Formal Methods in Software Development
Exercise 5 (December 20)

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 5: 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 `Exercise5.java`. For this purpose, extract the implementation of each method into a separate class `Exercise5_`*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. Please also note that JML has some restrictions about which values may be used in `\old(...)` expressions; e.g., `\result` may not be used there. You can, however, write the following instead:

```
(\exists int r; r == \result; ... \old(...r...) ...)
```

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. 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.

Also try the more modern OpenJML tool set with the corresponding commands `openjml`, `openjmlrac`, and `openjmlrun` and report your experience with these[2].

Finally, use the extended static checker `escjava2` to further validate the functions[3]; you may use the option `-NoCautions` to suppress any cautions you may get from system libraries. 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` and `openjml` on the class,
- the outputs of running `jmlrac` and `openjmlrun` on the class,
- the outputs of running `escjava2` on the class.

[1] The JML toolsuite only work with Java 5; on the course VM command aliases select this version.
[2] The OpenJML toolset only works with Java 8; on the course VM the command scripts select this version.
[3] ESC/Java2 only works with Java 5; on the course VM the command script selects this version.

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` 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.

1. Specify the method

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

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

2. Specify the method

   ```
   public static int maximumElement1(int[] a)
   ```

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

3. Specify the method

   ```
   public static int maximumElement2(int[] a)
   ```

   that takes an integer array a of non-negative integers and returns the largest element in a.

4. Specify the method

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

   that returns a new array that contains the elements of a with n copies of value x inserted at position p (elements may be also inserted at at the beginning or at the end of a).

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 remains unchanged. 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 Overflow
   ```

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

   Also `escjava2` has problems with reasoning about `\old(a[e.pos])`; use the trick explained above of introducing an existentially quantified variable `r` whose value is `e.pos` and write `\old(e[r])` instead.

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).

Also please note that various tools have restrictions with respect to their support of JML respectively to their capabilities of reasoning about JML. If you encounter some problems apparently related to such restrictions, document them and try to find some workaround (e.g., by commenting out problematic parts of the specification).