Prof. Dr. A. Poetzsch-Heffter
Mathias Weber, M.Sc.

University of Kaiserslautern
Department of Computer Science
Software Technology Group

Advanced Aspects of Object-Oriented Programming (SS 2013)

Practice Sheet 5 (Hints and Comments)

Exercise 1 Wildcards and Type Bounds 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 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 and Type Bounds 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++;
    }
  }
}
```

Solution for c)

```
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 3 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 4 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 (JCompoment 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 ;
}
```

Exercise 5 Generics (optional)

a)
```
List<String>[] stringLists = new List<String>[1]; // assume this to be legal
List<Integer> intList = Arrays.asList(42); // create a list containing Integer objects
Object[] objects = stringLists;            // arrays are covariant: A[] is a subtype of B[],
                                           // if A is a subtype of B -> this assignment
                                           // is legal (both statically and dynamically)
objects[0] = intList;                      // store some junk into the array
                                           // statically: objects[0] has type Object,
                                           // intList has type List<Integer> (subtype of Object) ,
                                           // dynamically: objects has type List[] (the type erasure
                                           // deleted the generic parameter), thus object[0] has type
                                           // List, intList has type List
                                           // -> assignment is legal (both statically and dynamically)
String s = stringLists[0].get(0);          // statically: stringLists[0] has type List<String> and get
                                           // returns a String
                                           // dynamically: stringLists[0] has type List, get returns
                                           // an Object, which is casted by an automatically inserted
                                           // cast to String, but the runtime type of the first element
                                           // is Integer. -> Results in a ClassCastException
```

This code snippet would break the guarantee of the type system, that the casts that are inserted by the compiler during the type erasure will never fail. Furthermore, one could not rely on the type of array components anymore, making arrays nearly impossible to use and possibly breaking old code. In general, the java type system guarantees, that in an array only objects of the component type are stored. It is possible to construct a similar code snippet as above, that tries to store an Integer object into a String array. That code would compile but throws an ArrayStoreException, when one actually tries to store the wrong type in the array. So a "illegal" array is never created. This is possible because the component type is available at runtime and therefor assignments can be checked against it.

b)
```
public static <A> List<A> flatten (List<? extends List<? extends A>> l) {
    List<A> ret = new LinkedList<A>();
    for (List<? extends A> e :l) {
        ret.addAll(e);
    }
    return ret;
}
```

The method takes a list of lists of anything. So a first attempt for the type of `l` would be `List<List<?> >`. The get and put principle says, that we need to use a extend wildcard: `List<? extends List<?> >`. We want to return a List of something, so as a first return type we could think about `List<?>`. If we leave it that way, the programmer, who calls this method, has to use wildcards in his code as well and cannot do anything useful with the returned list. We now want to ensure, that the wildcard in the nested list of the parameter and the one in the return type are subtypes , so we introduce a type parameter and a extends clause on this parameter, leading to the signature above. With this version the caller can use concrete types and be sure to get back what he expects.

Depending on the usage of the return value, the type inference algorithm may fail to find a correct value for the type parameter, such that the programmer has to provide the parameter explicitly at the call site. If the method `flatten` is implemented in the class `GenericClass` this look like

```
List<List<Integer>> li = new ArrayList<List<Integer>>();
List<Number> ln = GenericClass.<Number>flatten(li);
```