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)
Solution Ex5.Q1, Q2, Q3 - A Simple Linked List
Solution Ex5.Q4 – List with Stack
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; }
    if (list.next.value == value) { return 1 + index(list.next, value); }
    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;
    } 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);
}
Ex5.Q4

```java
public int pop() throws EmptyStackException {
    if (list == null) {
        throw new EmptyStackException();
    }
    int res = list.value;
    list = list.next;
    return res;
}

class MyStack {
    List.Node list;
    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 `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)
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; }
}

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);
        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; }
}
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
Polymorphism

public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}

- A Deer IS-A Animal
- A Deer IS-A Vegetarian
- A Deer IS-A Deer
- A Deer IS-A Object

Deer d = new Deer();
Animal a = d;
Vegetarian v = d;
Object o = d;
The Object class in Java
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

<table>
<thead>
<tr>
<th>Abstract class</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>- An abstract class can provide complete code, default code, and/or just stubs that have to be overridden</td>
<td>- An interface cannot provide any code, much less default code</td>
</tr>
<tr>
<td>- May declare methods as protected abstract</td>
<td>- All methods declared are implicitly public abstract</td>
</tr>
<tr>
<td>- A class may extend <strong>only one</strong> abstract class</td>
<td>- A class may implement several interfaces</td>
</tr>
</tbody>
</table>

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-Cla**sses:** **ThingFactory**

```java
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: Stack (again): Voluntary Exercise Submission
Hints

A keyword represents an interface

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

http://de.wikipedia.org/wiki/Klassendiagramm
Q1: Classes, Interfaces, Casting

a) Which types cannot be instantiated?

b) Which lines from StaticCasts.txt are invalid?

c) Which lines from DynamicCasts.txt are invalid?

```java
interface A {
}
abstract class B implements A {
}
interface C extends A {
}
class D extends B implements C {
}
class E extends B {
}
class F implements C {
}

```
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
Ex6.Q4 „Optional“: ChunkedStack

- Third stack implementation: list of arrays
  - Arrays have a fixed capacity
  - Class remembers size of current chunk (i.e., first array in the list)

a) Implement all methods according to the interface
  - **push**: Add element to stack; if needed grow stack by creating a new array and prepending(!) it to the list. Store element as 0-th element of the new array.
  - **pop**: Remove element from stack and remove chunk if needed

b) Use the Tests to test the implementation. Don’t forget to modify the FactoryMethod to create a *ChunkedStack*

c) Use the Tests to check the implementation’s performance: The test prints the run times on the console. Is the RT-behavior good or bad? Plot the numbers!
4) „Optional“: ChunkedStack (ctd.)

d) size() is called quite often. What is remarkable about its implementation? Will size() get faster or slower when the stack grows?

e) What would be a better way to implement size()? Implement it and check the implementation’s performance again - and plot it! Has the RT-behavior improved?

Each call of size() goes through the whole list to count ist size – more efficient: store and update