Formal Methods in Software Development
Exercise 7 (December 17)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

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, Matrikelnummer, and email address,
   - a section for each part of the exercise with the requested deliverables and optionally any explanations or comments you would like to make;
2. the JML-annotated `.java` file(s) used in the exercise.

Email submissions are *not* accepted.

Exercise 7: JML Specifications

Formalize the method specifications given below in the JML *heavy-weight* format by a precondition (`requires`), frame condition (`assignable`), and postcondition (`ensures`) and attach the specification to the method implementations provided in file `Exercise7.java`. For this purpose, extract the implementation of each method into a separate class `Exercise7_`*I* (where *I* is the number of the method in the list below) and give this class a `main` function that allows you to test the implementation by a call 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 `jml` to type-check the specification[1]. Then use the runtime assertion compiler `jmlc` and the corresponding 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. Also try the alternative more modern tool set `openjmlrac/openjmlrun` and report your experience with this. If you detect that the runtime assertion compiler fails for some part of the specification, you may comment this part out as an informal property `(* ...*)` and repeat the check with the simplified specification.

Second, use the extended static checker `escjava2` to further validate the functions; you may use the option `-NoCautions` to suppress any cautions you may get from system libraries. Also try the alternative extended static checker `openjmlesc` and report your experience with this. Before checking, comment out the `main` functions such that only the specified functions are checked.

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 outputs of running `jmlrac` and `openjmlrun` on the class,
- the outputs of running `escjava2` and `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).

[1] The old JML toolsuite tools `jml/jmlc/jmlrac` do not work with Java 8 which is the default on the course virtual machine. Therefore, before starting these tools from a terminal, execute `PATH=/software/java/bin:$PATH` to make an older Java version the default. Be sure to subsequently not use this terminal to run any other Java programs, in particular not the new OpenJML tools `openjmlrac/openjmlrun` which require Java 8.

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.

1. Specify the method

   ```
   public static int minimumPosition(int[] a)
   ```

   that takes an integer array a and returns the position of the smallest element in the array.

2. Specify the method

   ```
   public static int minimumElement(int[] a)
   ```

   that takes an integer array a and returns the smallest element in the array.

3. Specify the method

   ```
   public static int maximumElement(int[] a)
   ```

   that takes an integer array a and returns the greatest element in the array.

4. Specify the method

   ```
   public static int[] overwrite(int[] a, int p, int n, int x)
   ```

   that returns a new array that is identical to a except that n elements starting at index p have been overwritten by x (the resulting array may be longer than b to accommodate the indices $p, \ldots, p + n - 1$).

5. Specify the method

   ```
   public static int replace(char[] a, char x, char y)
   ```

   that takes a character array a and replaces in it every character x by y. The return value of the function indicates the smallest position where a replacement has been performed (-1, if no replacement has been performed).

6. Specify the method

   ```
   public static boolean add1(int[] a, int[] b)
   ```

   that takes two arrays a and b that hold non-negative integers and adds to every element of a the corresponding element of b unless this would result in an overflow (i.e., unless one element is bigger than the difference of `Integer.MAX_VALUE` and the other element); in that case the value is set to `Integer.MAX_VALUE`. The return value of the function indicates whether such an "overflow" has occurred.

   Hint: you should rule out that a caller passes as a and b the *same* array (why?).

7. Specify the method

   ```
   public static void add2(int[] a, int[] b) throws Truncated
   ```

that behaves like `add1`, except that at the first occurrence of an "overflow" an exception is thrown that contains the position of the overflow; from that position on all elements of *a* remain unchanged.

Hint: `jmlc` complains about the use of `e.pos` in the specification of `Truncated`, you may comment out the corresponding specification clause. Furthermore, you may ignore the warning of `escjava2` about the possible violation of a `modifies` clause of class `Overflow`; this is due to an underspecification of the superclass `Exception`.

Please note that the given informal specifications may be too weak (e.g., preconditions may be missing) or ambiguous (but not plainly wrong) and that the implementations may be incorrect. If you detect problems, explain them, fix them such that specification and code match and re-run your checks (please apply common sense and consier the probable intention of the developer/client in order to decide how to complete the specification and/or fix the implementation).