Assignment 3:
Extended Java Typechecking

Software Analysis

Due date: 2023-05-02 at 23:00

1. The assignment

Your assignment in a nutshell:

1. Pick a software **project** written in Java and consisting of at least 1'000 lines of code. The source code should make significant *usage of arrays, strings, or both*. The project can be:
   - an open-source project available on GitHub or other public repositories,
   - a package or component of the standard Java API,
   - a project you developed in the past (for example an assignment or course project), or
   - any other project whose source code is available and that you can share.
2. Add the necessary *annotations* to the project so that the Checker Framework's **index checker**[1] is able to rule out any **out-of-bound accesses** in the code.
3. Write a short **report** discussing your work.
   Maximum length of the report: 8 pages (A4 with readable formatting).

The assignment must be done *individually*.
This assignment contributes to **25%** of your overall grade in the course.

[1] See https://checkerframework.org/manual/#index-checker

2. Tools and documentation

The Checker Framework is available, together with Java 11 and Java 17 OpenJDK compilers, in the Docker image `bugcounting/satools:y23`. Otherwise, instructions to install the Checker Framework are available on the project's website:

`https://checkerframework.org`

The same page includes detailed documentation of the framework as a whole, as well as of each checker that is available:

`https://checkerframework.org/manual/`

Make sure to read at least the tutorial, and the documentation for the *index checker*. See also Appendix A in this assignment description for some help to configure the Checker Framework on your project.

2.1. Plagiarism policy

You are allowed to learn from any examples that you find useful; however, you are required to:

1. write down the solution completely on your own; and,
2. if there is a publicly available example that you especially drew inspiration from, credit it in the report (explaining what is similar and how your solution differ).

Failure to do so will be considered plagiarism. (If you have doubts about the application of these rules, ask the instructors *before submitting* your solution.)

2.1.1. ChatGPT & Co.

The plagiarism policy also applies to AI tools such as ChatGPT or CoPilot:

1. You are allowed to use the help of such tools; however, you remain entirely responsible for the solution that you submit.
2. If you use any such tools, you must add a section to the report that mentions which tools you used and for what tasks, how you checked the correctness and completeness of their suggestions, and what modifications (if any) you introduced on top of the tool's output.
3. If you use a text-based tool such as ChatGPT, also show a couple of examples of prompts that you provided, with a summary of the tool's response.

Failure to abide by these rules, including failing to disclose using AI tools, will be considered plagiarism.

3. Tips and suggestions

3.1. Checkers

The Checker Framework is a collection of plugins, each targeting a different set of extended type annotations. In this assignment you should focus on the *index checker*, which supports annotations to specify the "safety" of indexes used to access arrays or strings. If you would like to explore the capabilities of a different checker available within the Checker Framework – for example because it matches better the characteristics of the project you are analyzing – you can use that *in addition to* the index checker. In this case, you may also want to check with us (the instructors) before doing so, so that we can advise you about whether your choice of checker is appropriate.

3.2. Choosing the project

Besides its size, the main requirement of the project is that it should perform some indexed access to arrays or strings data, so that the annotations of the index checker can be meaningfully used to check absence of out of bound errors.

Here are a few examples of suitable projects used by students of previous years. You may want to look for projects of similar size, complexity, and characteristics. As usual, you should do your own analysis independent of the other students'.

- `https://github.com/zxing/zxing`
- `https://github.com/Simmetrics/simmetrics`
- `https://github.com/TheAlgorithms/Java`
- `https://github.com/vbohush/SortingAlgorithmAnimations`
- `https://github.com/phishman3579/java-algorithms-implementation`

3.3. Warnings

The Checker Framework may generate a large number of warnings for a project. In this case, you do not have to address all of them but only up to **30 warnings** in the whole project. If you only address a subset of all warnings reported by the Checker Framework, discuss in the report how you selected the warnings to address, and motivate your selection.

3.4. Annotations

Checkers require "specification" annotations to perform any meaningful analysis; some annotations are implicitly assumed whereas others have to be written out explicitly. In particular, the index checker may miss some checks unless you annotate all variables that are used by the program as indexes of some arrays or strings. Obviously, you cannot make your project trivially pass checks by simply not adding any meaningful annotations. Instead, start by adding

some basic annotations that encode the expected properties of the main methods and fields. Then, running the checker will report errors, which you can remove by adding more type annotations in a way that documents the properties of intermediate values in the overall computation.

Try to write annotations *consistently* within each piece of code that implements a certain functionality. For example, if a method processes an array `a`, make sure to annotate *all* variables that are used as indexes of `a` within the method's body. If you only annotate some and ignore others, the checker will lack necessary information to correctly reason about the method's array accesses correctness, which typically leads to reporting many spurious warnings.

For the index checker, note that annotations about an array's *content* go between the array's base type and the square brackets. For example, `int @ArrayLen(7) [] a` declares an integer array `a` of length 7. In contrast, annotations about *each individual element* of the array (that is, about the array's base type) go before the base type itself. For example, `@Positive int [] b` declares an array `b` of positive integers. Of course, you can combine both kinds of annotations – about an array and its base type – in the same declaration.

3.5. Modifying the implementation

As much as possible, you should not change the *implementation* of your project but only add type annotations for the Checker Framework to work. This is in contrast with what you did for Assignment 2, where you had to modify the implementation to fix true positive warnings; in this Assignment 3, you have to add type annotations to reflect actual program behavior.

Remember that type checking has only the limited information about the program state that is expressed by means of type annotations, which limits its reasoning capabilities about program behavior. It's quite possible that, also due to limitations of the Checker Framework, you will not be able to express all necessary annotations to make type checking succeed in all cases even if the program is actually correct (i.e., it does not incur any out of bound errors). One of the goals of this exercise is to make you understand the extended type checking's limitations, and how to work around them.

Sometimes, you may have to add `@SuppressWarning` annotations to bypass typechecking errors in parts of the code where the type correctness properties are first established. Use `@SuppressWarning` judiciously and only when it is strictly necessary; and add a comment next to each `@SuppressWarning` explaining why you had to use that annotation.

An alternative to `@SuppressWarning` is adding an `assert(P)` with an `@AssumeAssertion` annotation in the `assert`'s message. In this case, the asserted predicate `P` should characterize the information about the program state at that location, in a way that justifies why the following type annotation is correct; correspondingly, `@AssumeAssertion` tells the Checker Framework to assume `P` and use that information to discharge constraints of the given checker (for example `@AssumeAssertion(nullness)` for the nullness checker).

As a last resort, you may have to modify the implementation to be able to successfully pass the checks. Where and how you used `@SuppressWarning` and `@AssumeAssertion` – and where and how you modified the implementation – are interesting aspects to discuss in the report.

3.6. What to write in the report

Topics that can be discussed in the report include:

- The choice of project – in particular, why it is a good candidate to apply the index checker.
- If you used a checker other than the index checker, why you selected it.
- A high-level description of the process you followed to add type annotations to the project.
- How many type annotations you included, and what kinds of properties they specify.
- Which features (if any) required a `@SuppressWarning` and why.
- Which features (if any) required to modify the implementation and why.
- Did using the checker help you find any bugs or other questionable design and implementation choices?
- How complex was it to apply the checker, and what benefits did you gain in return?
- Compare the checker's trade-off between complexity of usage and analysis power to that of other software analysis techniques you're familiar with (in particular, those used in previous assignments).

4. How and what to turn in

Turn in:

1. The following artifacts in a project named `Assignment3` in your assigned GitLab project for Software Analysis.[2]
   a) Your **annotated project** (including all annotations necessary to pass the Checker-Framework's index checker analysis)
   b) A shell **script** or **build configuration** file (such as a Maven `pom.xml` file, or just a simple `Makefile`) that compiles the project with all dependencies, and runs the Checker Framework's index checker on it

   The scripts can assume that the Checker Framework, a Java 11 and/or Java 17 JDK compiler, and Maven and/or Gradle are available as in the environment provided by the Docker image `bugcounting/satools:y23`; any other dependency must be included or pulled by the build script. Make sure the build process works without problems: if it does not run effortlessly, your submission may not be accepted or lose points.
2. The **report** in PDF format as a single file using *iCorsi* under *Assignment 3*.

[2] The same project you used for the previous assignments; see details in Assignment 1's description.

A. Installing and Using the Checker Framework

The rest of this section gives some instructions on how to run the Checker Framework with common tools, and includes specific pointers to the official documentation.

A.1. Modules

If your project uses *modules* (which have been available since Java 9), the source code will include a file `module-info.java`. To make the modules of the Checker Framework and your project's be able to access each other, add the line `requires org.checkerframework.checker.qual;` to the project's `module-info.java`.

A.2. Using the Command Line

If you are compiling your project directly from the command line, the simplest way to use the Checker Framework is through its wrapper script of the Java compiler, which is available as command `javacheck` in the Docker image `bugcounting/satools:y23`. To replicate that behavior in your installation see the instructions at https://checkerframework.org/manual/#javac-wrapper.

To use the **index checker** from the command line, you pass option `-processor index` to `javacheck`.

The Docker image uses `javac` and `java` of Java 11 by default. To use Java 17 instead, you can run the command `java_version 17` (`java_version 11` switches back to Java 11).

A.3. Using Maven

If your project compiles using Maven, follow the instructions at: https://checkerframework.org/manual/#maven. Be aware that those instructions mix up instructions for different Java versions. If you are using Java 11 or later, you can skip step 2 in those instructions.

Also note that those instructions enable the nullness checker. To enable the index checker instead, replace:

```
<annotationProcessor>org.checkerframework.checker.nullness.NullnessChecker</annotationProcessor>
```

in the profile definitions in step 3 with the following:

```
<annotationProcessor>org.checkerframework.checker.index.IndexChecker</annotationProcessor>
```

A complete `pom.xml` using these build directives is available here in the Checker Framework's source code.

A.4. Using IntelliJ

If your project compiles with Maven, you don't need to configure anything specific in IntelliJ since you will just call Maven for compilation. If you want to compile your project directly within IntelliJ (without passing through Maven), follow the instructions at https://checkerframework.org/manual/#intellij.

A.5. Importing annotations

The typing annotations of each checker you use in your project must be imported in every class where they are actually used. For example, to use the index checker's annotation `@IndexFor` import it with:

```
import org.checkerframework.checker.index.qual.IndexFor;
```

The fully qualified name of annotations is available in the API documentation.