

Advanced Aspects of Object-Oriented Programming (SS 2015)

Practice Sheet 5 (Hints and Comments)

Exercise 1 Wildcards I

```
a) // legal, parameters are the same on both sides
Vector<LinkedList<String>> v1 = new Vector<LinkedList<String>>();

// illegal, rhs has the parameter LinkedList<String> and lhs List<String>,
// thus the parameters are not equal and due to the invariance
// of generic types, the assignment is invalid.
Vector<List<String>> v2 = v1;

// legal, note that a lot of information about the element type are lost,
// and cannot be recovered, this may lead to problems when using this vector
Vector<?> v3 = v1;

// legal, LinkedList<String> matches <? extends List<?>>
// because String matches ? and LinkedList is a subtype of List
Vector<? extends List<?>> v4 = v1;

// illegal, the rhs has an unknown element type, which may be
// incompatible with List
Vector<? extends List<?>> v5 = v3;

// legal
Vector<? extends List<String>> v6 = v1;

// illegal, the parameter of LinkedList<?> may be anything, especially not String,
// thus LinkedList<String> is not a supertype of LinkedList<?>
Vector<? super LinkedList<?>> v7 = v1;

b) List<Number> l1;
// all methods of the interface List and the superinterfaces are available.
// - add (and similar methods) accept any subtype of Number
// - get (and similar) return a value with the static type Number

List<? super Number> l2;
// all methods are available
// - add accepts any subtype of Number
// - get returns its values with the static type Object

List<? extends Number> l3;
// - add is not available, because the compiler cannot ensure, that
// the dynamic type of the added object is compatible to generic
// element type of the List. This would need a runtime check, which
// is impossible due to type erasure.
// - get return the value with static type Number

List<?> l4;
// - add is not available, see above
// - get returns the value with static type Object, which is possible
// because Object is the common supertype for all objects.
```

Exercise 2 Wildcards II

Solution for a) and b)

```
import java.util.Collection;
import java.util.Iterator;

public class CollectionTools {
    public static <A> void copyFromArray(A[] arr, Collection<? super A> coll) {
```

```

    for (A el : arr) {
        coll.add(el);
    }
}

public static <A> void copyToArray(Collection<? extends A> coll, A[] arr) {
    Iterator<? extends A> iter = coll.iterator();
    int i=0;
    while(iter.hasNext() && i<arr.length) {
        arr[i] = iter.next();
        i++;
    }
}
}
}

```

Exercise 3 Bounded Type Parameters

```

import java.util.*;

public class MinMaxWrapper<A extends Comparable<A>>
    implements Set<A> {
    private Set<A> theSet;

    public MinMaxWrapper(Set<A> set) {
        this.theSet = set;
    }

    public boolean add(A e) {
        return theSet.add(e);
    }
    public boolean addAll(Collection<? extends A> c) {
        return theSet.addAll(c);
    }
    public void clear() {
        theSet.clear();
    }
    public boolean contains(Object o) {
        return theSet.contains(o);
    }
    public boolean containsAll(Collection<?> c) {
        return theSet.containsAll(c);
    }
    public boolean equals(Object o) {
        return theSet.equals(o);
    }
    public int hashCode() {
        return theSet.hashCode();
    }
    public boolean isEmpty() {
        return theSet.isEmpty();
    }
    public Iterator<A> iterator() {
        return theSet.iterator();
    }
    public boolean remove(Object o) {
        return theSet.remove(o);
    }
    public boolean removeAll(Collection<?> c) {
        return theSet.removeAll(c);
    }
    public boolean retainAll(Collection<?> c) {
        return theSet.retainAll(c);
    }
    public int size() {
        return theSet.size();
    }
}

    public Object[] toArray() {
        return theSet.toArray();
    }
    public <T> T[] toArray(T[] a) {
        return theSet.toArray(a);
    }

    public A getMinimum() {
        A res;
        Iterator<A> iter = theSet.iterator();
        if(iter.hasNext())
            res = iter.next();
        else
            return null;

        A next;
        while(iter.hasNext()) {
            next = iter.next();
            if(next.compareTo(res)<0) {
                res = next;
            }
        }

        return res;
    }

    public A getMaximum() {
        A res;
        Iterator<A> iter = theSet.iterator();
        if(iter.hasNext())
            res = iter.next();
        else
            return null;

        A next;
        while(iter.hasNext()) {
            next = iter.next();
            if(next.compareTo(res)>0)
                res = next;
        }

        return res;
    }
}
}

```

Exercise 4 Extended Iterators

The following code is one possibility to choose the generic types. It is possible to achieve the same (or nearly the same) behaviour and reusability by putting wildcards into the interfaces and removing some wildcard from the iterator classes. It was a design decision, that the Transformer and Predicate interfaces should be as easy as possible.

You can find such iterators in the *common collections* framework of the apache foundation (see <http://commons.apache.org/collections/>, the repository contains a version with generics).

a) Transformer

```
public interface Transformer<A,B> {
    public B transform (A in);
}
```

Iterator

```
import java.util.*;

public class TransformingIterator <I,O> implements Iterator<O> {

    private Iterator<? extends I> in;
    private Transformer<? super I, ? extends O> transformer;

    public TransformingIterator(Iterator<? extends I> inputIterator,
        Transformer<? super I, ? extends O> transformer) {
        this.in = inputIterator;
        this.transformer = transformer;
    }

    public boolean hasNext() {
        return in.hasNext();
    }

    public O next() {
        return transformer.transform(in.next());
    }

    public void remove() {
        in.remove();
    }
}
```

b) `public class CurrencyTransformer implements Transformer<Number, String> {`

```
    public String transform (Number n) {
        return "EUR_" + new Double(Math.floor(n.doubleValue() * 100) / 100).toString();
    }
}

import java.util.*;

public class Bill {
    public static void main (String... arg) {
        List<Double> doubles = Arrays.asList(19.248, 7.0, -9.0, 1.8882, -0.1992);

        System.out.println("Normal_iterator:");
        Iterator<?> it = doubles.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }

        System.out.println("Extended_iterators:");
        Iterator<String> truncIt = new TransformingIterator<Double, String>(doubles.iterator(),
            new CurrencyTransformer());
        while (truncIt.hasNext()) {
            System.out.println(truncIt.next());
        }
    }
}
```

Exercise 5 Aliasing

An alias is called dynamic, if the involved variables belong to the stack, e.g. local variables or method parameters.

- a) Desired dynamic alias: iterate through an array. The loop variable tmp aliases the array elements one after another.

```
class C {
    Object[] values;

    public void m () {
        for (Integer tmp : values) {
            ...
        }
    }
}
```

Undesired dynamic alias: Store an alias to an object in the local variable tmp; hideAll components and make the aliased one visible again. This breaks the invariant and is therefor an undesired alias.

```
class C {
    // invariant: either all components are visible or all are hidden.
    JComponent[] values;

    public void m () {
        JComponent tmp = values[0];
        hideAll();
        tmp.setVisible(true);
    }

    public void hideAll() {
        for (JComponent c : values) {
            c.setVisible(false);
        }
    }
}
```

- b) Aliasing means, that in a program state more than one reference to an object exists. Capturing occurs if a reference, which was passed as a parameter to a method, is stored in a field of the called object. If this is done, the reference remains available even after the method has been finished. I.e. Capturing happens, if a new static alias to a parameter object is established. Leaking means, that a reference to an object is passed outside. This means, that the caller of the leaking method can establish an alias to the leaked object.
- c) Instead of returning the reference to the original array, return a copy of the array. Depending on the characteristics of the type Identity and the security needs, it may be sufficient to return a new array, which references the original Identity objects, or one needs to return a deep copy of the array.

```
public Identity[] getSigners() {
    Identity[] signers = new Identity[this.signers.length];
    System.arraycopy(this.signers, 0, signers, 0, signers.length);
    return signers ;
}
```