Informatik II (D-ITET)

Tutorial 6

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

- Exercise 5: Solution discussion
- More Java insights (Inheritance and Interfaces)
- Exercise 6: Overview (Inheritance, Polymorphism, Interfaces)
Announcement

- The week following Easter holiday, this tutorials will not take a place! 15-16, Wednesday-Thursday, April
  
  - Mi 13-14 HG G 3  Anwar Hithnawi
  - Do 13-14 CAB G 51  Anwar Hithnawi
  - Do 13-14 HG D 5.3  Hossein Shafagh

- Check tutorial list: [http://www.vs.inf.ethz.ch/edu/FS2015/I2/]
Solution Ex5.Q1, Q2, Q3 - A Simple Linked List
Solution Ex5.Q4 – List with Stack
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(); }
    Lists.last(list).next = new List(value, null);
}

public static void concat(List head, List tail) throws IllegalArgumentException {
    if (head == null) { throw new IllegalArgumentException(); }
    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) {
        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) {
        list.next = remove(list.next, index-1);
    } else {
        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);
}
Exercise Sheet 5: Solutions and Remarks

Variables and Methods: (Naming convention)

- beginWithLowerCase
- areVeryDescriptiveAnd
- upperCaseSeparated
- Classes with capital letters: class MyClass{
- C++ notation: attributes start with \textit{m} (mValue, mNext), not the case with temporary and passed parameters (int tmp)

Please, comment more! important @author: your names
Outlook

- Exercise 5: Solution discussion

- More Java insights (Inheritance and Interfaces)

- Exercise 6: Overview (Inheritance, Polymorphism, Interfaces)
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

**Person**

- Name
- Age
- Address
- PhoneNumber

**toString**

**getters**

- getName
- getAge
- getAddress
- getPhoneNumber

**mutators**

- setAddress(newAddress)
- setPhoneNumber(newPhoneNumber)

**Attributes**

**Accessors**

**Mutators**

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) {
        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; }
}
Class Student

- defines a constructor
- calls the basis class constructor through the usage of `super`

```java
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; }
}
```
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
Static & Dynamic Cast

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

Person ps = s $\rightarrow$ ok
Person pe = e $\rightarrow$ ok
Student sp = p $\rightarrow$ compilation error
Student sps = ps $\rightarrow$ compilation error
Student dsps = (Student) ps $\rightarrow$ ok
Employee deps = (Employee) ps $\rightarrow$ runtime error
Static & Dynamic Cast

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

- p instanceof Person → true
- p instanceof Student → false

- s instanceof Person → true
- s instanceof Student → true
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

Object

Person

Student

Employee

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
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;
    }

    ...
}
```
Factory Classes

- Intuitively: The idea behind interface is, that functionality is replaceable. However: What is the point behind replaceability, if we have to modify code at several different places, in order to replace objects?

- Therefore: Factory-Classes
Factory Classes

- Intuitively: The idea behind interface is, that functionality is replaceable. However: What is the point behind replaceability, if we have to modify code at several different places, in order to replace objects?

- Therefore: Factory-Classes
Factory-Classes: The difference

interface Thing

getThing(), setThing()

class SimpleThing implements ThingInterface

class ComplexThing implements ThingInterface

- «Traditional» Implementation (at 100x in the code...):
  Thing myThing = new SimpleThing();

- Implementation with «ThingFactory»:
  Thing myThing = ThingFactory.createThing();
Factory-Classes: ThingFactory

class ThingFactory {
    public static Thing createThing() {
        return new SimpleThing();
    }
}

Outlook

- Exercise 5: Solution discussion
- More Java insights (Inheritance and Interfaces)
- Exercise 6: Overview (Inheritance, Polymorphism, Interfaces)
Tips for Exercise Sheet 6

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

A keyword represents an interface

There is a notation for the dependance of the instantiation of the interface. Thermal sensor instantiates the iSensor interface.

Keyword | Property
--- | ---
<<interface>> | ISensor
aktivieren() | lesen()

Section with attributes (detailed representation)

Section with operations (detailed representation)

Private client as a specialized Person

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 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.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!
- **Ex5.Q1:** `toString`, `add`, `size`
  - Can be practically inherited
- **Ex5.q3:** `sort`
  - Similar (same idea)
  - Small modifications, so it can work with generic object elements
  - Interface `Comparable`

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

- You are allowed to cast it to `Comparable` without checking
Hints Ex6.Q3b

- Trivial

- Will be used in your tests for your list!
  - First implement…
  - …then look for errors in GenericLists 😊
Tips on Ex6.Q4 – Generics

- 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:

  ```java
  Object obj = Kollektion.getNext();
  if (obj instanceof MyType)
    doSomething((MyType)obj);
  ```
Tips on Ex6.Q4 – Generics

- **U7 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` and `pair.second`

  - **An object** `pair of type` `MyPair<Integer>` contains two `Integer` references: `pair.first` and `pair.second`

- **Advantage of generics:**
  - Type check at the compiler time which increases type safety
  - Compiler takes care of type casting
Tips on Ex6.Q4 – Generics

- ArrayList Container

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

- Filter: „can obtainTestat”
Tips on Ex6.Q4 – 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
   
   - filterRaw(ArrayList)
   - No Generics: ArrayList as raw type (compiler warnings)
   - Filter out all students who do not have enough points for the Testat...

c. Implementation of filterGeneric
   
   - filterGeneric(ArrayList<ArrayList<Student>>)
   - 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 Ex6.Q4 : A Stack Again (Advanced)

- Non-trivial (that’s why no point)

- Self-test: Whoever can won’t have any programming problems during the exam.
  - Promised 😊

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

- Implements the `IStack` Interface
  - Can be used in `u6a2.StackFactory.create()`
Happy Holidays!