Prof. Dr. A. Poetzsch-Heffter
Dipl.-Inform. Kathrin Geilmann

University of Kaiserslautern
Department of Computer Science
Software Technology Group

Advanced Aspects of Object-Oriented Programming (SS 2011)

Practice Sheet 6 (Hints and Comments)

Exercise 1 Questions

Exercise 2 Type Erasure

a) The erasure of types is defined in JLS 4.6. Applying this leads to the following code:

```
public class ComparableTest implements Comparable {
  public int compareTo(Object a) { return(-1); }
  public int compareTo(Object b) { return( 1); }

  public static void main(String args[]) {
    ComparableTest C=new ComparableTest();
  }
}
```

b) The erasures of the `compareTo`-methods are equal. The compiler cannot decide which method to call, i.e. he cannot write a call into the bytecode, even at runtime it would be impossible to decide which of the methods is the right one, because the generic information is not present anymore.

Exercise 3 Generics

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);
```

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/` for a version without generics and `http://larvalabs.com/collections/` for a generic version).

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) Predicate

```
public interface Predicate<A> {
    boolean evaluate(A o);
}
```

Iterator. Note, that you have to ensure that multiple subsequent calls to hasNext() do not modify the iterator.

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class FilteringIterator<E> implements Iterator<E> {
    public FilteringIterator(Iterator<E> inputIterator, Predicate<? super E> p) {
        in = inputIterator;
        this.p  = p;
    }

    private Predicate<? super E> p;
    private Iterator<E> in;

    private boolean nextSet = false; // the next object has been calculated
    private E nextObj = null; // the value that will be returned next by the iterator

    public boolean hasNext() {
        // If we already know the value, that will be returned by the next call to next()
```

```
        // then return true, i.e. multiple calls to hasNext() without intermediate
        // calls to next() will not move the cursor of the iterator this.in
        if (nextSet) {
            return true;
        } else
        return setNext();
    }

    public E next() {
        if (!nextSet) {
            if (!setNext()) {
                throw new NoSuchElementException();
            }
        }
        nextSet = false; // consume the current next object
        return nextObj;
    }

    // remove the object from the underlying collection, that has been returned last by the
    // next() method.
    // Note: this method cannot be called if hasNext has been called after the last next-call,
    // because hasNext() changes the underlying iterator.
    public void remove() {
        // easiest implementation, that matches the documentation of the interface Iterator<E>
        // return new UnsupportedOperationException();
        if (nextSet) {
            throw new IllegalStateException("remove() cannot be called");
        }
        in.remove();
    }

    // set nextObj to the next object, return true if successfull and false
    // if there are no more objects
    private boolean setNext() {
        while (in.hasNext()) {
            E n = in.next();
            if (p.evaluate(n)) {
                nextObj = n;
                nextSet = true;
                return true;
            }
        }
        return false;
    }
}
```

c)

```
public class PositivePredicate implements Predicate<Double> {
    public boolean evaluate(Double n) {
        return n.compareTo(0.0) >= 0; // value of n >= 0
    }
}

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