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 2015)

Practice Sheet 9 (Hints and Comments)

Exercise 1 Behavioral Subtyping

a) At the begin of method `getChar()` the state is valid (since this is the precondition of this method). This means that `0 <= lo && lo <= cur && cur <= hi`. Since we are going to increment cur, we have to make sure that `cur != hi`. If `cur == hi` we call `refill()` which may change `lo` and `hi` since it may change `state` and `state` depends on `lo` and `hi`. Method `refill()` may not change `valid`, so we know that `cur` is between `lo` and `hi` after returning from `refill`. If still `cur == hi` we can not proceed and return `-1`. Before incrementing `cur` we therefore know that `0 <= lo && lo <= cur && cur < hi`. After incrementing `cur` we have `0 <= lo && lo < cur && cur <= hi`. We access `buff` at index `cur-lo-1` which is `>= 0`. And since `hi-lo <= buff.length` we know that `cur-lo-1 < buff.length`.

b) File `StringReader.java`:

```
public interface StringReader extends Reader {

  /*@ public normal_behavior
    @   requires    s != null;
    @   assignable  valid, state;
    @   ensures     valid && \result == this;
    @*/
  public StringReader init(String s);
}
```

File `StringReaderImpl.java`:

```
public class StringReaderImpl extends BuffReader implements StringReader {
  private /*@ spec_public @*/ String str;
  //@ public represents svalid <- lo == 0 && hi == str.length() && buff.equals(str.toCharArray());

  public StringReader init(String s) {
    str = s;
    buff = str.toCharArray();
    lo = 0;
    cur = 0;
    hi = buff.length;
    return this;
  }

  /*@ also
    @   public normal_behavior
    @     requires    valid && cur < hi;
    @     assignable  state;
    @     ensures     \result == str.charAt(\old(cur));
    @ also
    @   public normal_behavior
    @     requires    valid && cur == hi;
    @     assignable  \nothing
    @     ensures     \result == -1;
    @ */
  public int getChar() {
    if(cur  == hi) {
      return -1;
    }
    cur++;
    return buff[cur-1];
  }

  public void refill() { }

  public void close(){ }

  //@ depends state  <- str;
  //@ depends svalid <- str;
}
```

c) In file `BuffReader.java`:

```
/*@ public normal_behavior
  @   requires    valid;
  @   requires    0 <= c && c <= 65535;
  @   requires    lo < cur;
  @   assignable  state;
  @   ensures     buff[cur-lo] == c;
  @*/
public void unread(int c){
  buff[cur-lo-1] = (char)c;
  cur--;
}
```

For the unread operation to work, at least one character has to be read from the reader.

The information about the inner workings of the buffered reader is not visible in `Reader`. Moving the method to the `Reader` interface and abstracting from the state would make the implementation much more complex.

The implementation of `StringReader` is no longer possible as given above, since it would only be valid to unread exactly the character that was previously read from the string, which would violate the specification of unread in `BuffReader`.

Exercise 2 Concurrent Access to Shared State

The read and write access to the field `stopRequested` is atomic, which means the field can only have the value **`true`** or **`false`**. This does not guarantee that there exists only a single instance of the field for all threads of the program. In the Java memory model, every thread has its own instance of this field. Synchronization of the value of such a field is only forced by entering a synchronized block or by writing to a volatile field. Since the example program neither uses synchronized blocks nor volatile fields, the value the main threads sets is never synchronized with the background thread.

The example can simply be fixed by declaring the field `stopRequested` as volatile.

Exercise 3 The Java Collections Framework and Thread Safety

a) An object (a component, a class,...) is thread-safe, if it behaves according to its specification when executed in a multi-threaded program context. A different possible definition is: An implementation is thread-safe if it is guaranteed to be free of race conditions when accessed by multiple threads simultaneously.

b) They synchronize on the wrapper object and delegate the execution of the calls to the wrapped object. This guarantees, that if all accesses to the wrapped collection are done via the wrapper, the collection is thread-safe. If wrapped the object is directly accessed, the behavior is unspecified.

c) The thread executing the code may get interrupted after the call to `containsKey`, another thread can now modify m and when the first thread continues, his assumption about the state of m is false. Solution: synchronize on m, for the parts where you need exclusive access to m.

```
Map<String,String> m = Collections.synchronizedMap(new HashMap<String,String>(...));
synchronized(m) {
   m.put("a", "b");
   if !(m.containsKey("a")) {
       m.put("c", "d");
   }
}
```

d) The ConcurrentHashMap allows simultaneous reads from the map, reading may also overlap writing, but it is ensured, that a read returns only results from writes that have been completed. The wrapper serializes all access to the underlying map. The `ConcurrentHashMap` needs less external synchronization in many cases, but in the same time, the results of read operations get even more unpredictable, additionally the behavior of iterators over the map changed.

e) Same problem as above. But now multiple thread can be concurrently in the Map implementation, in order to guarantee mutual exclusive access for a thread, we have to protect all possible concurrent accesses with a synchronize block.