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

Date of Issue: 16.06.15
Deadline: 23.06.15
(before the lecture as PDF via E-Mail)

Exercise 1 Behavioral Subtyping

In this exericse we look at the `Reader` example found at the end of slide set 5 (specification and checking).

a) Give a detailed explanation why the execution of method `BuffReader.getChar()` cannot lead to an `ArrayIndexOutOfBoundsException`.

b) Implement a `StringReader`. First add an interface equivalent to the `BlankReader` including the specification of the behavior. Refine the behavior of the `getChar()` method such that it returns the next character of the string if the position is not at the end and that it returns -1 if the reader is at the end of the string. Second add an implementation of `StringReader` that fulfills the specification.

c) Add a method `void unread(int c)` to `BuffReader`. The method pushes back a single character by copying it to the front of the buffer. After the method returns, the next character to be read is `c`. Add a specification and an implementation of the method in `BuffReader`. Is it easily possible to move the method to the `Reader` interface instead? How does this change influence the implementation of `StringReader`?

Exercise 2 Concurrent Access to Shared State

Read and write operations on fields are atomic except for fields of type `long` and `double` (§17.7). Analize the following code (taken and adapted from the book "Effective Java Second Edition" by Joshua Bloch):

```
package concurrency.mt;

import java.util.concurrent.TimeUnit;

public class StopThread {
  private static boolean stopRequested;

  public static void main(String[] args) throws InterruptedException {
    Thread backgroundThread = new Thread(new Runnable() {
      public void run() {
        int i = 0;
        while(!stopRequested) {
          i++;
        }
        System.out.println("Thread terminated.");
      }
    });
    backgroundThread.start();

    TimeUnit.SECONDS.sleep(1);
    stopRequested = true;
    System.out.println("Program sould terminate now...");
  }
}
```

The expected behavior is that the program terminates after one second by telling the background thread to stop working. On some machines, this program does not terminate. Explain this unintended behavior. How can the program be fixed?

Exercise 3 The Java Collections Framework and Thread Safety

a) Explain the term “thread-safe”.

b) Inform yourself about the synchronization wrappers for the collection classes of `java.util`. How do they work and which guarantees are given?

c) Are there problems with the following code, if it is executed in a multi-threaded program context? If yes, explain and fix the issues.

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

d) Java 5 introduced the package `java.util.concurrent`, that contains classes to support common tasks in multi-threaded programs. Consider the class `ConcurrentHashMap`. What is the difference between a `ConcurrentHashMap` and a synchronized Map as returned by the synchronization wrapper? Which are the (dis-)advantages of using a `ConcurrentHashMap` over a synchronized Map, and vice versa?

e) Are there problems with the following code, if it is executed in a multi-threaded program context? If yes, explain and fix the issues.

```
ConcurrentHashMap<String,String> m = new ConcurrentHashMap<String,String>(...);
m.put("a", "b");
if !(m.containsKey("a")) {
    m.put("c", "d");
}
```