Informatik II (D-ITET)
Tutorial 7

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

- Exercise 6: Solution discussion
- Exercise 7: Overview (Generics, Binary Trees, Reversi)
Polymorphism

- **Polymorphism** is the ability of an object to take on many forms.

```java
// assume appropriate ctors
Person p[] = new Person[3];
p[0] = new Person("Tim");
p[1] = new Student("Cara", 1234);
p[2] = new Faculty("Mia", "ABCD");

for(int i = 0; i < p.length; i++)
{
    System.out.println( p[i] );
}
```

**Output**

Tim
1234: Cara
ABCD: Mia
Solution U6.A1 – Classes and Interfaces

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

X x = new X();

Can be instantiated:  
Non-abstract classes (D, E, F)
Solution U6.A1 – Classes and Interfaces

Type casts

- **Static** (*implicit cast*):
  only subclasses to parent classes

- **Dynamic** (*explicit cast*):
  \[ T \ t = (T) \text{obj}; \]
  Valid, if the actual object pointed to by reference \text{obj} is of type \( T \) (including all children of \( T \))

```java
public static void d3()
{
    B b = new D();
    A a = (A) b;
    C c = (C) b; // cross-cast!
    D d = (D) b;
    E e = (E) b;
}
```
public static void c1() {
    D d = new D();
    A a = d;
    B b = d;
    C c = d;
    E e = d;
    F f = d;
}

public static void c6() {
    C c = new F();
    A a = c;
    B b = c;
    D d = c;
    E e = c;
    F f = c;
}
Solution U6.A1 – Classes and Interfaces

Interfaces vs. Abstract Class: why Interfaces?

- Functionality is an important point in the program. *what* is done *where* and *who* has access?

- Interfaces represent exactly this concept. It is guaranteed, *what* is done exactly and the interface defines it (*who* and *where*). The implementation (*how*) is completely irrelevant.
Solution U6.A2 – IStack expanded

Eclipse DEMO
Solution U6.A3 – Generic Lists

```java
/**
 * Inserts a value into a sorted list so that
 * the resulting list is still sorted.
 * The sort order is ascending.
 */
private GenericList insertSorted(GenericList list, Object value) {
    if (list == null)
        return new GenericList(value, null);

    Comparable lhs = (Comparable) value;
    Comparable rhs = (Comparable) list.value;
    if (lhs.compareTo(rhs))
        return new GenericList(value, list);

    list.next = insertSorted(list.next, value);
    return list;
}
```

Generic objects compared through interfaces
Outlook

- Exercise 6: Solution discussion

- Exercise 7: Overview (Generics, Binary Trees, Reversi)
Tips on U7.A1 – 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:

```java
Object obj = Kollektion.getNext();
if (obj instanceof MyType)
  doSomething((MyType)obj);
```
Tips on U7.A1 – 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 U7.A1 – 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 U7.A1 – 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 to Ex7.Q2 – Tic-Tac-Toe

- Reflections on game trees…

- Think about how the attribute of a node is calculated based on the attributes of the successor, when you are for example in your opponent’s place

- Theoretical exercise: Draw a game tree given the following game state:

```
    X
   / 
  X O
```

- Mark all situations (starting from the bottom) with \{-1, 0, 1\} depending or possible outcome of the game.

- What is the optimal move?
Tips on U7.A3 – Binary Tree

- Each node contains pointers to:
  - Left successor
  - Right successor
  - (Parent)

- Recursive traversal:
  - Pre-order: P-L-R
  - In-order: L-P-R
  - Post-order: L-R-P
Tips on U7.A3 – Binary Search Trees

- **Structure:**
  - The nodes contain data elements, or pointers to data elements *(record)*
  - Each node also has a key attribute *(key)*
  - The set of key attributes is totally ordered *(a≤b)*
  - Search is done by key comparison

- For every node with key attribute *s*, we have:
  - All keys in the left subtree are smaller than *s*
  - All keys in the right subtree are greater than *s*

- The subtrees are also binary search trees

- See elementary methods in the slides of lecture 7!
Tips on U7.A3 – Binary Search Tree

Methods to implement:

- height, isLeaf, hasOneChild
- preOrder, inOrder, postOrder
- insert
- find
- getSmallest & remove
Tips on U7.A3 – Binary Search Tree

Subtask a (by hand)
Delete, replace smallest element of the right subtree

Subtask b
Implementation of a binary search tree

IBinarySearchTreeUtils<T>

UtilsFactory.create() should generate a Utils for the type String ➔
new MyTreeUtils<String>()

getSmallest(): the smallest element in the given tree
Tips on U7.A3 – Binary Search Tree - Remove

Strategy: Replace with smallest element in the right subtree

```
40
  /   \
20     70
  / \
10   30
  /  \
9    31
  /   \
50     60
     /  \
61     51
```


Reversi

- This task starts a series, that aims to implement Reversi Player

- Online platform to test players in advance
  - reversi.ethz.ch

- Rules and more information:
  - Pictures from last year: http://www.vs.inf.ethz.ch/edu/FS2016/I2/reversi/
  - Login for reversi-papers:
    - username: i2bib
    - password: reversi
Reversi

- Reversi tournament at the end of the semester
  - Mittwoch, 31.05.2017, ab 12:30 Uhr, im CABinett (Stuz2)

- In case of problems with the framework:
  1. Documentation
  2. Me
  3. Leyna Sadamori (leyna.sadamori@inf.ethz.ch)
Last year prizes
Tips on U7.A4 – Reversi [Part 1]

- First, the basic principles of the game are to be implemented

- Later, strategies are developed to improve the game of the computer player
  - Part 1: RandomPlayer
  - Part 2: GreedyPlayer
  - Part 3: MinMaxPlayer
  - Part 4: AlphaBetaPlayer
  - Tournament: WinnerPlayer 😊
Tip

Tips on U7.A4 – Reversi [Part 1]

- Resources are found on the Reversi website
- Note the installation instructions (Eclipse >3.2)
- HumanPlayer in u7a3
- Trick:
  - You first create an Abstract class (PlayerBase,…etc)
  - Implement functions for your different players in your general useful helper functions
Tips on U7.A4a – Play! [Part 1]

- Setup Framework
- Play a game against your team mate (or yourself)
- Take snapshot
Tips on U7.A4b – ReversiPlayer

package reversi;
public interface ReversiPlayer
{
    void initialize(int myColor, long timeLimit);
    Coordinates nextMove(GameBoard gb);
}

package randomTeam;
public abstract class PlayerBase implements ReversiPlayer
{
    private int m_color = 0;
    private long m_timeout = 0;
    protected final int getColor() { return m_color; }
    protected final long getTimeout() { return m_timeout; }
    ...
    protected abstract void foo();
}

package randomTeam;
public class RandomPlayer extends PlayerBase {
    protected void foo() { ... }
    ...
}

- Implement a computer player, that randomly selects a valid move
- Possible strategy (naïve)
  - Pick a random move
  - Then check whether it is valid or not
    - If valid → return
    - If not valid → ?
- Possible strategy
  - In an array, mark all possible moves
  - Randomly select a move from this array
    - Extremely more efficient
    - Standard approach afterwards → evaluate moves
Happy Holidays!