FACULTY OF INFORMATICS

# ADVANCED JAVA PROGRAMMING

Assignment 3

Author FEDERICO LAGRASTA

CLAUDIO MAGGIONI

November 6, 2022

# A

## 1

The code will trigger a compile time error at line 14, since the variable $z$ cannot be used inside a lambda as it is not final or effectively final. This is because line 14 attempts to mutate the variable, thus making the variable not effectively final.

## 2

The code will not compile.

# B

## 1

According to the contract satisfied by the `java.util.HashSet` class, there is no guarantee on the iteration order of the elements. Therefore, `set.stream()` evaluates to a stream whose ordering contractually does not reflect the order of the list. However, by analyzing the Java 17 standard library sources, given that:

- The elements of the source list are *Integer*s and are already in natural order;
- *HashSet* uses a *HashMap* by encapsulation to store its elements;
- The *Spliterator* returned by *HashSet* is the *Spliterator* over the keys of the backing *HashHap*;
- *HashMap*'s spliterator implementation iterates over the key objects in the natural order of each object's hash code;
- *Integer*'s `int hashCode()` implementation just returns the boxed *int*;

we can say that on any JVM this code is ran where the implementation of the standard library matches the sources provided with OpenJDK 17, we can say that in practice the *Spliterator* backing the stream returned by `set.stream()` returns the elements in the correct order.

However, the `java.util.stream` package documentation[1], under the *Ordering* section, explicitly mentions that the *Stream* returned by an *HashSet* is unordered. Therefore, also by the Stream API contract, we can say that the collection of this stream may result into a list that contractually is not in the order the elements were when creating the stream. However, in practice, since the created stream is sequential, the elements are kept in the given order as there is no performance advantage (unlike with parallel streams) in unordering them. Therefore, according to this contract, the method may throw a *RuntimeException* even if this never happens in practice with any JVM with a standard library implementation matching the sources provided with OpenJDK 17.

## 2

What said in the previous section still applies, however in this case line 31 makes the stream parallel, and since the stream is contractually unordered there is no guarantee the elements may be collected in order since the stream API implementation may not follow the original ordering for parallelization performance reasons.

[1] `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/package-summary.html#Ordering`