Overview

- Debriefing Exercise 6
- Briefing Exercise 7
U6.A1 – Classes and interfaces

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

Type casts

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

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

```java
class B { public void m(); } 
class A extends B { }
class C extends B { }
class D extends B { }
class E extends D { }

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;
}
```
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
Java Generics

- Generics allow parameterization of types
  - Input to formal parameters are values (e.g. \( f(\text{int } a) \))
  - Input to type parameters are types e.g. `ArrayList<T>`

- Reuse the same code for different input types
- Same algorithm
  - E.g. sorting Integers, Floats, Students etc.

- Stronger type checks at compile time
  - Compile-time errors are easier to fix than run-time errors
  - No need to typecast

- Code is easier to maintain and read
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`
Generics

- All classes inherit from Object \(\text{(abstract base class)}\)
- Cast when extended from container (here List)
  
  \[
  \text{MyType} \ \text{Elem} = (\text{MyType}) \ \text{Kollektion.getNext}();
  \]
  such casts can lead to runtime ClassCastException

Better this way:

\[
\text{Object} \ \text{obj} = \ \text{Kollektion.getNext}();
\]

\[
\text{if}(\ \text{obj} \ \text{instanceof} \ \text{MyType} )
\]

\[
\text{doSomething}( (\text{MyType})\text{obj} );
\]
/**
 * 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.smallerThan(rhs))
        return new GenericList(value, list);

    list.next = insertSorted(list.next, value);
    return list;
}
U6.A3 Implementing the Comparable interface

```java
package u6a3;

/**
 * abstract class for geometric objects
 */
public abstract class GeometricObject implements Comparable {
    public abstract int area();

    public boolean smallerThan(Comparable rhs) {
        GeometricObject other = (GeometricObject) rhs;
        return this.area() < other.area();
    }
}
```
Overview

- Debriefing Exercise 6
- Briefing Exercise 7
U07.A01

- **ArrayList and Generics**
  - Each group consists of multiple students: 
    ```java
    ArrayList<Student> group
    ```
  - There are multiple groups of students: 
    ```java
    ArrayList<ArrayList<Student>> groups;
    ```

a) Implement factory method
b) Implement filterRaw (without generics: ArrayList)
c) Implement filterGeneric (using Generics: ArrayList<Student>)
U7.A2 Tic-Tac-Toe

- Game tree given the following game state
- Mark all situations (starting from the bottom) with \{-1, 0, 1\} depending on the possible outcome of the game
- What is the optimal move?
Reminder: Binary Trees

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

- Recursive traversal:
  - Preorder: W-L-R
  - Inorder: L-W-R
  - Postorder: L-R-W
Tree traversal:


Inorder: A, B, C, D, E, F, G, H, I

Postorder: A, C, E, D, B, H, I, G, F
Binary Search Tree

- **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
U7.A3 Binary Search Trees

a) Delete elements 15, 12, 20

b) Implement

IBinarySearchTreeUtils<T> and
UtilsFactory.create

- height
- isLeaf
- hasOneChild
- preOrder
- inOrder
- postOrder
- insert
- find
- remove
- unlinkSmallest
U7.A4 Reversi

- Game

- Ongoing series until the end of the semester
- Tournament at the end!
- Cool prizes!
U7.A4 Reversi

a) Reversi framework
   - Setup the framework
   - Play a game against your team mate (or yourself)
   - Take snapshot

b) Implement a Random Player
   - 2 strategies
     1. Find a random move. If valid accept, otherwise?
     2. Find all possible moves. Choose one at random.
How to do it?

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

```java
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();
}
```

```java
package randomTeam;
public class RandomPlayer extends PlayerBase {
    protected void foo() {
        ...
    }
    ...
}
```
Have Fun!