CSSE4630 Week 2 Lab: Nullness Checkers

Mark Utting

Version 1.1

1 Introduction

You have all used simple type systems, in Java and other languages. They are good for catching certain kinds of errors before you even run the program. That is, they are *static analysis tools*.

In this lab, we want to try out a type-checker system that goes a little further, and can guarantee that your program has **no null-pointer errors**! We don't need to invent or learn a completely new language to do this, since the Java compiler supports *pluggable type -checkers*.

We will experiment with using the Nullness Checker from the open-source project `https://checkerframework.org`. Here are some useful links:

- Manual: `https://checkerframework.org/manual` — See Chapter 3 for the Nullness Checker.
- Tutorial: `https://checkerframework.org/tutorial` — focus on Step 3 (Getting Started), which is about the Nullness Checker.
- Alternative: if you cannot get it installed on your computer, you could try small examples in the online Live Demo page `http://eisop.uwaterloo.ca/live`.

Here are the annotations I suggest you will find most helpful:

- **@NonNull** = this variable can never be null. (this is the default, so you usually do not need to add it explicitly).
- **@Nullable** = this variable can be null, or non-null.
- **@MonotonicNonNull** = this variable can start off null, but once it becomes non-null, it NEVER goes back to null again.
- **@EnsuresNonNull("field")** = if you annote a method with this annotation, it means that **this.field** is guaranteed to be non-null just after that method returns.

2 Analysing the Garden Planner

1. Download the **GardenPlanner.zip** file from Blackboard, and unzip it. This includes (pre-installed) the Checker Framework from `https://checkerframework.org`.
2. Open the project with IntelliJ IDEA (Open Project), build the `garden_planner2` module, then open `src/garden_planner/textui/TextUI.java` and run the 'main' method. This should print some textual output like the following, giving the costings of a default garden design.

```
Garden Planner v0.2
Garden design is:
    Rectangle 1.00 1.00 1.00 2.00
    Rectangle 3.00 1.00 2.00 2.00
    Rectangle 6.00 1.00 1.00 2.00
Total garden area is:     8.00 m2.
Total wall length is:    20.00 m.
Total soil required:      1.60 m3.
Total garden cost is: $ 469.60.
```

3. The garden designs and costing calculations are managed by the `garden_planner/model/GardenPlanner.` class, so we will focus on analysing this 'model' class in this lab, to see if it is safe from Null-Pointer Errors, or to make it safer, if necessary. The recommended 'Checker.org' philosophy is:

   > Before you run a checker, annotate the code, based on its documentation. Then, run the checker to uncover bugs in the code or the documentation.
   >
   > Don't do the opposite, which is to run the checker and then add annotations according to the warnings issued. This approach is less systematic, so you may overlook some annotations. It often leads to confusion and poor results. It leads users to make changes not for any principled reason, but to "make the type-checker happy", even when the changes are in conflict with the documentation or the code. Also see "Annotations are a specification", below [in Section 2.4.2 of the manual].

So start by reading the code of the main **GardenPlanner** class, in the **garden_planner/model** package. Decide where you need to add some **@Nullable** annotations.

Note 1: the default assumption of the Nullness Checker is that every variable, every parameter, every function result etc., is automatically annotated with **@NonNull** unless you explicitly put an **@Nullable** annotation there instead. So you will need to add a **@Nullable** annotation everywhere you think a pointer should be allowed to be null.

Note 2: you will need to import each annotation that you use. For example:

```
import org.checkerframework.checker.nullness.qual.Nullable;
```

4. Now run the Nullness Checker. For example, run the following command lines, in the **src/** directory (you may need to adjust these for your machine — e.g. use ';' as the classpath separator on Windows instead of ':').

```
CLASSPATH=".:../../checker-framework-3.5.0/checker/dist/checker.jar"
javac -cp "$CLASSPATH"  \
    -processor org.checkerframework.checker.nullness.NullnessChecker \
    -J--add-opens=jdk.compiler/com.sun.tools.javac.comp=ALL-UNNAMED \
    garden_planner/model/RectBed.java \
    garden_planner/model/GardenPlanner.java \
    garden_planner/textui/TextUI.java
```

Alternatively, in IntelliJ you can add the Nullness checker as an 'annotation processor' during the project build. To do this, go into **File / Settings / Build, Execution, Deployment / Compiler / Annotation Processors** and change two settings:

(a) Turn on 'Enable annotation processing'.

(b) Add **org.checkerframework.checker.nullness.NullnessChecker** into the list of annotation processors.

Then click OK to save the settings, and use **Build / Rebuild garden_planner** to force a recompile of all the Java source files.

5. How many errors do you get? Discuss different ways of fixing the errors with your neighbour or tutor. Add annotations or remove them until you get no Nullness Checker errors.

   Note: Feel free to experiment with changing the behaviour of the GardenPlanner class a little. For example, does it help to initialise some of the fields to non-null values in the constructor, so that they are never null? Think about how this will change the experience of programmers who are using GardenPlanner — will it make the API easier to use or harder? More error-prone, or less error-prone?

   For example, GardenPlanner is safer from null errors if callers always remember to call either createBasicDesign or readBeds before using other features. But can you make it more robust, so that it is *always* safe?

3 Analysing your Own Code

Choose a Java project that you've written or contributed to in the last year or two (e.g. one of your COMP4403 projects?). Try running the Nullness Checker on a few of the key classes in that project, and see how useful the error messages are. How many potential null-pointer problems were there in your code?