The Checker Framework Manual

http://pag.csail.mit.edu/jsr308/

Version 0.9.6 (29 Jul 2009)

For the impatient: Section 1.2 describes how to **install and use** pluggable type-checkers.

Contents

1 Introduction **4**
- 1.1 How it works: Pluggable types . . . 4
- 1.2 Installation . . . 4
  - 1.2.1 Unix/Linux/MacOS installation . . . 4
  - 1.2.2 Windows installation . . . 5
- 1.3 Example use: detecting a null pointer bug . . . 6

2 Using a checker **7**
- 2.1 Writing annotations . . . 7
  - 2.1.1 Distributing your annotated project . . . 7
- 2.2 Running a checker . . . 7
  - 2.2.1 Checker auto-discovery . . . 8
  - 2.2.2 Ant task . . . 8
  - 2.2.3 Maven plugin . . . 9
  - 2.2.4 Eclipse . . . 9
  - 2.2.5 tIDE . . . 9
- 2.3 Checking partially-annotated programs: handling unannotated code . . . 9
- 2.4 Suppressing warnings . . . 10
- 2.5 Polymorphism and generics . . . 11
  - 2.5.1 Generics (parametric polymorphism or type polymorphism) . . . 11
  - 2.5.2 Qualifier polymorphism . . . 11
- 2.6 Unused fields and dependent types . . . 12
  - 2.6.1 Unused fields . . . 12
  - 2.6.2 Dependent types . . . 13
  - 2.6.3 Example . . . 13
- 2.7 The effective qualifier on a type (defaults and inference) . . . 14
  - 2.7.1 Default qualifier for unannotated types . . . 14
  - 2.7.2 Automatic type refinement (flow-sensitive type qualifier inference) . . . 15
- 2.8 What the checker guarantees . . . 16
- 2.9 Writing annotations in comments for backward compatibility . . . 17
  - 2.9.1 Annotations in comments . . . 17
  - 2.9.2 Import statements . . . 17
  - 2.9.3 Migrating away from annotations in comments . . . 17
- 2.10 Tips about writing annotations . . . 18
  - 2.10.1 Annotations on constructor invocations . . . 18
  - 2.10.2 Annotations indicate normal behavior . . . 18

2.10.3 Annotations indicate a contract . . . 18

3 Nullness checker **19**
3.1 Nullness annotations . . . 19
3.1.1 `@Raw` annotation for partially-initialized objects . . . 20
3.2 Writing nullness annotations . . . 21
3.2.1 Implicit qualifiers . . . 21
3.2.2 Default annotation . . . 21
3.2.3 Inference of `@NonNull` and `@Nullable` annotations . . . 21
3.3 What the Nullness checker checks . . . 22
3.3.1 Suppressing warnings with assertions . . . 22
3.4 Examples . . . 22
3.4.1 Tiny examples . . . 22
3.4.2 Annotated library . . . 23
3.5 Other tools for nullness checking . . . 23
3.5.1 Which tool is right for you? . . . 23
3.5.2 Compatibility note about FindBugs `@Nullable` . . . 24

4 Interning checker **24**
4.1 Interning annotations . . . 25
4.2 Annotating your code with `@Interned` . . . 25
4.2.1 Implicit qualifiers . . . 25
4.3 What the Interning checker checks . . . 26
4.4 Examples . . . 26

5 IGJ checker **26**
5.1 IGJ and Mutability . . . 26
5.2 IGJ Annotations . . . 27
5.3 What the IGJ checker checks . . . 27
5.4 Implicit qualifiers . . . 27
5.5 Annotation IGJ Dialect . . . 28
5.5.1 Semantic Changes . . . 28
5.5.2 Syntax Changes . . . 28
5.5.3 Templating Over Immutability: `@I` . . . 28
5.6 Examples . . . 29

6 Javari checker **29**
6.1 Javari annotations . . . 30
6.2 Writing Javari annotations . . . 30
6.2.1 Implicit qualifiers . . . 30
6.2.2 Inference of Javari annotations . . . 30
6.3 What the Javari checker checks . . . 30
6.4 Examples . . . 30

7 Lock checker **31**
7.1 Lock annotations . . . 31
7.1.1 Relationship to annotations in *Java Concurrency in Practice* . . . 31

8 Basic checker **32**
8.1 Using the Basic checker . . . 32
8.2 Basic checker example . . . 32

9 Annotating libraries **33**
9.1 Using stub classes . . . 33
9.1.1 Creating a stub file . . . 34
9.1.2 Using a stub file . . . 34
9.1.3 Stub file format . . . 34
9.1.4 Known problems . . . 35
9.2 Using skeleton files (distributed annotated JDKs) . . . 35

10 How to create a new checker **35**
10.1 The parts of a checker . . . 36
10.2 Annotations: Type qualifiers and hierarchy . . . 36
10.2.1 Declaratively defining the qualifier and type hierarchy . . . 36
10.2.2 Procedurally defining the qualifier and type hierarchy . . . 37
10.2.3 Defining a default annotation . . . 37
10.2.4 Bottom qualifier . . . 38
10.3 Type Factory: Implicit annotations . . . 38
10.3.1 Declaratively specifying implicit annotations . . . 38
10.3.2 Procedurally specifying implicit annotations . . . 39
10.4 Visitor: Type rules . . . 39
10.5 The checker class: Compiler interface . . . 40
10.5.1 Bundling multiple checkers . . . 40
10.6 Testing framework . . . 41
10.7 Debugging options . . . 41
10.8 javac implementation survival guide . . . 41
10.9 When to use (and not use) type qualifiers . . . 42

11 Troubleshooting and getting help **43**
11.1 Common problems and solutions . . . 43
11.1.1 Known problems in the framework . . . 44
11.1.2 Known problems in the Nullness checker . . . 44
11.2 How to report problems . . . 44
11.3 Installing the source release . . . 44
11.3.1 The short instructions (for Linux only) . . . 45
11.3.2 The longer instructions . . . 45
11.3.3 Building from source . . . 45
11.4 Learning more . . . 46
11.5 Comparison to other tools . . . 46
11.6 Credits and changelog . . . 46

1 Introduction

The Checker Framework enhances Java's type system to make it more powerful and useful. This lets software developers detect and prevent errors in their Java programs.

The Checker Framework comes with 4 checkers for specific types of errors:

1. Nullness checker for null pointer errors (see Section 3)
2. Interning checker for errors in equality testing and interning (see Section 4)
3. IGJ checker for mutation errors (incorrect side effects), based on the IGJ type system (see Section 5)
4. Javari checker for mutation errors (incorrect side effects), based on the Javari type system (see Section 6)

These checkers are easy to use and are invoked as arguments to `javac`.

The Checker Framework also enables you to write new checkers of your own; see Sections 8 and 10.

1.1 How it works: Pluggable types

The Checker Framework supports adding pluggable type systems to the Java language in a backward-compatible way. Java's built-in typechecker finds and prevents many errors — but it doesn't find and prevent *enough* errors. The Checker Framework lets you run an additional typechecker as a plug-in to the javac compiler. Your code stays completely backward-compatible: your code compiles with any Java compiler, it runs on any JVM, and your coworkers don't have to use the enhanced type system if they don't want to. You can check only part of your program, and type inference tools exist to help you annotate your code.

A type system designer uses the Checker Framework to define type qualifiers and their semantics, and a compiler plug-in (a "checker") enforces the semantics. Programmers can write the type qualifiers in their programs and use the plug-in to detect or prevent errors. The Checker Framework is useful both to programmers who wish to write error-free code, and to type system designers who wish to evaluate and deploy their type systems.

This document uses the terms "checker", "checker plugin", "type-checking compiler plugin", and "annotation processor" as synonyms.

1.2 Installation

This section describes how to install the binary release of the Checker Framework. The binary release contains everything that you need, both to run checkers and to write your own checkers. As an alternative, the source release (Section 11.3) is useful if you wish to examine or modify the implementation of checkers or of the framework itself.

Requirement: You must have **JDK 6** or later installed. You can get JDK 6 from Sun or elsewhere. If you are using Apple Mac OS X, you can either use Apple's implementation or SoyLatte.

For Unix/Linux/MacOS installation instructions, see Section 1.2.1. For Windows installation instructions, see Section 1.2.2.

1.2.1 Unix/Linux/MacOS installation

These instructions assume that you use the bash or sh shell. If you use a different shell, you may need to slightly adjust the commands.

1. Download the latest Checker Framework distribution and unzip it. You can put it anywhere you like; a standard place is in a new directory named `jsr308`.

```
export JSR308=$HOME/jsr308
mkdir ${JSR308}
cd ${JSR308}
wget http://groups.csail.mit.edu/pag/jsr308/current/jsr308-checkers.zip
unzip jsr308-checkers.zip
```

2. The download includes an updated version of the javac compiler, called the "Type Annotations compiler" or "JSR 308 compiler", that will be shipped with Java 7. In order to use the updated compiler when you type `javac`, add the directory `.../checkers/binary` to your path.
   Place the following commands in your `.bashrc` file (and also execute it on the command line, or log out and back in):

   ```
   export JSR308=$HOME/jsr308
   export PATH=$JSR308/checkers/binary:${PATH}
   ```

3. Verify that the installation works. From the command line, run:

   ```
   javac -version
   ```

   The output should be:

   ```
   javac 1.7.0-jsr308-0.9.6
   ```

That's all there is to it! Now you are ready to start using the checkers.

Section 1.3 walks you through a simple example. More detailed instructions for using a checker appear in Section 2.

1.2.2 Windows installation

1. Download the latest Checker Framework distribution and unzip it to create a `checkers` directory. You can put it anywhere you like; a standard place is in a new directory under `C:\Program Files`.
   (a) Save the file `http://groups.csail.mit.edu/pag/jsr308/current/jsr308-checkers.zip` to your Desktop.
   (b) Double-click the `jsr308-checkers.zip` file on your computer. Click on the `checkers` directory, then Select `Extract all files`, and use `C:\Program Files` as the destination. You will obtain a new `C:\Program Files\checkers` folder.
2. The download includes an updated version of the javac compiler, called the "Type Annotations compiler" or "JSR 308 compiler", that will be shipped with Java 7. In order to use the updated compiler when you type `javac`, add the directory `C:\Program Files\checkers\binary` to your path variable. Also set a CHECKERS variable.
   To set an environment variable, you have two options: make the change temporarily or permanently.
   - To make the change **temporarily**, type at the command shell prompt:
     `path = `*`newdir`*`;%PATH%`
     For example:

     ```
     path = C:\Program Files\checkers\binary;%PATH%
     set CHECKERS = C:\Program Files\checkers
     ```

     This is a temporary change that endures until the window is closed, and you must re-do it every time you start a new command shell.
   - To make the change **permanently**, Right-click the `My Computer` icon and select `Properties`. Select the `Advanced` tab and click the `Environment Variables` button. In the `System Variables` pane, select `Path` from the list and click `Edit`. In the `Edit System Variable` dialog box, move the cursor to the beginning of the string in the `Variable Value` field and type the full directory name followed by a semicolon (`;`).
     Similarly, set the CHECKERS variable.
     This is a permanent change that only needs to be done once ever.
3. Verify that the installation works. From the command line, run:

   ```
   javac -version
   ```

   The output should be:

   ```
   javac 1.7.0-jsr308-0.9.6
   ```

That's all there is to it! Now you are ready to start using the checkers.

Section 1.3 walks you through a simple example. More detailed instructions for using a checker appear in Section 2.

1.3 Example use: detecting a null pointer bug

To run a checker on a source file, just run javac as usual, passing the `-processor` flag. For instance, if you usually run the compiler like this:

```
javac Foo.java Bar.java
```

then you will instead run it like this (where `javac` is the JSR 308 compiler that is distributed with the Checker Framework):

```
javac -processor ProcessorName Foo.java Bar.java
```

(If you usually do your coding within an IDE, you will need to configure the IDE to use the correct version of javac and to pass the command-line argument. See your IDE documentation for details.)

1. Let's consider this very simple Java class. One local variable is annotated as `NonNull`, indicating that `ref` must be a reference to a non-null object. Save the file as `GetStarted.java`.

   ```
   import checkers.nullness.quals.*;

   public class GetStarted {
       void sample() {
           @NonNull Object ref = new Object();
       }
   }
   ```

2. Run the nullness checker on the class. Either run this from the command line:

   ```
   javac -processor checkers.nullness.NullnessChecker GetStarted.java
   ```

   or compile from within your IDE, which you have customized to use the JSR 308 compiler and to pass the extra arguments.
   The compilation should complete without any errors.
3. Let's introduce an error now. Modify `ref`'s assignment to:

   ```
   @NonNull Object ref = null;
   ```

4. Run the nullness checker again, just as before. This run should emit the following error:

   ```
   GetStarted.java:5: incompatible types.
   found   : @Nullable <nulltype>
   required: @NonNull Object
                   @NonNull Object ref = null;
                                         ^
   1 error
   ```

   The type qualifiers (e.g. `@NonNull`) are permitted anywhere that would write a type, including generics and casts; see Section 2.1.

   ```
   @Interned String intern() { ... }             // return value
   int compareTo(@NonNull String other) { ... }  // parameter
   @NonNull List<@Interned String> messages;     // non-null list of interned Strings
   ```

2 Using a checker

Finding bugs with a checker plugin is a two-step process:

1. The programmer writes annotations, such as `@NonNull` and `@Interned`, that specify additional information about Java types. (Or, the programmer uses an inference tool to automatically insert annotations in his code: see Sections 3.2.3 and 6.2.2.) It is possible to annotate only part of your code: see Section 2.3.
2. The checker reports whether the program contains any erroneous code — that is, code that is inconsistent with the annotations.

2.1 Writing annotations

The syntax of type qualifier annotations in Java 7 is specified by JSR 308 [Ern08]. Ordinary Java permits annotations on declarations. JSR 308 permits annotations anywhere that you would write a type, including generics and casts. You can also write annotations to indicate type qualifiers for array levels and receivers. Here are a few examples:

```
@Interned String intern() { ... }              // return value
int compareTo(@NonNull String other) { ... }  // parameter
String toString() @ReadOnly { ... }            // receiver ("this" parameter)
@NonNull List<@Interned String> messages;      // generics:  non-null list of interned Strings
@Interned String @NonNull [] messages;         // arrays:  non-null array of interned Strings
myDate = (@ReadOnly Date) readonlyObject;      // cast
```

You can also write the annotations within comments, as in `List</*@NonNull*/ String>`. The Type Annotations compiler, which is distributed with the Checker Framework, will still process the annotations. However, your code will remain compilable by people who are not using the JSR 308 or Java 7 compiler. For more details, see Section 2.9.

2.1.1 Distributing your annotated project

If your code contains any annotations (outside of comments, see Section 2.9), or any import statements for the annotations, then your code has a dependency on the annotation declarations. You also will need to provide the annotation declarations as well, if you decide to distribute your project.

For your convenience, inside the the checkers distribution `.zip` file is a jar file, `checkers-quals.jar`, that only contains the distributed qualifiers. You may include the jar file in your distribution.

Your clients need to have the annotations jar in the classpath when compiling your project. When running it though, they most likely don't require the annotations declarations (unless the annotation classes are loaded via reflection, which would be unusual).

2.2 Running a checker

To run a checker plugin, run the compiler `javac` as usual, but pass the `-processor` *`plugin_class`* command-line option. (You might run the compiler from the command line as shown below, or your IDE might run the javac command on your behalf, in which case see the IDE documentation to learn how to customize it.) Remember that you must be using the Type Annotations version of `javac`, which you already installed (see Section 1.2).

Two concrete examples (using the Nullness checker) are:

```
javac -processor checkers.nullness.NullnessChecker MyFile.java
javac -processor checkers.nullness.NullnessChecker -sourcepath checkers/jdk/nullness/src MyFile.java
```

For a discussion of the `-sourcepath` argument, see Section 9.2.

The checker is run only on the Java files specified on the command line (or created by another annotation processor). The checker does not analyze other classes (e.g., pre-compiled classes, or classes whose source code is available on the classpath), but it does check the *uses* of those classes in the source code being compiled.

The javac compiler halts compilation as soon as an error is found in a source file. You can pass `-Awarns` in the command-line to treat checker errors as warnings. This option allows you to see all the type-checking errors at once, rather than just the errors in the first file that contains errors.

You can always compile the code without the `-processor` command-line option, but in that case no checking of the type annotations is performed.

2.2.1 Checker auto-discovery

"Auto-discovery" makes the `javac` compiler always run a checker plugin, even if you do not explicitly pass the `-processor` command-line option. This can make your command line shorter, and ensures that your code is checked even if you forget the command-line option.

To enable auto-discovery, place a configuration file named `META-INF/services/javax.annotation.processing.Processor` in your classpath. The file contains the names of the checker plugins to be used, listed one per line. For instance, to run the Nullness and the Interning checkers automatically, the configuration file should contain:

```
checkers.nullness.NullnessChecker
checkers.interning.InterningChecker
```

You can disable this auto-discovery mechanism by passing the `-proc:none` command-line option to `javac`.

2.2.2 Ant task

If you use the Ant build tool to compile your software, then you can add an Ant task that runs a checker. We assume that your Ant file already contains a compilation target that uses the `javac` task.

First, set the `jsr308javac` property:

```
<!-- Boilerplate to set jsr308javac property. Is there a better way? -->
<property environment="env"/>
<condition property="isUnix">
  <os family="unix" />
</condition>
<condition property="isWindows">
  <os family="windows" />
</condition>
<target name="init-jsr308javac-unix" if="isUnix">
  <property name="jsr308javac" value="${env.CHECKERS}/binary/javac" />
</target>
<target name="init-jsr308javac-windows" if="isWindows">
  <property name="jsr308javac" value="${env.CHECKERS}/binary/javac.bat" />
</target>
```

The `property` target makes environment variables (such as your home directory) available to Ant.

Next, duplicate the compilation target, then modify it slightly as indicated in this example, filling in each ellipsis (...) from the original compilation target:

```
<target name="check-nullness"
        description="Check for nullness errors."
        depends="clean,...,init-jsr308javac-unix,init-jsr308javac-windows">
  <javac ...
         fork="yes"
         executable="${jsr308javac}">
    <compilerarg value="-version"/>
    <compilerarg line="-target 5"/>
    <compilerarg line="-processor checkers.nullness.NullnessChecker"/>
    <compilerarg line="-sourcepath ${env.CHECKERS}/jdk/nullness/src"/>
    <compilerarg value="-implicit:class"/>
    <classpath>
      <pathelement location="${env.annotations}/checkers/checkers.jar"/>
      ...
    </classpath>
    ...
  </javac>
</target>
```

In the example, the target is named `check-nullness`, but you can name it whatever you like.

The target assumes the existence of a `clean` target that removes all `.class` files. That is necessary because Ant's `javac` target doesn't re-compile `.java` files for which a `.class` file already exists.

The `executable` and `fork` fields of the `javac` task ensure that an external javac program is called. Otherwise, Ant will run javac via a Java method call, and there is no guarantee that it will get the JSR 308 version that is distributed with the Checker Framework.

The `-version` compiler argument is just for debugging; you may omit it.

The `-target 5` compiler argument is optional, if you use Java 5 in ordinary compilation when not performing pluggable type-checking.

The `-processor ...` compiler argument indicates which checker to run. You can supply additional arguments to the checker as well.

The `-implicit:class` compiler argument causes annotation processing to be performed on implicitly compiled files. (An implicitly compiled file is one that was not specified on the command line, but for which the source code is newer than the `.class` file.) This is the default, but supplying the argument explicitly suppresses a compiler warning.

2.2.3 Maven plugin

Adam Warski has written a Maven2 plugin that runs a checker. The plugin is available at `http://www.warski.org/checkersplugin.html`.

2.2.4 Eclipse

There are two ways to run a checker from within the Eclipse IDE: via Ant or using an Eclipse plug-in.

Using an Ant task Add an Ant target as described in Section 2.2.2. You can run the Ant target by executing the following steps (instructions copied from `http://www.eclipse.org/documentation/?topic=/org.eclipse.platform.doc.user/gettingStarted/qs-84_run_ant.htm`):

1. Select `build.xml` in one of the navigation views and choose **Run As** > **Ant Build...** from its context menu.
2. A launch configuration dialog is opened on a launch configuration for this Ant buildfile.
3. In the **Targets** tab, select the new ant task (e.g., check-interning).
4. Click **Run**.
5. The Ant buildfile is run, and the output is sent to the Console view.

Eclipse plug-in A prototype Eclipse plug-in for running a checker is available at `http://groups.csail.mit.edu/pag/jsr308/eclipse/`. The website contains instructions for installing and using the plug-in. The plug-in is experimental now, but some people have used it successfully (and we have fixed all bugs that have been reported so far).

2.2.5 tIDE

tIDE, an open-source Java IDE, supports the Checker Framework. See its documentation at `http://tide.olympe-network.com/`.

2.3 Checking partially-annotated programs: handling unannotated code

Sometimes, you wish to type-check only part of your program. You might focus on the most mission-critical or error-prone part of your code. When you start to use a checker, you may not wish to annotate your entire program right away. You may not have source code (or enough knowledge to annotate) the libraries that your program uses.

If annotated code uses unannotated code, then the checker may issue warnings. For example, the Nullness checker (Section 3) will warn whenever an unannotated method result is used in a non-null context:

```
@NonNull myvar = unannotated_method();   // WARNING: unannotated_method may return a null value
```

If the call can return null, you should fix the bug in your program by removing the @NonNull annotation in your own program.

If the library call never returns null, there are several ways to eliminate the compiler warnings.

1. Annotate unannotated_method in full. This approach provides the the strongest guarantees, but may require you to annotate additional methods that unannotated_method calls.
2. Annotate only the signature of unannotated_method, and suppress warnings in its body. Two ways to do this are via a @SuppressWarnings annotation or by not running the checker on that file (see Section 2.4).
3. Suppress all warnings related to uses of unannotated_method via the skipClasses processor option (see Section 2.4). Since this can suppress more warnings than you may expect, it is usually better to annotate at least the method's signature. If you choose the boundary between the annotated and unannotated code wisely, then you only have to annotate the signatures of a few classes/methods (e.g., the public interface to a library or package).

Section 9 discusses adding annotations to signatures when you do not have source code available. Section 2.4 discusses suppressing warnings.

If you annotate additional libraries, please share them with us so that we can distribute the annotations with the Checker Framework; see Section 11.2.

2.4 Suppressing warnings

You may wish to suppress checker warnings because of unannotated libraries or un-annotated portions of your own code, because of application invariants that are beyond the capabilities of the type system, because of checker limitations, because you are interested in only some of the guarantees provided by a checker, or for other reasons. You can suppress warnings via

- the @SuppressWarnings annotation,
- the -AskipClasses command-line option,
- the javac -Alint command-line option, or
- not using the -processor switch to javac.

You can suppress specific errors and warnings by use of the @SuppressWarnings("*annotationname*") annotation, for example @SuppressWarnings("interning"). This may be placed on program elements such as a class, method, or local variable declaration. It is good practice to suppress warnings in the smallest possible scope. For example, if a particular expression causes a false positive warning, you should extract that expression into a local variable and place a @SuppressWarnings annotation on the variable declaration. As another example, if you have annotated the signatures but not the bodies of the methods in a class or package, put a @SuppressWarnings annotation on the class declaration or on the package's package-info.java file.

You can suppress all errors and warnings at all uses of a given class. Set the -AskipClasses command-line option to a regular expression that matches classes for which warnings and errors should be suppressed. For example, if you use "-AskipClasses=^java\." on the command line (with appropriate quoting) when invoking javac, then the checkers will suppress all warnings within those classes, all warnings relating to invalid arguments, and all warnings relating to incorrect use of the return value.

You can suppress an entire class of warnings via javac's -Alint command-line option. The -Alint option uses the same syntax as javac's -Xlint option. Following -Alint=, write a list of option names. If the option name is preceded by a hyphen (-), that disables the option; otherwise it enables it. For example: -Alint=-dotequals causes the Interning checker (Section 4) not to output advice about when a.equals(b) could be replaced by a==b.

You can also compile parts of your code without use of the -processor switch to javac. No checking is done during such compilations.

Finally, some checkers have special rules. For example, the Nullness checker (Section 3) uses assert statements that contain null checks to suppress warnings (Section 3.3.1).

2.5 Polymorphism and generics

2.5.1 Generics (parametric polymorphism or type polymorphism)

The Checker Framework fully supports qualified Java generic types (also known in the literature as "parametric polymorphism"). Before running a checker, we recommend that you eliminate raw types (e.g., List as opposed to List<...>) from your code. Using generics helps prevent type errors just as using a pluggable type-checker does.

When instantiating a generic type, clients supply the qualifier along with the type argument, as in List<@NonNull String>.

The declaration (that is, the implementation) of a generic class may use the extends clause to restrict the types and qualifiers that may be used for instantiating. For example, given the declaration class MyClass<T extends @NonNull Object> {...}, a client could use MyClass<@NonNull String> but not MyClass<@Nullable String>.

Style note: When using the Nullness checker (Section 3), programmers sometimes write extends @NonNull Object even though it's the default. The reason is that code with no extends clause, like

```
class C<T> { ... }
```

typically means that class C can be instantiated with any type argument at all. But in the Nullness type system, to permit all type arguments, to obtain that effect one must write

```
class C<T extends @Nullable Object> { ... }
```

Type annotations on generic type variables A type annotation on a generic type variable overrides/ignores any type qualifier (in the same type hierarchy) on the corresponding actual type argument. For example, @Nullable T applies the type qualifier @Nullable to the (unqualified) Java type of the type argument T.

Here is an example of applying a type annotation to a generic type variable:

```
class MyClass2<T> {
  ...
  @Nullable T = null;
  ...
}
```

The type annotation does not restrict how MyClass2 may be instantiated (only the optional extends clause on the declaration of type variable T would do so). In other words, both MyClass2<@NonNull String> and MyClass2<@Nullable String> are legal, and in both cases @Nullable T means @Nullable String. In MyClass2<@Interned String>, @Nullable T means @Nullable @Interned String.

2.5.2 Qualifier polymorphism

The Checker Framework also supports type *qualifier* polymorphism for methods, which permits a single method to have multiple different qualified type signatures.

A polymorphic qualifier's definition is marked with @PolymorphicQualifier. For example, @PolyNull is a polymorphic type qualifier for the Nullness type system:

```
@PolymorphicQualifier
public @interface PolyNull { }
```

A method written using a polymorphic qualifier conceptually has multiple versions, somewhat like a template in C++. In each version, the polymorphic qualifier has been replaced by another qualifier from the hierarchy. See the examples in Section 2.5.2.

The method body must type-check with all signatures. A method call is type-correct if it type-checks under any signature.

Polymorphic qualifiers can be used within a method body. They may not be used on classes or fields.

Examples of using polymorphic qualifiers As an example of the use of `@PolyNull`, method `Class.cast` returns null if and only if its argument is `null`:

```
@PolyNull T cast(@PolyNull Object obj) { ... }
```

This is like writing:

```
 @NonNull T cast( @NonNull Object obj) { ... }
@Nullable T cast(@Nullable Object obj) { ... }
```

except that the latter is not legal Java, since it defines two methods with the same Java signature.

As another example, consider

```
@PolyNull T max(@PolyNull T x, @PolyNull T y);
```

which is like writing

```
 @NonNull T max( @NonNull T x,  @NonNull T y);
@Nullable T max(@Nullable T x, @Nullable T y);
```

One way of thinking about which one of the two `max` variants is selected is that the nullness annotations of (the declared types of) both arguments are *unified* to a type that is a subtype of both. If both arguments are `@NonNull`, their unification is `@NonNull`, and the result is `@NonNull`. But if even one of the arguments is `@Nullable`, then the result is `@Nullable`.

It does not make sense to write only a single instance of a polymorphic qualifier in a method definition, as in

```
void m(@PolyNull Object obj)
```

which expands to

```
void m(@NonNull Object obj)
void m(@Nullable Object obj)
```

which is no different than writing just

```
void m(@Nullable Object obj)
```

The benefit of polymorphic qualifiers comes when one is used multiple times in a method, since then each instance turns into the same type qualifier. Most frequently, the polymorphic qualifier appears on both the return type and at least one formal parameter. It can also be useful to have polymorphic qualifiers on (only) multiple formal parameters, especially if the method side-effects one of its arguments.

2.6 Unused fields and dependent types

Sometimes, the type of a field depends on the qualifier on the receiver. The Checker Framework supports two varieties of such a field: fields that may not be used if the receiver has a given qualifier, and fields whose qualifier changes based on the qualifier of the receiver.

2.6.1 Unused fields

A Java subtype can have more fields than its supertype. You can simulate the same effect for type qualifiers: a given field may not be accessed via a reference with a supertype qualifier, but can be accessed via a reference with a subtype qualifier.

This permits you to restrict use of a field to certain contexts.

The `@Unused` annotation on a field declares that the field may not be accessed via a receiver of the given qualified type (or any supertype).

2.6.2 Dependent types

A variable has a *dependent type* if its type depends on some other value or type.

The Checker Framework supports a form of dependent types, via the `@Dependent` annotation. This annotation changes the type of a field or variable, based on the qualified type of the receiver (`this`). This can be viewed as a more expressive form of polymorphism (see Section 2.5). It can also be seen as a way of linking the meanings of two type qualifier hierarchies.

When the `@Unused` annotation is sufficient, you should use it instead of `@Dependent`.

2.6.3 Example

Suppose we have a class `Person` and a field `spouse` that is non-`null` if the person is married. We could declare this as

```
class Person {
  ...
  // non-null if this person is married
  @Nullable Person spouse;
  ...
}
```

Now, suppose that we have defined the qualifier hierarchy in which `@Single` (meaning "not married") is a supertype of `@Married`. A more informative declaration would be

```
class Person {
  ...
  @Nullable @Dependent(result=NonNull.class, when=Married.class) Person spouse;
  ...
}
```

If a person is known to be `@Married`, the `spouse` field is known to be non-`null`:

```
class Person {
  ...

  void celebrateWeddingAnniversary() @Married {
    System.out.println("Happy anniversary, "
                       + spouse.toString()); // no possible null pointer exception
  }

  ...
}
```

Without the `@Dependent` annotation on the declaration of the `spouse` variable, the Nullness Checker would complain that `toString` was being invoked on a possibly-`null` value.

An even better declaration is

```
class Person {
  ...
  @Unused(when=Single.class) @NonNull Person spouse;
  ...
}
```

Then, if a person is known to be `@Married` (or more appropriately non-`@Single`), the `spouse` field is known to be non-`null`. Also, if a person is known to be `@Single`, the `spouse` field may not be accessed:

```
@Single Person person = ...;
Person spouse = person.spouse;  // invalid field access
...
```

2.7 The effective qualifier on a type (defaults and inference)

A checker sometimes treats a type as having a slightly different qualifier than what is written on the type — especially if the programmer wrote no qualifier at all. Most readers can skip this section on first reading, because you will probably find the system simply "does what you mean", without forcing you to write too many qualifiers in your program.

The following steps determine the effective qualifier on a type — the qualifier that the checkers treat as being present.

1. The type system adds implicit qualifiers. Implicit qualifiers can be built into a type system (Section 10.3), in which case the type system's documentation should explain all of the type system's implicit qualifiers. Or, a programmer may introduce an implicit annotations on each use of class *C* by writing a qualifier on the declaration of class *C*.
   - Example 1 (built-in): In the Nullness type system, `enum` values are never null, nor is a method receiver.
   - Example 2 (built-in): In the Interning type system, string literals and `enum` values are always interned.
2. If a type qualifier is present in the source code, that qualifier is used.
   If the type has an implicit qualifier, then it is an error to write an explicit qualifier that is equal to (redundant with) or a supertype of (weaker than) the implicit qualifier. A programmer may strengthen (write a subtype of) an implicit qualifier, however.
3. If there is no implicit or explicit qualifier on a type, then a default qualifier may be applied; see Section 2.7.1.

   At this point, every type has a qualifier.
4. The type system may refine a qualified type on a local variable — that is, treat it as a subtype of how it was declared or defaulted. This refinement is always sound and has the effect of eliminating false positive error messages. See Section 2.7.2.

2.7.1 Default qualifier for unannotated types

A type system designer, or an end-user programmer, can cause unannotated references to be treated as if they had a default annotation.

There are several defaulting mechanisms, for convenience and flexibility. When determining the default qualifier for a use of a type, the following rules are used in order, until one applies.

- Use the innermost user-written `@DefaultQualifier`, as explained in this section.
- Use the default specified by the type system designer (Section 10.2.3).
- Use `@Unqualified`, which the framework inserts to avoid ambiguity and simplify the programming interface for type system designers. Users do not have to worry about this detail.

The end-user programmer specifies a default qualifier by writing the `@DefaultQualifier` annotation on a package, class, method, or variable declaration. The argument to `@DefaultQualifier` is the fully qualified `String` name of an annotation, and its optional second argument indicates where the default applies. If the second argument is omitted, the specified annotation is the default in all locations. See the Javadoc of `DefaultQualifier` for details.

If you wish to write multiple `@DefaultQualifier` annotations at a single location, use `@DefaultQualifiers` instead.

If `@DefaultQualifier[s]` is placed on a package (via the `package-info.java` file), then it applies to the given package *and* all subpackages.

Setting a default for class declarations may have an unexpected result: a type qualifier on a class declaration gives an implicit qualifier (Section 2.7) that can only be strengthened, not weakened. Thus, you may want to put explicit qualifiers on class declarations, or exclude class declarations from defaulting.

Example This example shows use of both `@DefaultQualifier` and `@DefaultQualifiers`. (The example uses the Nullness type system (Section 3) and the IGJ type system (Section 5).)

```
@DefaultQualifiers({
  @DefaultQualifier("checkers.nullness.quals.NonNull"),
  @DefaultQualifier("checkers.igj.quals.Mutable")
})
class MyClass {

  public boolean compile(File myFile) { // myFile has type "@NonNull @Mutable File"
    if (!myFile.exists())          // no warning: myFile is non-null
      return false;
    @Nullable File srcPath = ...;  // must annotate to specify "@Nullable File"
    ...
    if (srcPath.exists())          // warning: srcPath might be null
      ...
  }

  @DefaultQualifier("checkers.igj.quals.ReadOnly")
  public boolean isJavaFile(File myfile) {  // myFile has type "@NonNull @ReadOnly File"
    ...
  }
}
```

When a default qualifier may not be specified Sometimes, the meaning of an unannotated reference is determined by the type system. For example, in the Interning type system, each type is either unqualified, or it has the `@Interned` qualifier. In such a case, specifying a default for unannotated types is not sensible.

In other cases, the type hierarchy has an explicit qualifier for every possible meaning. For example, the Nullness type system has `@Nullable` types and `@NonNull` types. It has no built-in meaning for unannotated types; a user may specify a default qualifier.

Permitting users to specify defaults is a reason you may wish to make your type hierarchy "complete", in the sense that there is a qualifier for every location in the hierarchy.

2.7.2 Automatic type refinement (flow-sensitive type qualifier inference)

In order to reduce the burden of annotating types in your program, the checkers soundly treat certain variables and expressions as having a subtype of their declared or defaulted (Section 2.7.1) type. This functionality never introduces unsoundness or causes an error to be missed: it merely suppresses false positive warnings.

By default, all checkers, including new checkers that you write, can take advantage of this functionality. Most of the time, users don't have to think about, and may not even notice, this feature of the framework. The checkers simply do the right thing even when a programmer forgets an annotation on a local variable, or when a programmers writes an unnecessarily general type in a declaration.

If you are curious or want more details about this feature, then read on.

As an example, the Nullness checker (Section 3) can automatically determine that certain variables are non-null, even if they were explicitly or by default annotated as nullable. A variable or expression can be treated as `@NonNull` from the time that it is either assigned a non-null value or checked against null (e.g., via an assertion, `if` statement, or being dereferenced), until it might be re-assigned (e.g., via an assignment that might affect this variable, or via a method call that might affect this variable).

As with explicit annotations, the implicitly non-null types permit dereferences and assignments to explicitly non-null types, without compiler warnings.

Consider this code, along with comments indicating whether the Nullness checker (Section 3) issues a warning. Note that the same expression may yield a warning or not depending on its context.

```
// Requires an argument of type @NonNull String
void parse(@NonNull String toParse) { ... }

// Argument does NOT have a @NonNull type
void lex(String toLex) {
  parse(toLex);        // warning:  toLex might be null
  if (toLex != null) {
    parse(toLex);      // no warning:  toLex is known to be non-null
  }
  parse(toLex);        // warning:  toLex might be null
  toLex = new String(...);
  parse(toLex);        // no warning:  toLex is known to be non-null
}
```

If you find instances where you think a value should be inferred to have (or not have) a given annotation, but the checker does not do so, please submit a bug report (see Section 11.2) that includes a small piece of Java code that reproduces the problem.

Type inference is never performed for method parameters of non-private methods and for non-private fields, because unknown client code could use them in arbitrary ways. The inferred information is never written to the `.class` file as user-written annotations are.

The inference indicates when a variable can be treated as having a subtype of its declared type — for instance, when an otherwise nullable type can be treated as a `@NonNull` one. The inference never treats a variable as a supertype of its declared type (e.g., an expression of `@NonNull` type is never inferred to be treated as possibly-null).

2.8 What the checker guarantees

A checker can guarantee that a particular property holds throughout the code. For example, the Nullness checker (Section 3) guarantees that every expression whose type is a `@NonNull` type never evaluates to null. The Interning checker (Section 4) guarantees that every expression whose type is an `@Interned` type evaluates to an interned value. The checker makes its guarantee by examining every part of your program and verifying that no part of the program violates the guarantee.

There are some limitations to the guarantee.

- Native methods and reflection can behave in a manner that is impossible for a compiler plugin to check. Such constructs may violate the property being checked. Similarly, deserialization and cloning can create objects that could not result from normal constructor calls, and that therefore may violate the property being checked.
- A compiler plugin can check only those parts of your program that you run it on. If you compile some parts of your program without the `-processor` switch or with the `-AskipClasses` property (in other words, without running the checker), or if you use the `@SuppressWarnings` annotation to suppress some errors or warnings, then there is no guarantee that the entire program satisfies the property being checked. An analogous situation is using an external library that was compiled without being checked by the compiler plugin.
- Your code should pass the Java compiler without errors or warnings. In particular, your code should use generic types, with no uses of raw types. Misuse of generics, including casting away generic types, can cause other errors to be missed.
- The Checker Framework does not yet support annotations on intersection types (see JLS §4.9). As a result, checkers cannot provide guarantees about intersection types.
- Specific checkers may have other limitations; see their documentation for details.

A checker can be useful in finding bugs or in verifying part of a program, even if the checker is unable to verify the correctness of an entire program.

If you find that a checker fails to issue a warning that it should, then please report a bug (see Section 11.2).

2.9 Writing annotations in comments for backward compatibility

Sometimes, your code needs to be compilable by people who are not using the Type Annotations or Java 7 compiler.

2.9.1 Annotations in comments

A Java 4 compiler does not permit use of annotations, and a Java 5 compiler only permits annotations on declarations (but not on generic arguments, casts, method receiver, etc.).

For compatibility with all Java versions, you may write any annotation inside a `/*...*/` Java comment, as in `List</*@NonNull*/ String>`. The Type Annotations compiler treats the code exactly as if you had not written the `/*` and `*/`. In other words, the Type Annotations compiler will recognize the annotation, but your code will still compile with any other Java compiler.

(**Note:** This is a feature of the Type Annotations compiler that is distributed along with the Checker Framework. It is not supported by the mainline OpenJDK compiler, which will ignore annotations written in comments. This is the only difference between the Type Annotations compiler and the OpenJDK compiler.)

In a single program, you may write some annotations in comments, and others without comments.

By default, the compiler ignores any comment that contains spaces at the beginning or end, or between the `@` and the annotation name. In other words, it reads `/*@NonNull*/` as an annotation but ignores `/* @NonNull*/` or `/*@ NonNull*/` or `/*@NonNull */`. This feature enables backward compatibility with code that contains comments that start with `@` but are not annotations. (The ESC/Java [FLL+02], JML [LBR06], and Splint [Eva96] tools all use "`/*@`" or "`/* @`" as a comment marker.) Compiler flag `-XDTA:spacesincomments` causes the compiler to parse annotation comments even when they contain spaces. You may need to use `-XDTA:spacesincomments` if you use Eclipse's "Source > Correct Indentation" command, since it inserts space in comments. But the annotation comments are less readable with spaces, so you may wish to disable inserting spaces: in the Formatter preferences, in the Comments tab, unselect the "enable block comment formatting" checkbox.

2.9.2 Import statements

When writing source code with annotations, it is more convenient to write a short form such as `@NonNull` instead of `@checkers.nullness.quals.NonNull`. There are two ways to do this.

- Write an import statement like: `import checkers.nullness.quals.*;`
  A disadvantage of this is that everyone who compiles the code (even using a non-JSR-308 compiler) must have the annotation definitions (e.g., the `checkers.jar` or `checkers-quals.jar` file) on their classpath. The reason is that a Java compiler issues an error if an imported package is not on the classpath. See Section 2.1.1.
- When you compile the code, set the shell environment variable `jsr308_imports`. This permits your code to compile whether or not the Type Annotations compiler is being used.
  In bash, you could write `export jsr308_imports='checkers.nullness.quals.*'`, or prefix the `javac` command by `jsr308_imports='checkers.nullness.quals.*'` . Alternately, you can set the system variable via the javac command line argument `-J-Djsr308_imports="checkers.nullness.quals.*"`.
  You can specify multiple packages separated by the classpath separator (same as the file path separator: `;` for Windows, and `:` for Unix and Mac.). For example, to implicitly import the Nullness and Interning qualifiers, set `jsr308_imports` to `checkers.nullness.quals.*:checkers.interning.quals.*`.

2.9.3 Migrating away from annotations in comments

If your codebase currently uses annotations in comments, but you are willing to use only compilers that support type annotations (such as any Java 7 compiler), then you can remove the comment characters around your annotations. This Unix command will do so, for all Java files in the current working directory or any subdirectory.

```
find . -type f -name '*.java' -print \
  | xargs grep -l -P '/\*\s*@([^ */]+)\s*\*/' \
  | xargs perl -pi.bak -e 's|/\*\s*@([^ */]+)\s*\*/|@\1|g'
```

You can customize this command:

- To process comments with embedded spaces and asterisks, change two instances of "`[^ */]`" to "`[^/]`".
- To ignore comments with leading or trailing spaces, remove the four instances of "`\s*`".
- To not make backups, remove "`.bak`".

If you are using implicit import statements (Section 2.9.2), you may also need to introduce explicit import statements into your code.

2.10 Tips about writing annotations

2.10.1 Annotations on constructor invocations

In the checkers distributed with the Checker Framework, an annotation on a constructor invocation is equivalent to a cast on a constructor result. That is, the following two expressions have identical semantics: one is just shorthand for the other.

```
new @ReadOnly Date()
(@ReadOnly Date) new Date()
```

However, you should rarely have to use this. The Checker Framework will determine the qualifier on the result, based on the "return value" annotation on the constructor definition.

2.10.2 Annotations indicate normal behavior

You should use annotations to indicate *normal* behavior. The annotation indicate all the values that you *want* to flow to reference — not every value that might possibly flow there if your program has a bug.

Many methods are guaranteed to throw an exception if they are passed `null` as an argument. Examples include

```
java.lang.Double.valueOf(String)
java.lang.String.contains(CharSequence)
org.junit.Assert.assertNotNull(Object)
com.google.common.base.Preconditions.checkNotNull(Object)
```

In some sense, `null` is a legal argument with a well-defined semantics: throw an exception. Another reason `@Nullable` might seem like a reasonable annotation is that it describes a possible program execution: it might be possible for `null` to flow there, if your program has a bug.

However, it is never useful for a programmer to pass `null`. It is the programmer's intention that `null` never flows there. If `null` does flow there, the program will not continue normally.

Therefore, you should mark such parameters as `@NonNull`, indicating the intended use of the method. The `@NonNull` annotation will enable compile-time warnings about possible run-time exceptions, which is the purpose of the checker.

2.10.3 Annotations indicate a contract

Annotations indicate guarantees that a client can depend upon. A subclass is permitted not permitted to weaken the contract; for example, if a method accepts `null` as an argument, then every overriding definition must also accept `null`. A subclass is permitted to strengthen the contract; for example, if a method does *not* accept `null` as an argument, then an overriding definition is permitted to accept `null`.

As a bad example, consider an erroneous `@Nullable` annotation at line 141 of `com/google/common/collect/Multiset.java`, version r78:

```
101  public interface Multiset<E> extends Collection<E> {
...
122    /**
123     * Adds a number of occurrences of an element to this multiset.
...
129     * @param element the element to add occurrences of; may be {@code null} only
130     *     if explicitly allowed by the implementation
...
137     * @throws NullPointerException if {@code element} is null and this
138     *     implementation does not permit null elements. Note that if {@code
139     *     occurrences} is zero, the implementation may opt to return normally.
140     */
141    int add(@Nullable E element, int occurrences);
```

There exist implementations of Multiset that permit `null` elements, and implementations of Multiset that do not permit `null` elements. A client with a variable `Multiset ms` does not know which variety of Multiset `ms` refers to. However, the `@Nullable` annotation promises that `ms.add(null, 1)` is permissible. (Recall from Section 2.10.2 that annotations should indicate normal behavior.)

If parameter `element` on line 141 were to be annotated, the correct annotation would be `@NonNull`. Given an arbitrary Multiset `ms`, the only way to be sure not to throw an exception is to pass only non-`null` elements to its `add` method. A particular class that implements Multiset could declare `add` to take a `@Nullable` parameter. That still satisfies the original contract. It strengthens the contract by promising even more: a client with such a reference can call `add(null)` with confidence that no `NullPointerException` will be thrown.

However, the best annotation for line 141 is no annotation at all. The reason is that each implementation of the Multiset interface should specify its own nullness properties when it specifies the type parameter for Multiset. For example, two clients could be written as

```
class MyNullPermittingMultiset implements Multiset<@Nullable Object> { ... }
class MyNullProhibitingMultiset implements Multiset<@NonNull Object> { ... }
```

or, more generally, as

```
class MyNullPermittingMultiset<E extends @Nullable Object> implements Multiset<E> { ... }
class MyNullProhibitingMultiset<E extends @NonNull Object> implements Multiset<E> { ... }
```

Then, the specification is more informative, and the Checker Framework is able to do more precise checking, than if line 141 has an annotation.

3 Nullness checker

If the Nullness checker issues no warnings for a given program, then running that program will never throw a null pointer exception. This guarantee enables a programmer to prevent errors from occurring when his program is run. See Section 3.3 for more details about the guarantee and what is checked.

You can control the behavior of the Nullness checker via the `-Alint` options `flow`, `cast`, and `cast:redundant`.

3.1 Nullness annotations

The Nullness checker uses two separate type hierarchies: one for nullness, and one for rawness (see Section 3.1.1).

The nullness hierarchy contains these qualifiers:

`@Nullable` indicates a type that includes the null value. The type `Boolean` is nullable; a variable of type `Boolean` always has one of the values `TRUE`, `FALSE`, or `null`.

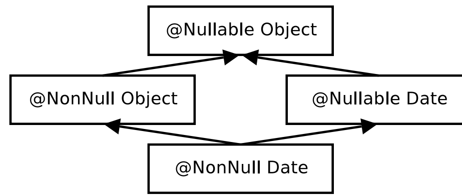

Figure 1: Partial type hierarchy for the Nullness type system. Java's `Object` is expressed as `@Nullable Object`. Programmers can omit most type qualifiers, because the default annotation (Section 3.2.2) is usually correct.

`@NonNull` indicates a type that does not include the null value. The type `boolean` is non-null; a variable of type `boolean` always has one of the values `true` or `false`. The type `@NonNull Boolean` is also non-null: a variable of type `@NonNull Boolean` always has one of the values `TRUE` or `FALSE` — never `null`. Dereferencing an expression of non-null type can never cause a null pointer exception. Furthermore, the object referenced by a `@NonNull` type is always fully initialized — that is, its `@NonNull` fields have been set to a non-null value.

`@PolyNull` indicates qualifier polymorphism. For a description of `@PolyNull`, see Section 2.5.2.

`@LazyNonNull` indicates a reference that may be `null`, but if it ever becomes non-`null`, then it never becomes `null` again. This is appropriate for lazily-initialized fields, among other uses. When the variable is read, its type is treated as `@Nullable`, but when the variable is assigned, its type is treated as `@NonNull`.
Because the Nullness checker works intraprocedurally (it analyzes one method at a time), when a `LazyNonNull` field is first read within a method, the field cannot be assumed to be non-null. The benefit of LazyNonNull over Nullable is its different interaction with flow-sensitive type qualifier refinement. After a check of a LazyNonNull field, all subsequent accesses *within that method* can be assumed to be NonNull, even after arbitrary external method calls that have access to the given field.

Figure 1 shows part of the type hierarchy for the Nullness type system.

3.1.1 `@Raw` annotation for partially-initialized objects

The rawness hierarchy indicates whether an object is fully initialized — that is, whether its fields have all been assigned. This is mostly relevant within the constructor, or for references to `this` that escape the constructor. The rawness hierarchy contains these qualifiers:

`@Raw` indicates a type that contains a partially-initialized object. In a partially-initialized object, fields that are annotated as `@NonNull` may be null because the field has not yet been assigned. Within the constructor, `this` has `@Raw` type until all the fields have been assigned.

`@NonRaw` indicates a type that contains a fully-initialized object. `NonRaw` is the default, so there is little need for a programmer to write this explicitly.

`@PolyRaw` indicates qualifier polymorphism over rawness (see Section 2.5.2).

During execution of a constructor, every field of non-primitive type starts out with the value `null`. If the field has `@NonNull` type, its initial value `null` violates the `@NonNull` type qualifier. In other words, during construction, the object is in an illegal state.

The `@Raw` type annotation represents a partially-initialized object. If a reference has `@Raw` type, then all of its `@NonNull` fields are treated as `@LazyNonNull`: when read, they are treated as being `@Nullable`, but when written, they are treated as being `@NonNull`.

The rawness hierarchy is **orthogonal** to the nullness hierarchy. It is legal for a reference to be `@NonNull @Raw`, `@Nullable @Raw`, `@NonNull @NonRaw`, or `@Nullable @NonRaw`. The nullness hierarchy tells you about the reference itself: might the reference be null? The rawness hierarchy tells you about the fields in the referred-to object: might those fields be null?

You can suppress warnings related to partially-initialized objects with `@SuppressWarnings("rawness")`. (Do not confuse this with the unrelated `@SuppressWarnings("rawtypes")` annotation for non-instantiated generic types!)

How an object becomes non-raw The Nullness checker issues an error if the constructor fails to initialize any non-null field. This ensures that the object is in a legal (non-raw) state by the time that the constructor exits. This is different than Java's test for definite assignment (see JLS ch.16), which does not apply to fields because fields have a default value of null.

Within the constructor, `this` has `@Raw` type. As soon as all of the `@NonNull` fields have been initialized, then `this` is treated as non-raw.

Suppose that class C extends class B, which extends class A. Within the C constructor, until the superclass constructor is called, `this` has type `@Raw C` and also `@Raw B` and `@Raw A`. After the superclass constructor has been called, then `this` has type `@Raw C` and also `@NonRaw B` and `@NonRaw A`.

A note about the terminology "raw" The name "raw" comes from a research paper that proposed this approach [FL03]. The `@Raw` annotation has nothing to do with the raw types of Java Generics.

3.2 Writing nullness annotations

3.2.1 Implicit qualifiers

As described in Section 2.7, the Nullness checker adds implicit qualifiers, reducing the number of annotations that must appear in your code. For example, enum types are implicitly non-null, so you never need to write `@NonNull MyEnumType`.

For a complete description of all implicit nullness qualifiers, see the Javadoc for `NullnessAnnotatedTypeFactory`.

3.2.2 Default annotation

Unannotated references are treated as if they had a default annotation, using the NNEL (non-null except locals) rule described below. A user may choose a different rule for defaults using the `@DefaultQualifier` annotation; see Section 2.7.1.

Here are three possible default rules you may wish to use. Other rules are possible but are not as useful.

- `@Nullable`: Unannotated types are regarded as possibly-null, or nullable. This default is backward-compatible with Java, which permits any reference to be null. You can activate this default by writing a `@DefaultQualifier("checkers.nullness.quals.Nullable")` annotation on a class or method declaration.
- `@NonNull`: Unannotated types are treated as non-null. You can activate this default via the `@DefaultQualifier("checkers.nullness.quals.NonNull")` annotation.
- Non-null except locals (NNEL): Unannotated types are treated as `@NonNull`, *except* that the unannotated raw type of a local variable is treated as `@Nullable`. (Any generic arguments to a local variable still default to `@NonNull`.) You can activate this default via the `@DefaultQualifier(value="checkers.nullness.quals.NonNull", locations={DefaultLocation .ALL_EXCEPT_LOCALS})` annotation.
  The NNEL default leads to the smallest number of explicit annotations in your code [PAC+08]. It is what we recommend. If you do not explicitly specify a different default, then NNEL is the default.

3.2.3 Inference of `@NonNull` and `@Nullable` annotations

It can be tedious to write annotations in your code. Two tools exist that can automatically infer annotations and insert them in your source code. (This is different than type qualifier refinement for local variables (Section 2.7.2), which infers a more specific type for local variables and uses them during type-checking but does not insert them in your source code. Type qualifier refinement is always enabled, no matter how annotations on signatures got inserted in your source code.)

Your choice of tool depends on what default annotation (see Section 3.2.2) your code uses. You only need one of these tools.

- Inference of `@Nullable`: If your code uses the standard NNEL (non-null-except-locals) default or the `NonNull` default, then use the AnnotateNullable tool of the Daikon invariant detector.
- Inference of `@NonNull`: If your code uses the Nullable default, use the Non-null checker and inferencer of the JastAdd Extensible Compiler.

3.3 What the Nullness checker checks

The checker issues a warning in two cases:

1. When an expression of non-`@NonNull` type is dereferenced, because it might cause a null pointer exception. Dereferences occur not only when a field is accessed, but when an array is indexed, an exception is caught, a lock is taken in a synchronized block, and more. For a complete description of all checks performed by the Nullness checker, see the Javadoc for `NullnessVisitor`.
2. When an expression of `@NonNull` type might become null, because it is a misuse of the type: the null value could flow to a dereference that the checker does not warn about.

This example shows both sorts of problems:

```
         Object   obj;  // might be null
@NonNull Object nnobj;  // never null
...
  obj.toString()  // checker warning:  dereference might cause null pointer exception
nnobj = obj;      // checker warning:  nnobj may become null
```

Parameter passing and return values are checked analogously to assignments.

3.3.1 Suppressing warnings with assertions

In addition to the other ways of suppressing warnings (Section 2.4), the Nullness checker assumes that assertions succeed. For example, it assumes that no null pointer exception can occur in code such as

```
assert x != null;
... x.f ...
```

(Another way of stating the Nullness checker's use of assertions is as an additional caveat to the guarantees provided by a checker (Section 2.8). The Nullness checker prevents null pointer errors in your code under the assumption that assertions are enabled, and it does not guarantee that all of your assertions succeed.)

3.4 Examples

3.4.1 Tiny examples

To try the Nullness checker on a source file that uses the `@NonNull` qualifier, use the following command (where `javac` is the JSR 308 compiler that is distributed with the Checker Framework):

```
javac -processor checkers.nullness.NullnessChecker examples/NullnessExample.java
```

Compilation will complete without warnings.

To see the checker warn about incorrect usage of annotations (and therefore the possibility of a null pointer exception at run time), use the following command:

```
javac -processor checkers.nullness.NullnessChecker examples/NullnessExampleWithWarnings.java
```

The compiler will issue three warnings regarding violation of the semantics of `@NonNull`.

edu.umd.cs.findbugs.annotations.NonNull	checkers.nullness.quals.NonNull
javax.annotation.Nonnull	checkers.nullness.quals.NonNull
org.jetbrains.annotations.NotNull	checkers.nullness.quals.NonNull
edu.umd.cs.findbugs.annotations.Nullable	checkers.nullness.quals.Nullable
edu.umd.cs.findbugs.annotations.CheckForNull	checkers.nullness.quals.Nullable
edu.umd.cs.findbugs.annotations.UnknownNullness	checkers.nullness.quals.Nullable
javax.annotation.Nullable	checkers.nullness.quals.Nullable
javax.annotation.CheckForNull	checkers.nullness.quals.Nullable
org.jetbrains.annotations.Nullable	checkers.nullness.quals.Nullable

Figure 2: Refactoring for converting nullness annotations from FindBugs, the JSR 305 proposal, and IntelliJ to the Checker Framework.

3.4.2 Annotated library

Three libraries that or annotated with nullness qualifiers are:

- The Nullness checker itself.
- The Daikon invariant detector. Run the command `make check-nullness`.
- The annotation scene library. To run the Nullness checker on the annotation scene library, first download the scene library suite (which includes build dependencies for the scene library as well as its source code) and extract it into your checkers installation. The checker can then be run on the annotation scene library with Apache Ant using the following commands:

  ```
  cd checkers
  ant -f scene-lib-test.xml
  ```

  You can view the annotated source code, which contains `@NonNull` annotations, in the `checkers/scene-lib-test/src/annotations/` directory.

3.5 Other tools for nullness checking

The Checker Framework's nullness annotation is similar to annotations used in IntelliJ IDEA, FindBugs, JML, the JSR 305 proposal, and others. Also see Section 11.5 for a comparison to other tools.

You might prefer to use the Checker Framework because it has a more powerful analysis that can warn you about more null pointer errors in your code.

If you have already annotated your code with a different nullness annotation, you can reuse that effort by converting them to the Checker Framework's nullness annotations. Perform the refactoring described in Figure 2.

Alternately, the Checker Framework can process those other annotations (as well as its own, if they also appear in your program). The Checker Framework has its own definition of the annotations on the left side of Figure 2, so that they can be used as type qualifiers. The Checker Framework interprets them according to the right side of Figure 2.

The Checker Framework may issue more or fewer errors than another tool. This is expected, since each tool uses a different analysis. Remember that the Checker Framework aims at soundness: never failing to report a possible null dereference, while at the same time limiting false reports.

Because some of the names are the same (`NonNull`, `Nullable`), it is unpleasant to use nullness annotations from multiple different packages in the same codebase. You can import at most one of the annotations with conflicting names; the other(s) must be written out fully rather than imported. Also, note FindBugs's non-standard meaning for `@Nullable` (Section 3.5.2).

3.5.1 Which tool is right for you?

Different tools are appropriate in different circumstances. Here is a brief comparison with FindBugs, but similar points apply to other tools.

The reason you might want to use the Checker Framework instead of FindBugs is that FindBugs has a less powerful nullness analysis that reports fewer errors. However, FindBugs does not require you to annotate your code as thoroughly as the Checker Framework does. Depending on the importance of your code, you may wish to do no nullness checking; the cursory checking of FindBugs; or the thorough checking of the Checker Framework. You might even want to ensure that both tools run, for example if your coworkers or some other organization are still using FindBugs. If you know that you will eventually want to use the Checker Framework, there is no point using FindBugs first; it is easier to go straight to using the Checker Framework.

FindBugs can find other errors in addition to nullness errors; here we focus on its nullness checks. Even if you use FindBugs for its other features, you may want to use the Checker Framework for analyses that can be expressed as pluggable type-checking, such as detecting nullness errors.

Regardless of whether you wish to use the FindBugs nullness analysis, you may continue running all of the other FindBugs analyses at the same time as the Checker Framework; there are no interactions among them.

If FindBugs (or any other tool) discovers a nullness error that the Checker Framework does not, please report it to us (see Section 11.2) so that we can enhance the Checker Framework.

3.5.2 Compatibility note about FindBugs `@Nullable`

FindBugs suppresses all warnings at uses of a `@Nullable` variable. (This inevitably surprises programmers! You have to use `@CheckForNull` to indicate a nullable variable that FindBugs should check.) For example:

```
// declare getObject() to possibly return null
@Nullable Object getObject() { ... }

void myMethod() {
  // FindBugs issues no warning about calling toString on a possibly-null reference
  @Nullable Object o = getObject();
  o.toString();
}
```

The Checker Framework does not emulate this behavior of FindBugs, even if you are using FindBugs annotations. The Checker Framework will issue more warnings than FindBugs, and some of them may be about real bugs in your program. If you wish to suppress warnings at a specific client use where the value is known to be non-null, you should do that at the client use. For example:

```
void myMethod() {
  // Two ways to make the Checker Framework not issue a warning, if
  // the programmer knows this particular invocation won't return null:

  @Nullable Object o1 = getObject();
  assert o1 != null;
  o1.toString();

  @SuppressWarnings("nullness")
  @NonNull o2 = getObject();
  o2.toString();
}
```

4 Interning checker

If the Interning checker issues no warnings for a given program, then all reference equality tests (i.e., “==”) in that program operate on interned types. Interning is a design pattern in which the same object is used whenever two different objects would be considered equal. Interning is also known as canonicalization or hash-consing, and it is

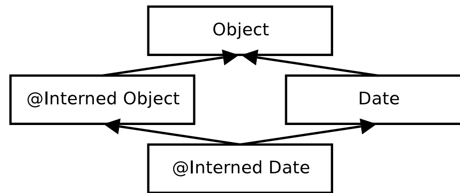

Figure 3: Type hierarchy for the Interning type system.

related to the flyweight design pattern. Interning can save memory and can speed up testing for equality by permitting use of ==; however, use of == on non-interned values can result in subtle bugs. For example:

```
Integer x = new Integer(22);
Integer y = new Integer(22);
System.out.println(x == y);  // prints false!
```

The Interning checker helps programmers to prevent such bugs. The Interning checker also helps to prevent performance problems that result from failure to use interning. (See Section 2.8 for caveats to the checker's guarantees.)

4.1 Interning annotations

Two qualifiers are part of the Interning type system.

`@Interned` indicates a type that includes only interned values (no non-interned values).
`@PolyInterned` indicates qualifier polymorphism. For a description of `@PolyInterned`, see Section 2.5.2.

4.2 Annotating your code with `@Interned`

In order to perform checking, you must annotate your code with the `@Interned` type annotation, which indicates a type for the canonical representation of an object:

```
          String s1 = ...;  // type is (uninterned) "String"
@Interned String s2 = ...;  // Java type is "String", but checker treats it as "Interned String"
```

The type system enforced by the checker plugin ensures that only interned values can be assigned to `s2`.
To specify that *all* objects of a given type are interned, annotate the class declaration:

```
public @Interned class MyInternedClass { ... }
```

This is equivalent to annotating every use of `MyInternedClass`, in a declaration or elsewhere. For example, `enum` classes are implicitly so annotated.

4.2.1 Implicit qualifiers

As described in Section 2.7, the Interning checker adds implicit qualifiers, reducing the number of annotations that must appear in your code. For example, String literals and the null literal are always considered interned, and object creation expressions (using `new`) are never considered `@Interned` unless they are annotated as such, as in

```
@Interned Double internedDoubleZero = new @Interned Double(0); // canonical representation for Double zero
```

For a complete description of all implicit interning qualifiers, see the Javadoc for `InterningAnnotatedTypeFactory`.

4.3 What the Interning checker checks

Objects of an `@Interned` type may be safely compared using the "==" operator.

The checker issues a warning in two cases:

1. When a reference (in)equality operator ("==" or "!=") has an operand of non-`@Interned` type.
2. When a non-`@Interned` type is used where an `@Interned` type is expected.

This example shows both sorts of problems:

```
          Object  obj;
@Interned Object iobj;
...
if (obj == iobj) { ... }  // checker warning: reference equality test is unsafe
iobj = obj;               // checker warning: iobj's referent may no longer be interned
```

The checker also issues a warning when `.equals` is used where == could be safely used. You can disable this behavior via the javac `-Alint` command-line option, like so: `-Alint=-dotequals`.

For a complete description of all checks performed by the checker, see the Javadoc for `InterningVisitor`.

4.4 Examples

To try the Interning checker on a source file that uses the `@Interned` qualifier, use the following command (where `javac` is the JSR 308 compiler that is distributed with the Checker Framework):

```
javac -processor checkers.interning.InterningChecker examples/InterningExample.java
```

Compilation will complete without warnings.

To see the checker warn about incorrect usage of annotations, use the following command:

```
javac -processor checkers.interning.InterningChecker examples/InterningExampleWithWarnings.java
```

The compiler will issue a warning regarding violation of the semantics of `@Interned`.

The Daikon invariant detector (`http://groups.csail.mit.edu/pag/daikon/`) is also annotated with `@Interned`. From directory `java`, run `make check-interning`.

5 IGJ checker

IGJ is a Java language extension that helps programmers to avoid mutation errors (unintended side effects). If the IGJ checker issues no warnings for a given program, then that program will never change objects that should not be changed. This guarantee enables a programmer to detect and prevent mutation-related errors. (See Section 2.8 for caveats to the guarantee.)

5.1 IGJ and Mutability

IGJ permits a programmer to express that a particular object should never be modified via any reference (object immutability), or that a reference should never be used to modify its referent (reference immutability). Once a programmer has expressed these facts, an automatic checker analyzes the code to either locate mutability bugs or to guarantee that the code contains no such bugs.

To learn the details of the IGJ language and type system, please see the ESEC/FSE 2007 paper "Object and reference immutability using Java generics" [ZPA+07]. The IGJ checker supports Annotation IGJ (Section 5.5), which is slightly different dialect of IGJ than that described in the ESEC/FSE paper.

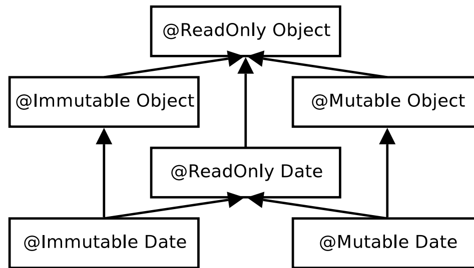

Figure 4: Type hierarchy for three of IGJ's type qualifiers.

5.2 IGJ Annotations

Each object is either immutable (it can never be modified) or mutable (it can be modified). The following qualifiers are part of the IGJ type system.

`@Immutable` An immutable reference always refers to an immutable object. Neither the reference, nor any aliasing reference, may modify the object.

`@Mutable` A mutable reference refers to a mutable object. The reference, or some aliasing mutable reference, may modify the object.

`@ReadOnly` A readonly reference cannot be used to modify its referent. The referent may be an immutable or a mutable object. In other words, it is possible for the referent to change via an aliasing mutable reference, even though the referent cannot be changed via the readonly reference.

`@AssignsFields` is similar to `@Mutable`, but permits only limited mutation — assignment of fields — and is intended for use by constructor helper methods.

`@I` simulates mutability overloading or the template behavior of generics. It can be applied to classes, methods, and parameters. See Section 5.5.3.

For additional details, see [ZPA+07].

5.3 What the IGJ checker checks

The IGJ checker issues an error whenever mutation happens through a readonly reference, when fields of a readonly reference which are not explicitly marked with `@Assignable` are reassigned, or when a readonly expression is assigned to a mutable variable. The checker also emits a warning when casts increase the mutability access of a reference.

5.4 Implicit qualifiers

As described in Section 2.7, the IGJ checker adds implicit qualifiers, reducing the number of annotations that must appear in your code.

For a complete description of all implicit nullness qualifiers, see the Javadoc for `NullnessAnnotatedTypeFactory`.

The default annotation (for types that are unannotated and not given an implicit qualifier) is as follows:

- `@Mutable` for almost all references. This is backward-compatible with Java, since Java permits any reference to be mutated.
- `@Readonly` for local variables. This qualifier may be refined by flow-sensitive local type refinement (see Section 2.7.2).
- `@Readonly` for type parameter and wildcard bounds. For example,

```
interface List<T extends Object> { ... }
```

is defaulted to

```
interface List<T extends @Readonly Object> { ... }
```

This default is not backward-compatible — that is, you may have to explicitly add `@Mutable` annotations to some type parameter bounds in order to make unannotated Java code type-check under IGJ. However, this reduces the number of annotations you must write overall (since most variables of generic type are in fact not modified), and permits more client code to type-check (otherwise a client could not write `List<@Readonly Date>`).

5.5 Annotation IGJ Dialect

The IGJ checker supports the Annotation IGJ dialect of IGJ. The syntax of Annotation IGJ is based on type annotations.

The syntax of the original IGJ dialect [ZPA+07] was based on Java 5's generics and annotation mechanisms. The original IGJ dialect was not backward-compatible with Java (either syntactically or semantically). The dialect of IGJ checked by the IGJ checker corrects these problems.

The differences between the Annotation IGJ dialect and the original IGJ dialect are as follows.

5.5.1 Semantic Changes

- Annotation IGJ does not permit covariant changes in generic type arguments, for backward compatibility with Java. In ordinary Java, types with different generic type arguments, such as `Vector<Integer>` and `Vector<Number>`, have no subtype relationship, even if the arguments (`Integer` and `Number`) do. The original IGJ dialect changed the Java subtyping rules to permit safely varying a type argument covariantly in certain circumstances. For example,

```
Vector<Mutable, Integer> <: Vector<ReadOnly, Integer>
                         <: Vector<ReadOnly, Number>
                         <: Vector<ReadOnly, Object>
```

- Annotation IGJ supports array immutability. The original IGJ dialect did not permit the (im)mutability of array elements to be specified, because the generics syntax used by the original IGJ dialect cannot be applied to array elements.

5.5.2 Syntax Changes

- Immutability is specified through type annotations [Ern08] (Section 5.2), not through a combination of generics and annotations. Use of type annotations makes Annotation IGJ backward compatible with Java syntax.
- Templating over Immutability: The annotation `@I(`*id*`)` is used to template over immutability. See Section 5.5.3.

5.5.3 Templating Over Immutability: @I

`@I` is a template annotation over IGJ Immutability annotations. It acts similarly to type variables in Java's generic types, and the name `@I` mimics the standard `<I>` type variable name used in code written in the original IGJ dialect. The annotation value string is used to distinguish between multiple instances of `@I` — in the generics-based original dialect, these would be expressed as two type variables `<I>` and `<J>`.

Usage on classes A class annotated with `@I` could be declared with any IGJ Immutability annotation. The actual immutability that `@I` is resolved to dictates the immutability type for all the non-static appearances of `@I` with the same value as the class declaration.

Example:

```
@I
public class FileDescriptor {
   private @Immutable Date creationData;
   private @I Date lastModData;
```

```
    public @I Date getLastModDate() @ReadOnly { }
}

...
void useFileDescriptor() {
    @Mutable FileDescriptor file =
                    new @Mutable FileDescriptor(...);
    ...
    @Mutable Data date = file.getLastModDate();

}
```

In the last example, `@I` was resolved to `@Mutable` for the instance file.

Usage on methods For example, it could be used for method parameters, return values, and the actual IGJ immutability value would be resolved based on the method invocation.

For example, the below method `getMidpoint` returns a `Point` with the same immutability type as the passed parameters if `p1` and `p2` match in immutability, otherwise `@I` is resolved to `@ReadOnly`:

```
static @I Point getMidpoint(@I Point p1, @I Point p2) { ... }
```

The `@I` annotation value distinguishes between `@I` declarations. So, the below method `findUnion` returns a collection of the same immutability type as the *first* collection parameter:

```
static <E> @I("First") Collection<E> findUnion(@I("First") Collection<E> col1,
                                               @I("Second") Collection<E> col2) { ... }
```

5.6 Examples

To try the IGJ checker on a source file that uses the IGJ qualifier, use the following command (where `javac` is the JSR 308 compiler that is distributed with the Checker Framework).

```
javac -processor checkers.igj.IGJChecker examples/IGJExample.java
```

The IGJ checker itself is also annotated with IGJ annotations.

6 Javari checker

Javari [TE05, QTE08] is a Java language extension that helps programmers to avoid mutation errors that result from unintended side effects. If the Javari checker issues no warnings for a given program, then that program will never change objects that should not be changed. This guarantee enables a programmer to detect and prevent mutation-related errors. (See Section 2.8 for caveats to the guarantee.) The Javari webpage (`http://groups.csail.mit.edu/pag/javari/`) contains papers that explain the Javari language and type system. By contrast to those papers, the Javari checker uses an annotation-based dialect of the Javari language.

The Javarifier tool infers Javari types for an existing program; see Section 6.2.2.

Also consider the IGJ checker (Section 5). The IGJ type system is more expressive than that of Javari, and the IGJ checker is a bit more robust. However, IGJ lacks a type inference tool such as Javarifier.

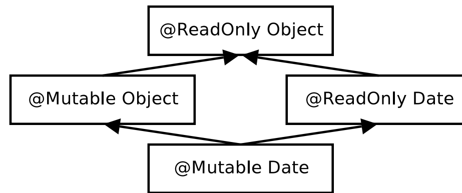

Figure 5: Type hierarchy for Javari's ReadOnly type qualifier.

6.1 Javari annotations

Five annotations are part of the Javari type system.

A programmer can write five annotations: `@ReadOnly`, `@Mutable`, `@Assignable`, `@PolyRead`, and `@QReadOnly`.

`@ReadOnly` indicates a type that provides only read-only access. A reference of this type may not be used to modify its referent, but aliasing references to that object might change it.

`@Mutable` indicates a mutable type.

`@Assignable` is a field annotation, not a type qualifier. It indicates that the given field may always be assigned, no matter what the type of the reference used to access the field.

`@QReadOnly` corresponds to Javari's "`? readonly`" for wildcard types. An example of its use is `List<@QReadOnly Date>`. It allows only the operations which are allowed for both readonly and mutable types.

`@PolyRead` (previously named `@RoMaybe`) specifies polymorphism over mutability; it simulates mutability overloading. It can be applied to methods and parameters. See Section 2.5.2 and the `@PolyRead` Javadoc for more details.

6.2 Writing Javari annotations

6.2.1 Implicit qualifiers

As described in Section 2.7, the Javari checker adds implicit qualifiers, reducing the number of annotations that must appear in your code.

For a complete description of all implicit nullness qualifiers, see the Javadoc for `JavariAnnotatedTypeFactory`.

6.2.2 Inference of Javari annotations

It can be tedious to write annotations in your code. The Javarifier tool (`http://groups.csail.mit.edu/pag/javari/javarifier/`) infers Javari types for an existing program. It automatically inserts Javari annotations in your Java program or in in `.class` files.

This has two benefits: it relieves the programmer of the tedium of writing annotations (though the programmer can always refine the inferred annotations), and it annotates libraries, permitting checking of programs that use those libraries.

6.3 What the Javari checker checks

The checker issues an error whenever mutation happens through a readonly reference, when fields of a readonly reference which are not explicitly marked with `@Assignable` are reassigned, or when a readonly expression is assigned to a mutable variable. The checker also emits a warning when casts increase the mutability access of a reference.

6.4 Examples

To try the Javari checker on a source file that uses the Javari qualifier, use the following command (where `javac` is the JSR 308 compiler that is distributed with the Checker Framework). Alternately, you may specify just one of the test files.

```
javac -processor checkers.javari.JavariChecker tests/javari/*.java
```

The compiler should issue the errors and warnings (if any) specified in the `.out` files with same name.

To run the test suite for the Javari checker, use `ant javari-tests`.

The Javari checker itself is also annotated with Javari annotations.

7 Lock checker

The Lock checker prevents certain kinds of concurrency errors. If the Lock checker issues no warnings for a given program, then the program holds the appropriate lock every time that it accesses a variable.

Note: This does *not* mean that your program has no concurrency errors. (You might have forgotten to annotate that a particular variable should only be accessed when a lock is held. You might released and re-acquire the lock, when correctness requires you to hold it throughout a computation. And, there are other concurrency errors that cannot, or should not, be solved with locks.) However, ensuring that your program program obeys its locking discipline is an easy and effective way to eliminate a common and important class of errors.

7.1 Lock annotations

The Lock checker uses two annotations. One is a type qualifier, and the other is a method annotation.

`@GuardedBy` indicates a type whose value may be accessed only when the given lock is held. See the Javadoc for an explanation of the argument. The lock acquisition and the field access may be arbitrarily far in the future.

`@Holding` is a method annotation (not a type) qualifier. It indicates that when the method is called, the given lock must be held by the caller.

A programmer may use the `@Holding` method annotation in two different ways.

- as specifying a higher-level synchronization protocol
- as being a method summary that simplifies reasoning

Both of these are useful, and the Lock checker supports both. The latter ("summary") use doesn't necessarily introduce a new correctness constraint "must hold this lock when accessing". Rather, it expresses a fact about execution — when execution reaches this point, the following locks are held — that enables people and tools to reason intra- rather than inter-procedurally. The point at which the lock must be held for correctness may come later in the body of the method or in a method it calls.

7.1.1 Relationship to annotations in *Java Concurrency in Practice*

The book *Java Concurrency in Practice* [GPB+06] defines a `@GuardedBy` annotation that is the inspiration for ours. The book's `@GuardedBy` serves two related purposes:

- When applied to a field, it means that the given lock must be held when accessing the field. The lock acquisition and the field access may be arbitrarily far in the future.
- When applied to a method, it means that the given lock must be held by the caller at the time that the method is called — in other words, at the time that execution passes the `@GuardedBy` annotation.

One rationale for reusing the annotation name for both purposes in JCIP is simplicity: there are fewer annotations to learn. Another rationale is that both variables and methods are "members" that can be "accessed"; variables can be accessed by reading or writing them (putfield, getfield), and methods can be accessed by calling them (invokevirtual, invokeinterface). In both cases, `@GuardedBy` creates preconditions for accessing so-annotated members.

The Lock checker renames the method annotation to `@Holding`, and it generalizes the `@GuardedBy` annotation into a type qualifier that can apply not just to a field but to an arbitrary type (including the type of a parameter, return value, local variable, generic type parameter, etc.). This makes the annotations more expressive and also more amenable to automated checking. It also accommodates the distinct (though related) meaning of the two annotations.

8 Basic checker

The Basic checker enforces only subtyping rules. It operates over annotations specified by a user on the command line. Thus, users can create a simple type checker without writing any code beyond definitions of the type qualifier annotations.

The Basic checker can accommodate all of the type system enhancements that can be declaratively specified (see Section 10). This includes type introduction rules (implicit annotations, e.g., literals are implicitly considered `@NonNull`) via the `@ImplicitFor` meta-annotation, and other features such as flow-sensitive type qualifier inference (Section 2.7.2) and qualifier polymorphism (Section 2.5.2).

The Basic checker is also useful to type system designers who wish to experiment with a checker before writing code; the Basic checker demonstrates the functionality that a checker inherits from the Checker Framework.

For type systems that require special checks (e.g., warning about dereferences of possibly-null values), you will need to write code and extend the framework as discussed in Section 10.

8.1 Using the Basic checker

The Basic checker is used in the same way as other checkers (using the `-processor` option; see Section 2), except that it requires an additional annotation processor argument via the standard "`-A`" switch:

- `-Aquals`: this option specifies a comma-no-space-separated list of the fully-qualified class names of the annotations used as qualifiers in the custom type system. It serves the same purpose as the `@TypeQualifiers` annotation used by other checkers (see section 10.5).
  The annotations listed in `-Aquals` must be accessible to the compiler during compilation in the classpath. In other words, they must already be compiled before you run the Basic checker with `javac`; it is not sufficient to supply their source files on the command line.

To suppress a warning issued by the basic checker, use a `@SuppressWarnings` annotation, with the argument being the unqualified, uncapitalized name of any of the annotations passed to `-Aquals`.

8.2 Basic checker example

Consider a hypothetical `Encrypted` type qualifier, which denotes that the representation of an object (such as a `String`, `CharSequence`, or `byte[]`) is encrypted. To use the Basic checker for the `Encrypted` type system, follow three steps.

1. Define an annotation for the `Encrypted` qualifier:

```
package myquals;

/**
 * Denotes that the representation of an object is encrypted.
 * ...
 */
@TypeQualifier
@SubtypeOf(Unqualified.class)
public @interface Encrypted {}
```

2. Write `@Encrypted` annotations in your program:

```
public @Encrypted String encrypt(String text) {
    // ...
}

// Only send encrypted data!
public void sendOverInternet(@Encrypted String msg) {
    // ...
```

```
}

void sendText() {
    // ...
    @Encrypted String ciphertext = encrypt(plaintext);
    sendOverInternet(ciphertext);
    // ...
}

void sendPassword() {
    String password = getUserPassword();
    sendOverInternet(password);
}
```

You may also need to add `@SuppressWarnings` annotations to the `encrypt` and `decrypt` methods.

3. Invoke the compiler with the Basic checker, specifying the `@Encrypted` annotation using the `-Aquals` option:

```
$ javac -processor checkers.basic.BasicChecker -Aquals=myquals.Encrypted YourProgram.java

YourProgram.java:42: incompatible types.
found   : java.lang.String
required: @myquals.Encrypted java.lang.String
    sendOverInternet(password);
                     ^
```

9 Annotating libraries

When annotated code uses an unannotated library, a checker may issue warnings. As described in Section 2.3, the best way to correct this problem is to add annotations to the library (though you can instead suppress all warnings related to an unannotated library by use of the `-AskipClasses` command-line option). This section tells you how to add annotations to a library for which you have no source code, because the library is library distributed only binary (`.class` or `.jar`) form.

The Checker Framework distribution contains annotations for popular libraries, such as the JDK. If you annotate additional libraries, please share them with us so that we can distribute the annotations with the Checker Framework; see Section 11.2.

You can determine the correct annotations for a library either automatically by running an inference tool, or manually by reading the documentation. Presently, type inference tools are available for the Nullness (Section 3.2.3) and Javari (Section 6.2.2) type systems.

You can make the annotations known to the JSR 308 compiler (and thus to the checkers) in two ways.

- You can use the stub class generation tool to create a "stub file" file containing classes with no method bodies, and annotate the stub classes file. Then, you can supply the stub files to the checker when compiling/checking your program. Section 9.1 describes how to use the stub class generation tools.
- You can annotate the compiled `.jar` or `.class` files using the annotation file utilities (`http://groups.csail.mit.edu/pag/jsr308/annotation-file-utilities/`). First, express the annotations textually as an annotation index file, and then the tools insert them in the compiled library class files. See the Annotation File Utilities documentation for full details.

9.1 Using stub classes

A stub file contains "stub classes" that contain annotated signatures. A checker uses those annotated signatures at compile time, instead of or in addition to annotations that appear in the library.

Section 9.1.1 describes how to create stub classes. Section 9.1.2 describes how to use stub classes. These sections illustrate stub classes via the example of creating a `@Interned`-annotated version of `java.lang.String`. (You don't need to repeat these steps, since such a stub class is already included in the Checker Framework distribution; see file `checkers/src/checkers/interning/jdk.astub`, which is reproduced in Section 9.1.3.)

9.1.1 Creating a stub file

1. Create a stub file by running the stub class generator. (`checkers.jar` must be on your classpath.)

   ```
   cd nullness-stub
   java checkers.util.stub.StubGenerator java.lang.String > String.astub
   ```

   Supply it with the fully-qualified name of the class for which you wish to generate a stub class. The stub class generator prints the stub class to standard out, so you may wish to redirect its output to a file.
2. Add import statements for the annotations. So you would need to add the following import statement at the beginning of the file:

   ```
   import checkers.interning.quals.Interned;
   ```
3. Add annotations to the stub class. For example, you might annotate the `String.intern()` method as follows:

   ```
   @Interned String intern();
   ```

   You may also remove irrelevant parts of the stub file; see Section 9.1.3.

9.1.2 Using a stub file

When you run `javac` with a given checker/processor, you can specify a list of the stub files or directories using `-Astubs=`*`file_or_path_name`*. The stub path entries are delimited by `File.pathSeparator` ('`:`' for Linux and Mac, '`;`' for Windows). When you supply a stub directory, the checker only considers the enclosed stub files whose names end with `.astub`.

The `-Astubs` argument causes the Checker Framework to read annotations from annotated stub classes in preference to the unannotated original library classes.

```
javac -processor checkers.interning.InterningChecker -Astubs=String.astub:stubs MyFile.java MyOtherFile.java ...
```

9.1.3 Stub file format

The stub file format is designed for simplicity, readability, and compactness. It reads like a Java file but contains only the necessary information for type checking.

As an illustration, the stub file for the Interning type system (Section 4) is as follows. This file appears as `checkers/src/checkers/interning/jdk.astub` in the Checker Framework distribution.

```
import checkers.interning.quals.Interned;

package java.lang;

// All instances of Class are interned.
@Interned class Class<T> { }

class String {
  // The only interning-related method in the JDK.
  @Interned String intern();
}
```

You can use a regular Java file as a stub file. However, you can omit information that is not relevant to pluggable type-checking; this makes the stub file smaller and easier for people to read and write. You can also put annotated signatures for multiple classes in a single stub file.

The stub file format differs from Java source code in the following ways:

Method bodies: The stub class does not require method bodies for classes; any method body may be replaced by a semicolon (`;`), as in an interface or abstract method declaration.

Method declarations: You only have to specify the methods that you need to annotate. Any method declaration may be omitted, in which case the checker reads its annotations from the library. (If you are using a stub class, then typically the library's version is unannotated.)

Non-type-related specifiers: Non-type-related Java specifiers (e.g., `public`, `final`, `volatile`) may be omitted.

Import statements: The only required import statements are the ones to import type annotations. Such imports must be at the beginning of the file. Other import statements are optional.

Multiple classes and packages: The stub file format permits having multiple classes and packages. The packages are separated by a package statement: `package my.package;`. Each package declaration may occur only once; in other words, all classes from a package must appear together.

9.1.4 Known problems

The Checker Framework stub file reader has several limitations:

- It does not handle `enum`s.
- It does not handle non-type annotations (e.g. IGJ's `Assignable`).
- It does not handle arrays yet.

9.2 Using skeleton files (distributed annotated JDKs)

The Checker Framework distribution contains annotated JDKs at the path `checkers/jdk/[checker-name]/src`. These are in another format called "skeleton classes". We are currently working on converting the skeleton files into stub files.

1. When you run `javac`, add a `-sourcepath` argument to indicate where to find the skeleton classes. Supply `-sourcepath` in addition to whatever other arguments you usually use, including `-classpath`.
   The `-sourcepath` argument causes the compiler to read annotations from annotated skeleton classes in preference to the unannotated original library classes. However, the compiler will use the originals on the classpath if no file is available on the sourcepath.
   ```
   javac -processor checkers.nullness.NullnessChecker -sourcepath checkers/jdk/nullness/src my_source_files
   ```
2. Run the compiled code as usual. Do *not* include the skeleton files on the classpath. If a skeleton method is called instead of the true library method, then your program will throw a `RuntimeException`.

10 How to create a new checker

This section describes how to extend the Checker Framework to create a checker — a type-checking compiler plugin that detects bugs or verifies their absence. After a programmer annotates a program, the checker plugin verifies that the code is consistent with the annotations. If you only want to *use* a checker, you do not need to read this section.

Writing a simple checker is easy! For example, here is a complete, useful type checker:

```
@TypeQualifier
@SubtypeOf(Unqualified.class)
public @interface Encrypted {}
```

This checker is so short because it builds on the Basic Checker (Section 8). See Section 8.2 for more details about this particular checker.

You can also customize a typestate checker. Two of these are available. One is by Adam Warski: `http://www.warski.org/typestate.html`. The other is by Daniel Wand: `http://typestate.ewand.de/`.

The rest of this section contains many details for people who want to more write powerful checkers. You won't need all of the details, at least at first. In addition to reading this section of the manual, you may find it helpful

to examine the implementations of the checkers that are distributed with the Checker Framework, or to create your checker by modifying another one. The Javadoc documentation of the framework and the checkers is in the distribution and is also available online at `http://pag.csail.mit.edu/jsr308/current/doc/`.

If you write a new checker, let us know so we can mention it here, link to it from our webpages, or include it in the Checker Framework distribution.

10.1 The parts of a checker

The Checker Framework provides abstract base classes (default implementations), and a specific checker overrides as little or as much of the default implementations as necessary. Sections 10.2–10.5 describe the components of a type system as written using the Checker Framework:

10.2 **Type qualifiers and hierarchy.** You define the annotations for the type system and the subtyping relationships among qualified types (for instance, that `@NonNull Object` is a subtype of `@Nullable Object`).

10.3 **Type introduction rules.** For some types and expressions, a qualifier should be treated as present even if a programmer did not explicitly write it. For example, in the Nullness type system every literal other than `null` has a `@NonNull` type; examples of literals include `"some string"` and `java.util.Date.class`.

10.4 **Type rules.** You specify the the type system semantics (type rules), violation of which yields a type error. There are two types of rules. Your checker automatically inherits rules related to the type hierarchy, such as that every assignment and pseudo-assignment satisfies a subtyping relationship. You write any additional rules. For example, in the Nullness type system, only references with a `@NonNull` type may be dereferenced.

10.5 **Interface to the compiler.** The compiler interface indicates which annotations are part of the type system, which command-line options and `@SuppressWarnings` annotations the checker recognizes, etc.

10.2 Annotations: Type qualifiers and hierarchy

A type system designer specifies the qualifiers in the type system and the type hierarchy that relates them.

Type qualifiers are defined as Java annotations [Dar06]. In Java, an annotation is defined using the Java `@interface` keyword. Write the `@TypeQualifier` annotation on the annotation definition to indicate that the annotation represents a type qualifier (e.g., `@NonNull` or `@Interned`) and should be processed by the checker. For example:

```
// Define an annotation for the @NonNull type qualifier.
@TypeQualifier
public @interface NonNull { }
```

(An annotation that is written on an annotation definition, such as `@TypeQualifier`, is called a *meta-annotation*.)

The type hierarchy induced by the qualifiers can be defined either declaratively via meta-annotations (Section 10.2.1), or procedurally through subclassing `QualifierHierarchy` or `TypeHierarchy` (Section 10.2.2).

10.2.1 Declaratively defining the qualifier and type hierarchy

Declaratively, the type system designer uses two meta-annotations (written on the declaration of qualifier annotations) to specify the qualifier hierarchy.

- `@SubtypeOf` denotes that a qualifier is the subtype of another qualifier or qualifiers, specified as an array of class literals. For example, for any type T, `@NonNull` T is a subtype of `@Nullable` T:

  ```
  @TypeQualifier
  @SubtypeOf( { Nullable.class } )
  public @interface NonNull { }
  ```

  (The actual definition of `NonNull` is slightly more complex.)

  `@SubtypeOf` accepts multiple annotation classes as an argument, permitting the type hierarchy to be an arbitrary DAG. For example, in the IGJ type system (Section 5.2), `@Mutable` and `@Immutable` induce two mutually exclusive subtypes of the `@ReadOnly` qualifier.

As a special case, the root qualifier needs to be annotated with `@Subtype( { } )`. The root qualifier is the qualifier that is a supertype of all other qualifiers. `Nullable` is the root of the Nullness type system, hence is defined as:

```
@TypeQualifier
@SubtypeOf( { } )
public @interface Nullable { }
```

All type qualifiers, except for polymorphic qualifiers, need to be properly annotated with `SubtypeOf`.
If the root of the hierarchy is the unqualified type, then its children will use `@SubtypeOf(Unqualified.class)`, but no `@SubtypeOf( { } )` annotation on the root is necessary. For an example, see the `Encrypted` type system of Section 10.

- `@PolymorphicQualifier` denotes that a qualifier is a polymorphic qualifier. For example:

```
@TypeQualifier
@PolymorphicQualifier
public @interface PolyNull { }
```

For a description of polymorphic qualifiers, see Section 2.5.2. A polymorphic qualifier needs no `@SubtypeOf` meta-annotation and need not be mentioned in any other `@SubtypeOf` meta-annotation.

The declarative and procedural mechanisms for specifying the hierarchy can be used together. In particular, when using the `@SubtypeOf` meta-annotation, further customizations may be performed procedurally (Section 10.2.2) by overriding the `isSubtype` method in the checker class (Section 10.5). However, the declarative mechanism is sufficient for most type systems.

10.2.2 Procedurally defining the qualifier and type hierarchy

While the declarative syntax suffices for many cases, more complex type hierarchies can be expressed by overriding, in `BaseTypeChecker`, either `createQualifierHierarchy` or `createTypeHierarchy` (typically only one of these needs to be overridden). For more details, see the Javadoc of those methods and of the classes `QualifierHierarchy` and `TypeHierarchy`.

The `QualifierHierarchy` class represents the qualifier hierarchy (not the type hierarchy), e.g., `Mutable` is a subtype of `ReadOnly`. A type-system designer may subclass `QualifierHierarchy` to express customized qualifier relationships (e.g., relationships based on annotation arguments).

The `TypeHierarchy` class represents relationships between annotated types, rather than merely type qualifiers, e.g., `@Mutable Date` is a subtype of `@ReadOnly Date`. The default `TypeHierarchy` uses `QualifierHierarchy` to determine all subtyping relationships. The default `TypeHierarchy` handles generic type arguments, array components, type variables, and wild-cards in a similar manner to the Java standard subtype relationship but with taking qualifiers into consideration. Some type systems may need to override that behavior. For instance, the Java Language Specification specifies that two generic types are subtypes only if their type arguments are identical: for example, `List<Date>` is not a subtype of `List<Object>`, or of any other generic `List`. (In the technical jargon, the generic arguments are “invariant”.) The Javari type system overrides this behavior to allow some type arguments to change covariantly in a type-safe manner (e.g., `List<@Mutable Date>` is a subtype of `List<@QReadOnly Date>`).

10.2.3 Defining a default annotation

A type system designer may set a default annotation. A user may override the default; see Section 2.7.1.

The type system designer may specify a default annotation declaratively, using the `@DefaultQualifierInHierarchy` meta-annotation. Note that the default will apply to any source code that the checker reads, including stub libraries, but will not apply to compiled `.class` files that the checker reads.

Alternately, the type system designer may specify a default procedurally, by calling the `QualifierDefaults.setAbsoluteDefault` method. You may do this even if you have declaratively defined the qualifier hierarchy; see the Nullness checker’s implementation for an example.

Recall that defaults are distinct from implicit annotations; see Sections 2.7 and 10.3.

10.2.4 Bottom qualifier

It is usually a good idea to have a bottom qualifier in your type hierarchy — a qualifier that is a (direct or indirect) subtype of every other qualifier. The reason is that this is the natural type for the `null` value, which is can be viewed as having any type at all.

Users should never write the bottom qualifier explicitly; it is merely used for the `null` value.

You might write the bottom qualifier like this:

```
package myTypeQuals;

import checkers.quals.*;
import com.sun.source.tree.Tree;

@TypeQualifier
@SubtypeOf({Prototype.class, NonPrototype.class})
@ImplicitFor(trees={Tree.Kind.NULL_LITERAL})
public @interface PrototypeBottom {}
```

10.3 Type Factory: Implicit annotations

For some types and expressions, a qualifier should be treated as present even if a programmer did not explicitly write it. For example, every literal (other than `null`) has a `@NonNull` type.

The implicit annotations may be specified declaratively and/or procedurally.

10.3.1 Declaratively specifying implicit annotations

The `@ImplicitFor` meta-annotation indicates implicit annotations. When written on a qualifier, `ImplicitFor` specifies the trees (AST nodes) and types for which the framework should automatically add that qualifier.

In short, the types and trees can be specified via any combination of five fields:

- `trees`: an array of `com.sun.source.tree.Tree.Kind`, e.g., `NEW_ARRAY` or `METHOD_INVOCATION`
- `types`: an array of `TypeKind`, e.g., `ARRAY` or `BOOLEAN`
- `treeClasses`: an array of class literals for classes implementing `Tree`, e.g., `LiteralTree.class` or `ExpressionTree.class`
- `typeClasses`: an array of class literals for classes implementing `javax.lang.model.type.TypeMirror`, e.g., `javax.lang.model.type.PrimitiveType`. Often you should use a subclass of `AnnotatedTypeMirror`
- `stringPatterns`: an array of regular expressions that will be matched against string literals, e.g., `"[01]+"` for a binary number. Useful for annotations that indicate the format of a string.

For example, consider the definitions of the `@NonNull` and `@Nullable` type qualifiers:

```
@TypeQualifier
@SubtypeOf( { Nullable.class } )
@ImplicitFor(
  types={TypeKind.PACKAGE},
  typeClasses={AnnotatedPrimitiveType.class},
  trees={
    Tree.Kind.NEW_CLASS,
    Tree.Kind.NEW_ARRAY,
    Tree.Kind.PLUS,
    // All literals except NULL_LITERAL:
    Tree.Kind.BOOLEAN_LITERAL, Tree.Kind.CHAR_LITERAL, Tree.Kind.DOUBLE_LITERAL, Tree.Kind.FLOAT_LITERAL,
    Tree.Kind.INT_LITERAL, Tree.Kind.LONG_LITERAL, Tree.Kind.STRING_LITERAL
  })
public @interface NonNull {  }


@TypeQualifier
```

```
@SubtypeOf({})
@ImplicitFor(trees={Tree.Kind.NULL_LITERAL})
public @interface Nullable { }
```

For more details, see the Javadoc for the `ImplicitFor` annotation, and the Javadoc for the javac classes that are linked from it. (You only need to understand a small amount about the javac AST, such as the `Tree.Kind` and `TypeKind` enums. All the information you need is in the Javadoc, and Section 10.8 can help you get started.)

10.3.2 Procedurally specifying implicit annotations

The Checker Framework provides a representation of annotated types, `AnnotatedTypeMirror`, that extends the standard `TypeMirror` interface but integrates a representation of the annotations into a type representation. A checker's *type factory* class, given an AST node, returns the annotated type of that expression. The Checker Framework's abstract *base type factory* class, `AnnotatedTypeFactory`, supplies a uniform, Tree-API-based interface for querying the annotations on a program element, regardless of whether that element is declared in a source file or in a class file. It also handles default annotations, and it optionally performs flow-sensitive local type inference.

`AnnotatedTypeFactory` inserts the qualifiers that the programmer explicitly inserted in the code. Yet, certain constructs should be treated as having a type qualifier even when the programmer has not written one. The type system designer may subclass `AnnotatedTypeFactory` and override `annotateImplicit(Tree,AnnotatedTypeMirror)` and `annotateImplicit(Element,AnnotatedTypeMirror)` to account for such constructs.

10.4 Visitor: Type rules

A type system's rules define which operations on values of a particular type are forbidden.

The framework provides a *base visitor class*, `BaseTypeVisitor`, that performs type-checking at each node of a source file's AST. It uses the visitor design pattern to traverse Java syntax trees as provided by Sun's Tree API, and issues a warning whenever the type system induced by the type qualifier is violated.

A checker's visitor overrides one method in the base visitor for each special rule in the type qualifier system. Most type-checkers override only a few methods in `BaseTypeVisitor`. For example, the visitor for the Nullness type system of Section 3 consists of a single 4-line method that warns if an expression of nullable type is dereferenced, as in:

```
myObject.hashCode();  // invalid dereference
```

By default, `BaseTypeVisitor` performs subtyping checks that are similar to Java subtype rules, but taking the type qualifiers into account. `BaseTypeVisitor` issues these errors:

- invalid assignment (type.incompatible) when an assignment from an expression type to an incompatible type. The assignment may be a simple assignment, or pseudo-assignment like return expressions or argument passing in a method invocation
  In particular, in every assignment and pseudo-assignment, the left-hand side of the assignment is a supertype of (or the same type as) the right-hand side. For example, this assignment is not permitted:
  ```
  @Nullable Object myObject;
  @NonNull Object myNonNullObject;
  ...
  myNonNullObject = myObject;  // invalid assignment
  ```
- invalid generic argument (generic.argument.invalid) when a type is bound to an incompatible generic type variable
- invalid method invocation (method.invocation.invalid) when a method is invoked on an object whose type is incompatible with the method receiver type
- invalid overriding parameter type (override.parameter.invalid) when a parameter in a method declaration is incompatible with that parameter in the overridden method's declaration
- invalid overriding return type (override.return.invalid) when a parameter in a method declaration is incompatible with that parameter in the overridden method's declaration

- invalid overriding receiver type (override.receiver.invalid) when a receiver in a method declaration is incompatible with that receiver in the overridden method's declaration

10.5 The checker class: Compiler interface

A checker's entry point is a subclass of `BaseTypeChecker`. This entry point, which we call the checker class, serves two roles: an interface to the compiler and a factory for constructing type-system classes.

Because the Checker Framework provides reasonable defaults, oftentimes the checker class has no work to do. Here are the complete definitions of the checker classes for the Interning and Nullness checkers:

```
@TypeQualifiers({ Interned.class, PolyInterned.class })
@SupportedLintOptions({"dotequals"})
public final class InterningChecker extends BaseTypeChecker { }

@TypeQualifiers({ Nullable.class, Raw.class, NonNull.class, PolyNull.class })
@SupportedLintOptions({"flow", "cast", "cast:redundant"})
public class NullnessChecker extends BaseTypeChecker { }
```

The checker class must be annotated by `@TypeQualifiers`, which lists the annotations that make up the type hierarchy for this checker (including polymorphic qualifiers), provided as an array of class literals. Each one is a type qualifier whose definition bears the `@TypeQualifier` meta-annotation (or is returned by the `BaseTypeChecker-.getSupportedTypeQualifiers` method).

The checker class bridges between the compiler and the checker plugin. It invokes the type-rule check visitor on every Java source file being compiler, and provides a simple API, `report`, to issue errors using the compiler error reporting mechanism.

Also, the checker class follows the factory method pattern to construct the concrete classes (e.g., visitor, factory) and annotation hierarchy representation. It is a convention that, for a type system named Foo, the compiler interface (checker), the visitor, and the annotated type factory are named as `FooChecker`, `FooVisitor`, and `FooAnnotatedTypeFactory`. `BaseTypeChecker` uses the convention to reflectively construct the components. Otherwise, the checker writer must specify the component classes for construction.

A checker can customize the default error messages through a `Properties`-loadable text file named `messages.properties` that appears in the same directory as the checker class. The property file keys are the strings passed to `report` (like `type.incompatible`) and the values are the strings to be printed (`cannot assign ...`). The `messages.properties` file only need to mention the new messages that the checker defines. It is also allowed to override messages defined in superclasses, but this is rarely needed.

10.5.1 Bundling multiple checkers

Users need to specify the checker class name in command line `-processor` flag to invoke each checker. When multiple related checkers need to be run together as a unit, users will have to pass each checker class name, like:

```
javac -processor DistanceUnitChecker -processor SpeedUnitChecker ... files ...
```

Alternatively, an aggregate checker class is declared to combine these multiple checkers. `AggregateChecker` forms a convenient base class for such situation, where the checkers can be declared in one method, like the following:

```
public class UnitCheckers extends AggregateChecker {
  protected abstract Collection<Class<? extends SourceChecker>>
  getSupportedCheckers() {
    return Arrays.asList(DistanceUnitChecker.class, SpeedUnitChecker);
  }
}
```

Now, users can simply pass `UnitCheckers` a single argument to the commandline:

```
javac -processor UnitCheckers ... files ...
```

10.6 Testing framework

[This section should discuss the testing framework that is used for testing the distributed checkers.]

10.7 Debugging options

The Checker Framework provides debugging options that can be helpful when writing checker. These are provided via the standard `javac` "`-A`" switch, which is used to pass options to an annotation processor.

- `-Anomsgtext`: use message keys (such as "`type.invalid`") rather than full message text when reporting errors or warnings
- `-Ashowchecks`: print debugging information for each pseudo-assignment check (as performed by `BaseTypeVisitor`; see Section 10.4 above)
- `-Afilenames`: prints the name of each file before type-checking it

The following example demonstrates how these options are used:

```
$ javac -processor checkers.interning.InterningChecker \
    examples/InternedExampleWithWarnings.java -Ashowchecks -Anomsgtext -Afilenames

[InterningChecker] InterningExampleWithWarnings.java
 success (line  18): STRING_LITERAL "foo"
     actual: DECLARED @checkers.interning.quals.Interned java.lang.String
   expected: DECLARED @checkers.interning.quals.Interned java.lang.String
 success (line  19): NEW_CLASS new String("bar")
     actual: DECLARED java.lang.String
   expected: DECLARED java.lang.String
examples/InterningExampleWithWarnings.java:21: (not.interned)
    if (foo == bar)
            ^
 success (line  22): STRING_LITERAL "foo == bar"
     actual: DECLARED @checkers.interning.quals.Interned java.lang.String
   expected: DECLARED java.lang.String
1 error
```

You can use any standard debugger to observe the execution of your checker. Set the main class to `com.sun.tools.javac.Main` and the bootclasspath to include the JSR308 langtools.

10.8 javac implementation survival guide

The implementation of Sun's javac compiler can be a bit daunting to a newcomer, and its documentation does not particularly help a newcomer to get oriented. But do not lose heart! This section helps you to understand the small part of javac that you need in order to write a checker. Other useful resources include the Java Infrastructure Developer's guide at `http://wiki.netbeans.org/Java_DevelopersGuide` and the compiler mailing list archives at `http://news.gmane.org/gmane.comp.java.openjdk.compiler.devel` (subscribe at `http://mail.openjdk.java.net/mailman/listinfo/compiler-dev`).

The Checker Framework uses Sun's Tree API to access a program's AST. This is specific to the Sun JDK. In the future, the Checker Framework can be migrated to use the Java Model AST of JSR 198 (Extension API for Integrated Development Environments) [Cro06], which gives access to the entire source code of a method in an implementation-neutral way.

A `Tree` is an AST node; it represents an arbitrary code snippet such as a method definition, a block, a statement, etc.

The `Tree` interface has many subinterfaces, that specify what kind of node is being handled. Trees are usually processed by a class implementing the `TreeVisitor` interface, through the `accept` method on `Tree`. Common implementations of `TreeVisitor` that you may want to extend are `SimpleTreeVisitor`, that visits a single node based on its type, `TreeScanner`, that visits all subnodes recursively, and `TreePathScanner`, that visits all subnodes recursively and stores the `TreePath` corresponding the the currently visited `Tree`. (Also note that the iterator given by `TreePath` used to have an implementation bug.)

In order to determine the kind of an object that extends Tree, use the getKind method, as opposed to the instanceof operator, since a Tree implementation might opt to implement more than one interface from this API. There is a utility class to perform operations on trees, Trees, but the framework is intended to do all the low-level tree processing, so you probably should not need to use this class.

An Element represents a program element such as packages, classes or methods. Element has 5 subinterfaces: ExecutableElement represents methods, constructors or initializers (anything invocable); PackageElement represents package elements, and contain package information; TypeElement represents the element of a class or an interface (note that TypeElement is an Element, not a Type; the corresponding Type is represented by DeclaredType; TypeParameterElement represents an element of a formal type parameter of a something with generics, and VariableElement represents the element associated with a variable. There is an ElementVisitor interface for visiting objects that Element, in a similar manner to the Tree visitors, with similar provided implementations. Use the asType method from Element to obtain a TypeMirror for the element.

Again, Element is an interface, so use getKind() to obtain the kind of an Element, as opposed to the instanceof operator, since an implementation of Element might also implement other element interfaces. There is an utility class for handling elements, Elements; the appropriate instance can be obtained by using the getElementUtils method on the ProcessingEnvironment object visible on factories and checkers. The framework should do most of the element processing that requires Elements, unless you are doing something non-trivial.

A TypeMirror represents a Java type. It is yet another interface you should be familiar with, with various subinterfaces, notable ones being DeclaredType for class and interface types, and ExecutableType for method, constructor and initializer types.

Note that a MethodTree resolves into a ExecutableType, while a MethodInvocationTree resolves into a DeclaredType if the return type is a class or an interface, an ArrayType if the return type is an array, a NoType if the return type is void, or a PrimitiveType if the return type is primitive.

Not every Tree corresponds to an Element (such as a BlockTree), not every Tree corresponds to a TypeMirror (again, such as a BlockTree), and not every TypeMirror has a corresponding Element (such as primitive types or arrays).

As one could expect by this point, TypeMirror is an interface, so use the appropriate getKind() method to distinguish the types, as opposed to the instanceof operator, since those are interfaces, and more than one can be implemented by a same object.

Note that the TypeMirror API makes no guarantees that the same type will always be represented by the same object; use the method recommended on the API if you need to compare two types.

TypeVisitor and implementations of visitors for TypeMirror are provided, but those classes should not be used or extended directly on the framework, since all checker plugin classes are meant to visit AnnotatedTypeMirror instead, modifying the annotations as needed. A Types utility class is provided by the ProcessingEnvironment as well, if you need to do more complex operations with types. In general, you should use AnnotatedTypeMirror and its subclasses as opposed to using TypeMirror and its subinterfaces.

An AnnotatedTypeMirror (defined in the Checker Framework, not in javac) represents an annotated type — a type along with all its annotations. It is modeled after Sun's TypeMirror. Similarly modeled visitors are presented: a AnnotatedTypeVisitor interface, implemented by SimpleAnnotatedTypeVisitor for visiting just one node, AnnotatedTypeScanner for visiting every node recursively.

In short: a Tree represents some snippet of code, an Element represents some program element, and a TypeMirror represents a Java type, but you usually should use AnnotatedTypeMirror, provided by the Checker Framework, instead of TypeMirror, as our implementation carries along with the types the annotation information at every node level. The AnnotatedTypeFactory (or its extension on your framework plugin) is responsible for producing AnnotatedTypeMirror objects for Tree and Element parameters it receives; those AnnotatedTypeMirror objects are then processed by the visitor class and checked by the checker class on your checker plugin.

10.9 When to use (and not use) type qualifiers

For some programming tasks, you can use either a Java subclass or a type qualifier. For instance, suppose that your code currently uses String to represent an address. You could create a new Address class and refactor your code

to use it, or you could create a @Address annotation and apply it to some uses of String in your code. If both of these are truly possible, then it is probably more foolproof to use the Java class. We do not encourage you to use type qualifiers as a poor substitute for classes. However, sometimes type qualifiers are a better choice.

Using a new class may your code incompatible with existing libraries or clients. Brian Goetz expands on this issues in an article on the pseudo-typedef antipattern [Goe06]. Even if compatibility is not a concern, a code change may introduce bugs, whereas adding annotations does not change the run-time behavior. It is possible to add annotations to existing code, including code you do not maintain or cannot change. It is possible to annotate primitive types without converting them to wrappers, which would make the code both uglier and slower.

Type qualifiers can be applied to any type, including final classes that cannot be subclassed.

Type qualifiers permit you to remove operations, with a compile-time guarantee. An example is mutating methods that are forbidden by immutable types (see Sections 5 and 5). More generally, type qualifiers permit creating a new supertype, not just a subtype, of an existing Java type.

The least important reason is efficiency. Type qualifiers can be more efficient, since there is no no run-time representation such as a wrapper or a separate class, nor introduction of dynamic dispatch for methods that could otherwise be statically dispatched.

11 Troubleshooting and getting help

11.1 Common problems and solutions

To verify that you are using the compiler you think you are, you can add -version to the command line. For instance, instead of running javac -g MyFile.java, you can run javac -version -g MyFile.java. Then, javac will print out its version number in addition to doing its normal processing.

- If you get the error

  ```
  com.sun.tools.javac.code.Symbol$CompletionFailure: class file for com.sun.source.tree.Tree not found
  ```

  then you are using the source installation and file tools.jar is not on your classpath. See the installation instructions (Section 1.2).
- If you get an error such as

  ```
  package checkers.nullness.quals does not exist
  ```

  despite no apparent use of import checkers.nullness.quals.*; in the source code, then perhaps jsr308_imports is set as a Java system property, a shell environment variable, or a command-line option (see Section 2.9.2). You can solve this by unsetting the variable/option, or by ensuring that the checkers.jar file is on your classpath.
- If a checker seems to be ignoring the annotation on a method, then it is possible that the checker is reading the method's signature from its .class file, but the .class file was not created by the JSR 308 compiler. You can check whether the annotations actually appear in the .class file by using the javap tool.
  If the annotations do not appear in the .class file, here are two ways to solve the problem:
  - Re-compile the method's class with the JSR 308 compiler. This will ensure that the type annotations are written to the class file, even if no type-checking happens during that execution.
  - Pass the method's file explicitly on the command line when type-checking, so that the compiler reads its source code instead of its .class file.
- If the compiler reports that it cannot find a method that appears in the JDK or another external library, then maybe the skeleton file for that class is incomplete. You can edit it to add the missing method. The libraries appear, for example, at checkers/jdk/nullness/src/ for the Nullness checker.
  The error might take one of these forms:

  ```
  method sleep in class Thread cannot be applied to given types
  cannot find symbol: constructor StringBuffer(StringBuffer)
  ```

11.1.1 Known problems in the framework

- The framework may not parse annotations from skeleton files if the skeleton files are older than the classfiles. Running `ant touch-jdk` solves this problem, by applying the `touch` program to each distributed skeleton file.
- The framework is missing a check for type argument subtyping in method invocations if the type arguments are inferred.
- The checks for enclosed types are not yet fully tested.

11.1.2 Known problems in the Nullness checker

- The Nullness checker is often able to determine that a call to `Map.get()` will not return null. This enables the checker to avoid issuing false positive warnings, in circumstances like the following.

```
@NonNull String value;
if (myMap.containsKey(key)) {
  value = myMap.get(key);
}
for (String keyInMap : myMap.keySet()) {
    value = myMap.get(keyInMap);
}
```

  The Nullness checker can sometimes fail to issue a warning if the map is modified or re-assigned between the check of `containsKey` and the call to `get`.
- The Nullness checker issues a warning when a constructor does not initialize every non-null field. However, because the checker does not fully implement all of Java's definite assignment rules (e.g., for `finally` blocks), the checker sometimes issues a false positive warning. The checker's behavior is sound but unnecessarily restrictive. If you encounter this problem in practice, please submit a bug report so that we can improve the checker.

11.2 How to report problems

If you have a problem with any checker, or with the Checker Framework, please let us know at `jsr308-bugs@lists.csail.mit.edu`. In addition to bug reports, we welcome suggestions, annotated libraries, bug fixes, new features, new checker plugins, and other improvements.

Please ensure that your bug report is clear and that it is complete. Otherwise, we may be unable to understand it or to reproduce it, either of which would prevent us from fixing the bug. Your bug report will be most helpful if you:

- Add `-verbose` to the javac options. This causes the compiler to output a lot of debugging information.
- Indicate exactly what you did. Show the exact commands (don't merely describe them in words). Don't skip any steps.
- Include all files that are necessary to reproduce the problem. This includes every file that is used by any of the commands you reported, and possibly other files as well.
- Indicate exactly what the result was (don't merely describe it in words). Also indicate what you expected the result to be — remember, a bug is a difference between desired and actual outcomes.
- Indicate which version of the JSR 308 compiler and Checker Framework you are using. You can determine the compiler version by running `javac -version`.

11.3 Installing the source release

The binary release (Section 1.2) contains everything that most users need, both to use the distributed checkers and to write your own checkers. This section describes how to install the source release. Doing so permits you to examine and modify the implementation of the distributed checkers and of the checker framework. It may also help you to debug problems more effectively.

11.3.1 The short instructions (for Linux only)

The following commands install the JSR 308 javac compiler and the Checker Framework, or update an existing installation. It currently works only on **Linux**. For more details, or if anything goes wrong, see the comments in the Makefile-jsr308-install file.

1. Execute the following commands:

```
cd
wget -nv -N http://groups.csail.mit.edu/pag/jsr308/current/Makefile-jsr308-install
make -f Makefile-jsr308-install
```

2. Set some environment variables according to the instructions at the top of file Makefile-jsr308-install.

11.3.2 The longer instructions

The following instructions give detailed steps for installing the source release of the Checker Framework.

1. Download and install the JSR 308 implementation; follow the instructions at http://groups.csail.mit.edu/pag/jsr308/current/README-jsr308.html#installing. This creates a langtools directory.
2. Download the Checker Framework distribution zipfile from http://groups.csail.mit.edu/pag/jsr308/current/jsr308-checkers.zip, and unzip it to create a checkers directory. We recommend that the checkers directory and the langtools directory be siblings. Example commands:

```
cd $JSR308
wget http://groups.csail.mit.edu/pag/jsr308/current/jsr308-checkers.zip
unzip jsr308-checkers.zip
```

   You will also need to adjust the path to javac in any Ant buildfiles, etc.
3. Optionally edit property compiler.lib in file checkers/build.properties. You don't have to do this if the checkers directory and the langtools directory are siblings.
4. Add to your classpath: $JSR308/jsr308-langtools/lib/tools.jar and $JSR308/checker-framework/checkers/checker (If you do not do this, you will have to supply the -cp option whenever you run javac and use a checker plugin.) Example commands:

```
export JSR308=${HOME}/jsr308
export CLASSPATH=${CLASSPATH}:$JSR308/jsr308-langtools/lib/tools.jar:$JSR308/checker-framework/checkers/checkers.jar
```

5. Test that everything works:
   - Run ant all-tests in the checkers directory:

```
cd checkers
ant all-tests
```

   - Run the Nullness checker examples (see Section 3.4).

JSR 308 extends the Java language to permit annotations to appear on types, as in List<@NonNull String> (see Section 2.1). This change will be part of the Java 7 language. We recommend that you write annotations in comments, as in List</*@NonNull*/ String> (see Section 2.9). The JSR 308 compiler still reads such annotations, but this syntax permits you to use a compiler other than the JSR 308 compiler. For example, you can compile your code with a Java 5 compiler, and you can use a checker as an external tool in an IDE such as Eclipse.

11.3.3 Building from source

Building (compiling) the checkers and framework from source creates the checkers.jar file. A pre-compiled checkers.jar is included in the distribution, so building it is optional. It is mostly useful for people who are developing compiler plug-ins (type-checkers). If you only want to *use* the compiler and existing plug-ins, it is sufficient to use the pre-compiled version.

1. Edit `checkers/build.properties` file so that the `compiler.lib` property specifies the location of the JSR 308 `javac.jar` library. (If you also installed the JSR 308 compiler from source, and you made the `checkers` and `langtools` directories siblings, then you don't need to edit `checkers/build.properties`.)
2. Run `ant` in the `checkers` directory:

```
cd checkers
ant
```

11.4 Learning more

The technical paper "Practical pluggable types for Java" [PAC+08] (`http://www.cs.washington.edu/homes/mernst/pubs/pluggable-checkers-issta2008.pdf`) gives more technical detail about many aspects of the Checker Framework and its implementation. The technical paper also describes a few features that are part of the distribution but are not yet documented in this manual. Finally, the technical paper describes case studies in which each of the checkers found previously-unknown errors in real software.

11.5 Comparison to other tools

A pluggable type-checker, such as those created by the Checker Framework, aims to help you prevent or detect all errors of a given variety. An alternate approach is to use a bug detector such as FindBugs, JLint, or PMD.

A pluggable type-checker differs from a bug detector in several ways:

- A type-checker aims to find *all* errors. Thus, it can verify the *absence* of errors: if the type checker says there are no null pointer errors in your code, then there are none. (This guarantee only holds for the code it checks, of course; see Section 2.8.)
  A bug detector aims to find *some* of the most obvious errors. Even if it reports no errors, then there may still be errors in your code.
  Both types of tools may issue false positive warnings; see Section 2.4.
- A type-checker requires you to annotate your code with type qualifiers, or to run an inference tool that does so for you. A bug detector may not require annotations. This means that it may be easier to get started running a bug detector.
- A type-checker may use more a more sophisticated and complete analysis. A bug detector typically does a more lightweight analysis, coupled with heuristics to suppress false positives.
  As one example, a type-checker can take advantage of annotations on generic type parameters, such as `List<@NonNull String>`, permitting it to be much more precise for code that uses generics.

A case study [PAC+08, §6] compared the Checker Framework's nullness checker with those of FindBugs, JLint, and PMD. The case study was on a well-tested program in daily use. The Checker Framework tool found 8 nullness errors. None of the other tools found any errors.

11.6 Credits and changelog

The Checker Framework distribution was developed in the MIT Program Analysis Group, with prime contributions from Mahmood Ali, Telmo Correa, Michael D. Ernst, and Matthew M. Papi. Many users have provided valuable feedback, for which we are grateful.

Differences from previous versions of the checkers and framework can be found in the `changelog-checkers.txt` file. This file is included in the checkers distribution and is also available on the web at `http://groups.csail.mit.edu/pag/jsr308/current/changelog-checkers.txt`.

References

[Cro06] Jose Cronembold. JSR 198: A standard extension API for Integrated Development Environments. `http://jcp.org/en/jsr/detail?id=198`, May 8, 2006.

[Dar06] Joe Darcy. JSR 269: Pluggable annotation processing API. http://jcp.org/en/jsr/detail?id=269, May 17, 2006. Public review version.

[Ern08] Michael D. Ernst. Type Annotations specification (JSR 308). http://pag.csail.mit.edu/jsr308/, September 12, 2008.

[Eva96] David Evans. Static detection of dynamic memory errors. In *PLDI 1996, Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 44–53, Philadelphia, PA, USA, May 21–24, 1996.

[FL03] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2003)*, pages 302–312, Anaheim, CA, USA, November 6–8, 2003.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI 2002, Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245, Berlin, Germany, June 17–19, 2002.

[Goe06] Brian Goetz. The pseudo-typedef antipattern: Extension is not type definition. http://www.ibm.com/developerworks/library/j-jtp02216.html, February 21, 2006.

[GPB+06] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3), March 2006.

[PAC+08] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 201–212, Seattle, WA, USA, July 22–24, 2008.

[QTE08] Jaime Quinonez, Matthew S. Tschantz, and Michael D. Ernst. Inference of reference immutability. In *ECOOP 2008 — Object-Oriented Programming, 22nd European Conference*, pages 616–641, Paphos, Cyprus, July 9–11, 2008.

[TE05] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, pages 211–230, San Diego, CA, USA, October 18–20, 2005.

[ZPA+07] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kieżun, and Michael D. Ernst. Object and reference immutability using Java generics. In *ESEC/FSE 2007: Proceedings of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 75–84, Dubrovnik, Croatia, September 5–7, 2007.