Formal Methods in Software Development
Exercise 1 (October 14)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

September 18, 2015

The result is to be submitted by the deadline stated above *via the Moodle interface* of the course as a *.zip or .tgz* file which contains

1. a PDF file with
   - a cover page with the course title, your name and email address,
   - the deliverables requested in the description of the exercise,
2. the JML-annotated Java files developed in the exercise.

Email submissions are *not* accepted.

Exercise 7: Checking JML Specifications

Annotate each method given in the files `Exercise1a.java` and `Exercise1b.java` by JML specifications the JML *heavy-weight* format using by a precondition (`requires`), frame condition (`assignable`), and postcondition (`ensures`). Additionally, give each class a `main` function that allows you to test the implementation by calls of the corresponding method.

Make preconditions as *weak* as possible; e.g. if the method can be reasonably applied to argument 0, do not require that the argument needs to be positive. Make postconditions as *strong* as possible; e.g. if a result is always positive, do not just ensure that the result is non-negative. Also do not forget to explicitly specify the null/non-null status and the lengths of arrays.

For each method, first use `javac` to compile the program (to make sure that it is syntactically correct and has no type errors) and then use `jml` to type-check the specification.

Second, use the runtime assertion compiler `jmlc` and the executor `jmlrac` to validate the specification respectively implementation by at least three calls of each method; the calls shall contain at least two different valid inputs and (if possible) also one invalid input (for arrays, use arrays with wrong length or content, not just null pointers). Please print after each method call some output to make sure that the method has not silently crashed. You may also try the alternative tool set `openjmlrac/openjmlrun`; please report your experience with this. If you detect that the runtime assertion compiler fails for some part of the specification, you may comment it out as an informal property `(* ...*)` and repeat the check with the simplified specification.

Third, use the extended static checker `escjava2` to further validate the code; use the option `-NoCautions` to suppress any cautions you may get from system libraries. You may also try the alternative extended static checker `openjmlesc` (please report your experience).

The deliverables of this exercise consist of

- a nicely formatted copy of the JML-annotated Java code for each class,
- the output of running `jml -Q` on the class,
- the output(s) of running `jmlrac`/`openjmlrun` on the class,
- the output of running `escjava2`/`openjmlesc` on the class.

both for the original and for the modified implementation of the method (if the implementation was modified) including an explanation of the detected error and how you fixed it.

Please note that the fact that `escjava2`/`openjmlesc` does not give a warning does not prove that the function indeed satisfies the specification (only that the tool could not find a violation); on the other hand, if the checker reports a warning, this does not necessarily mean that the program indeed violates its specification (only that the tool could not verify its correctness).

Recommendation: it is better to split pre/post-conditions that form conjunctions into multiple `requires` respectively `ensure` clauses (one for each formula of the conjunction); if an error is reported, it is then clear, to which formula it refers.