    } else {
        insertAt(list.next, index-1, value);
    }
}

public static void insertAt(List list, int index, List newList) throws IndexOutOfBoundsException {
    if (newList == null) { return; }
    if (list == null || index < 0) { throw new IndexOutOfBoundsException(); }
    if (index == 0) {
        u5a1.Lists.last(newList).next = list.next;
        list.next = newList;
    } else {
        insertAt(list.next, index-1, newList);
    }
}

public static List remove(List list, int index) throws IndexOutOfBoundsException {
    if (list == null || index < 0) { throw new IndexOutOfBoundsException(); }
    if (index == 0) { return list.next; }
    list.next = remove(list.next, index-1);
    return list;
}
public static List insertSorted(List list, int value) {  
    if (list == null) { return new List(value, null); }  
    if (value < list.value) { return new List(value, list); }  
    list.next = insertSorted(list.next, value);  
    return list;  
}

public static List sort(List list) {  
    if (list == null) { return null; }  
    return insertSorted(sort(list.next), list.value);  
}
public int pop() throws EmptyStackException {
    if (list == null) { throw new EmptyStackException(); }
    int res = list.value;
    list = list.next;
    return res;
}

public int peek() throws EmptyStackException {
    if (list == null) { throw new EmptyStackException(); }
    return Lists.valueAt(list, 0);
}

public void push(int number) {
    list = Lists.add(list, number);
}

public boolean empty() {
    return list == null;
}

public int size() {
    return Lists.size(list);
}
Outlook

- Exercise 5 solution discussion
- Exercise 6 hints
Java Object Oriented Concepts

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
- Generics
- Some nice insights about Java OO concepts (and syntax): https://docs.oracle.com/javase/tutorial/java/javaOO/
The Object class in Java

If you want to know more about the java.lang package:
https://docs.oracle.com/javase/8/docs/api/java/lang/package-tree.html
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, but wrappers around them exist (Integer for int)!
  - 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 {
        ......
    }
    ```
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
Class Person

Person
Name
Age
Address
PhoneNumber

toString
getName
getAge
getAddress
getPhoneNumber

setAttribute(newAddress)
setPhoneNumber(newPhoneNumber)

Attributes
Accessors
"Getters"

Mutators
"Setters"

Getter and Setter Methods
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) {
        this.m_name = name;
        this.m_age = age;
        this.m_address = address;
        this.m_phone = phone;
    }

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

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

More on objects:
https://docs.oracle.com/javase/tutorial/java/javaOO/objectcreation.html
Class Student

<table>
<thead>
<tr>
<th>Person</th>
<th>Student</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>Age</td>
<td>Age</td>
</tr>
<tr>
<td>Address</td>
<td>Address</td>
</tr>
<tr>
<td>PhoneNumber</td>
<td>PhoneNumber</td>
</tr>
<tr>
<td><strong>toString()</strong></td>
<td><strong>toString()</strong></td>
</tr>
<tr>
<td><strong>getName()</strong></td>
<td><strong>getName()</strong></td>
</tr>
<tr>
<td><strong>getAge()</strong></td>
<td><strong>getAge()</strong></td>
</tr>
<tr>
<td><strong>getAddress()</strong></td>
<td><strong>getAddress()</strong></td>
</tr>
<tr>
<td><strong>getPhoneNumber()</strong></td>
<td><strong>getPhoneNumber()</strong></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>setAddress(newAddress)</strong></td>
<td><strong>setAddress(newAddress)</strong></td>
</tr>
<tr>
<td><strong>setPhoneNumber(newPhoneNumber)</strong></td>
<td><strong>setPhoneNumber(newPhoneNumber)</strong></td>
</tr>
</tbody>
</table>
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);
        this.m_legi = legi;
    }

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

    public String getLegi() { return this.m_legi; }
}

Class Student
- extends an existing concept, the class Person
- contains a supplementary field: m_legi with Getter: getLegi()
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);
        this.m_legi = legi;
    }

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

    public String getLegi() { return this.m_legi; }
}

Class Student

- defines a constructor
- calls the base class (superclass) 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
Reminder: Type Conversion

```java
int someNumber = 12;
char someChar = (char)someNumber;
    // Explicit conversion to character 'A'
char anotherChar = 'B';
int anotherNumber = anotherChar;
    // Implicit conversion to 13
```

- byte can be converted to short, int, long, float, or double
- short can be converted to int, long, float, or double
- char can be converted to int, long, float, or double
- int can be converted to long, float, or double
- long can be converted to float or double
- float can be converted to double

Static & Dynamic Cast

Person p = new Person(...);
Student s = new Student(...);
Employee e = new Employee(...);

Person ps = s  →  ok
Person pe = e  →  ok
Student sp = p  →  compilation error
Student sps = ps  →  compilation error
Student dsp = (Student) ps  →  ok
Employee deps = (Employee) ps  →  runtime error
Static & Dynamic Cast

p instanceof Person → true
p instanceof Student → false
s instanceof Person → true
s instanceof Student → true

Person p = new Person(...);
Student s = new Student(...);
Employee e = new Employee(...);
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

Source: [http://docs.oracle.com/javase/tutorial/java/IandI/index.html](http://docs.oracle.com/javase/tutorial/java/IandI/index.html)
Example: Interface IStack

```java
public interface IStack {
    int size();
    void push(Object obj);
    Object pop();
    Object peek();
    boolean empty();
}
```

```java
public class MyStack implements IStack {
    private int size;

    public int size() {
        return size;
    }

    public void push(Object obj) {
        ... 
    }

    ... 
}
```
Example: Abstract Class `BaseStack`

```java
public abstract class BaseStack implements IStack {
    public abstract int size();
    public abstract void push(Object obj);
    public abstract Object pop();
    public Object peek() {
        Object top = pop();
        push(top);
        return top;
    }
    public boolean empty() {
        return size() == 0;
    }
}
```

```java
public class MyStack extends BaseStack {
    private GenericList first;

    public Object peek() {
        return first.value;
    }
    ...
}
```
Hints for Exercise Sheet 6

- Q1: Classes, Interfaces
- Q2: Interfaces and Implementation
- Q3: Polymorphism
- Q4: Generics
- Q5: Stack (again): Voluntary Exercise Submission
Hints: UML

A keyword represents an interface

```
<<interface>>
ISensor
aktivieren()
lesen()
```

Notation for the dependance of the instantiation of the interface. Thermal sensor instantiates the iSensor interface

```
<<interface>>
ISensor
aktivieren()
lesen()
```

Wärmesensor
aktivieren()
lesen()

--

Keyword

Property

Section with attributes (detailed representation)

```
<<gui>>
Window {abstract}
+ size: Area = (100,100)
# visibility: Boolean = true
+display()
+hide()
```

Section with operations (detailed representation)

```
Person
name: String
vorname: String
```

Private client as a specialized Person

```
Privatkunde
kundennummer: Integer
```

http://de.wikipedia.org/wiki/Klassendiagramm
Hints Ex6.Q2 – Factory Method

A factory method builds an object which implements a certain interface, but the inner functionality of the object is hidden.

→ Programmer 1 implements different lists which implement the IList interface.

→ Programmer 2 uses lists but doesn’t want to know about the functionality. When Programmer 1 writes a new implementation, Programmer 2 has to rewrite all lines of new ListA() as new ListB().

→ Programmer 1 puts a factory method at disposal and Programmer 2 can always call for example Factory.giveMeNewList() and gets an object from the newest implementation of the IList interface.
Hints Ex6.Q2 – Factory Method

```java
interface Currency {
    String getSymbol();
}

// Concrete Rupee Class code
class Rupee implements Currency {
    @Override
    public String getSymbol() {
        return "Rs";
    }
}

// Concrete SGD class Code
class SGDDollar implements Currency {
    @Override
    public String getSymbol() {
        return "SGD";
    }
}

// Concrete US Dollar code
class USDollar implements Currency {
    @Override
    public String getSymbol() {
        return "USD";
    }
}

// Factory Class code
class CurrencyFactory {
    public static Currency createCurrency (String country) {
        if (country.equalsIgnoreCase("India")) {
            return new Rupee();
        } else if (country.equalsIgnoreCase("Singapore")) {
            return new SGDDollar();
        } else if (country.equalsIgnoreCase("US")) {
            return new USDollar();
        } else {
            throw new IllegalArgumentException("No such currency");
        }
    }
}

// Factory client code
public class Factory {
    public static void main(String args[]) {
        String country = args[0];
        Currency rupee = CurrencyFactory.createCurrency(country);
        System.out.println(rupee.getSymbol());
    }
}
```

Source: [http://javarevisited.blogspot.ch/2011/12/factory-design-pattern-java-example.html#ixzz2Q9Qv3fXp](http://javarevisited.blogspot.ch/2011/12/factory-design-pattern-java-example.html#ixzz2Q9Qv3fXp)
Hints Ex6.Q3 – Generic Lists

- Exercise sheet 5
  - Elements of the list: Integers
    - `int`

- Exercise sheet 6
  - Elements of the list: generic objects
    - `Object`

- Build your own utility class: `ListUtils`
  - `implements IListUtils: manage generic lists`
    - Compare with the utility classes in Q1 and Q3 of Exercise Sheet 5
    - This time the utility class is instantiated (not static)
Hints Ex6.Q3 – Generic Lists

class List {
    int value;
    List next;

    public List(int v, List e){
        value = v;
        next = e;
    }
}
Hints Ex6.Q3 – Generic Lists

class GenericList {
    Object value;
    GenericList next;

    public GenericList(Object v, GenericList e){
        value = v;
        next = e;
    }
}
Hints Ex6.Q3a and c – Generic Lists

- Methods are not static anymore!
- Solution Ex5.Q1: `toString`, `add`, `size`
  - Can be easily passed
- Solution Ex5.q3: `sort`
  - Similar (same idea)
  - Minimal interfacing, this way, generic object can also be used:
  - Interface `Comparable`

```java
public interface Comparable{
    boolean smallerThan(Comparable rhs);
}
```

- You can cast as `Comparable` without checking the type
Hints Ex6.Q3b

- Trivial

- Will be used in your tests for your list!
  - First implement…
  - …then look for errors in GenericLists 😊
Tips on Ex6Q4 – Generics

- U6 Generics
  - All classes inherit from `Object` (abstract base class)
  - cast when extended from container (here List)
    ```java
    MyType Elem = (MyType) Kollektion.getNext();
    such casts can lead to runtime ClassCastException
    ```

Better this way:
```
Object obj = Kollektion.getNext();
if ( obj instanceof MyType )
  doSomething( (MyType)obj );
```
Tips on Ex6Q4 – Generics

- U6 Generics
  - Collection of Java Generics (generic class)
    ```java
class MyPair<T>
{
    public T first, second;
}
```
  - An object `pair of type MyPair<Float>` contains two Float references:
    `pair.first` und `pair.second`
  - An object `pair of type MyPair<Integer>` contains two Integer references: `pair.first` und `pair.second`

- Advantage of generics:
  - Type testing through the compiler and its acquisition through the container → No dynamic casting is necessary
Tips on Ex6Q4 – Generics

- ArrayList Container

Double nesting:
- ArrayList contains groups
  ```java
  ArrayList<ArrayList<Student>> groups;
  ```
- Groups contain students
  ```java
  ArrayList<Student> group;
  ```

- Filter: "can obtain Testat"
Tips on Ex6Q4 – Generics

a. FilterFactory and (empty) IFilter implementation
   Input: ArrayList of groups, that are actually ArrayLists of students.
   Output: ArrayList of students obtaining the Testat.

b. Implementation of filterRaw
   - No Generics: ArrayList as raw type (compiler warnings)
   - Filter out all students who do not have enough points for the Testat...
   - When taking them out first from ArrayList, then cast to Student

c. Implementation of filterGeneric
   - ArrayList<T> indicates what is stored inside it
   - Type checking when adding elements to the list
   - ArrayList<T> directly provides objects of the correct type (no casting required)
Hints for E6.Q5 : A Stack Again (voluntary)

- Non-trivial (that’s why it's on a voluntary basis)

- Self-test: Whoever can do it will have no problem during the exam.
  - Promised 😊

- Combines the efficiency of arrays to the effortless growth of lists

- Implements the IStack Interface
  - Can be used in u6a2.StackFactory.create()
Have Fun!

Source: http://www.bonkersworld.net/object-world/