

The Checker Framework: Custom pluggable types for Java

<http://types.cs.washington.edu/checker-framework/>

Version 1.1.1 (19 Sep 2010)

For the impatient: Section 1.2 (page 9) describes how to **install and use** pluggable type-checkers.

Contents

1	Introduction	8
1.1	How it works: Pluggable types	8
1.2	Installation	9
1.3	Example use: detecting a null pointer bug	9
2	Using a checker	11
2.1	Writing annotations	11
2.1.1	Distributing your annotated project	12
2.2	Running a checker	12
2.2.1	Summary of command-line options	13
2.2.2	Checker auto-discovery	13
2.3	What the checker guarantees	13
2.4	Tips about writing annotations	14
2.4.1	How to get started annotating legacy code	14
2.4.2	Do not annotate local variables unless necessary	15
2.4.3	Annotations indicate normal behavior	15
2.4.4	Subclasses must respect superclass annotations	15
2.4.5	Annotations on constructor invocations	16
2.4.6	When to use (and not use) type qualifiers	17
3	Nullness checker	18
3.1	Nullness annotations	18
3.1.1	Nullness qualifiers	18
3.1.2	Nullness method annotations	19
3.1.3	Rawness qualifiers	19
3.1.4	Map key qualifiers	19
3.2	Writing nullness annotations	20
3.2.1	Implicit qualifiers	20
3.2.2	Default annotation	20
3.2.3	Conditional nullness	20
3.2.4	Inference of <code>@NonNull</code> and <code>@Nullable</code> annotations	21
3.3	What the Nullness checker checks	21
3.4	Suppressing nullness warnings	22
3.4.1	Suppressing warnings with assertions and method calls	22
3.4.2	Suppressing warnings on nullness-checking routines and defensive programming	23
3.5	<code>@Raw</code> annotation for partially-initialized objects	24
3.6	Map key annotations	25
3.7	Examples	26
3.7.1	Tiny examples	26
3.7.2	Annotated library	26

3.8	Other tools for nullness checking	26
3.8.1	Which tool is right for you?	27
3.8.2	Incompatibility note about FindBugs @Nullable	27
4	Interning checker	29
4.1	Interning annotations	30
4.2	Annotating your code with @Interned	30
4.2.1	Implicit qualifiers	30
4.3	What the Interning checker checks	30
4.4	Examples	31
5	IGJ immutability checker	32
5.1	IGJ and Mutability	32
5.2	IGJ Annotations	32
5.3	What the IGJ checker checks	33
5.4	Implicit and default qualifiers	33
5.5	Annotation IGJ Dialect	33
5.5.1	Semantic Changes	34
5.5.2	Syntax Changes	34
5.5.3	Templating Over Immutability: @I	34
5.6	Examples	35
6	Javari immutability checker	36
6.1	Javari annotations	36
6.2	Writing Javari annotations	37
6.2.1	Implicit qualifiers	37
6.2.2	Inference of Javari annotations	37
6.3	What the Javari checker checks	37
6.4	Examples	37
7	Lock checker	38
7.1	Lock annotations	38
7.1.1	Examples	38
7.1.2	Discussion of @Holding	39
7.1.3	Relationship to annotations in <i>Java Concurrency in Practice</i>	40
8	Fake Enum checker	41
8.1	Fake enum annotations	41
8.2	What the Fenum checker checks	42
8.3	Running the Fenum checker	42
8.4	Suppressing warnings	42
8.5	Example	42
8.6	References	43
9	Tainting checker	44
9.1	Tainting annotations	44
9.2	Tips on writing @Untainted annotations	44
9.3	@Tainted and @Untainted can be used for many purposes	45
10	Linear checker for preventing aliasing	46
10.1	Linear annotations	46
10.2	Limitations	47

11	Regex checker for regular expression syntax	48
11.1	Regex annotations	48
12	Property file checker	49
12.1	Generic property file checker	49
12.2	Internationalization checker	50
12.2.1	Internationalization annotations	50
12.2.2	Running the Internationalization Checker	50
12.3	Compiler Message Key checker	50
13	Basic checker	52
13.1	Using the Basic checker	52
13.2	Basic checker example	52
14	Typestate checker	55
14.1	Comparison to flow-sensitive type refinement	55
15	External checkers	56
15.1	Units and dimensions checker	56
16	Advanced type system features	57
16.1	Polymorphism and generics	57
16.1.1	Generics (parametric polymorphism or type polymorphism)	57
16.1.2	Qualifier polymorphism	59
16.2	Unused fields and dependent types	61
16.2.1	Unused fields	61
16.2.2	Dependent types	61
16.2.3	Example	61
16.3	The effective qualifier on a type (defaults and inference)	62
16.3.1	Default qualifier for unannotated types	62
16.3.2	Automatic type refinement (flow-sensitive type qualifier inference)	64
16.3.3	Fields and flow sensitivity analysis	65
16.4	Inexpressible types	65
17	Handling warnings and legacy code	66
17.1	Checking partially-annotated programs: handling unannotated code	66
17.2	Suppressing warnings	66
17.3	Writing annotations in comments for backward compatibility	68
17.3.1	Annotations in comments	68
17.3.2	Implicit import statements	68
17.3.3	Migrating away from annotations in comments	69
18	Annotating libraries	70
18.1	Using stub classes	70
18.1.1	Creating a stub file	70
18.1.2	Using a stub file	71
18.1.3	Stub file format	71
18.1.4	Distributing stub files	72
18.1.5	Known problems	72
18.1.6	Style tips for stub files	72
18.2	Using distributed annotated JDKs	72

19	How to create a new checker	73
19.1	Relationship of the Checker Framework to other tools	73
19.2	The parts of a checker	74
19.3	Annotations: Type qualifiers and hierarchy	74
19.3.1	Declaratively defining the qualifier and type hierarchy	74
19.3.2	Procedurally defining the qualifier and type hierarchy	75
19.3.3	Defining a default annotation	76
19.3.4	Completeness of the type hierarchy	76
19.4	Type factory: Implicit annotations	77
19.4.1	Declaratively specifying implicit annotations	77
19.4.2	Procedurally specifying implicit annotations	77
19.5	Visitor: Type rules	78
19.5.1	AST traversal	78
19.5.2	Avoid hardcoding	79
19.6	The checker class: Compiler interface	79
19.6.1	Bundling multiple checkers	79
19.7	Testing framework	80
19.8	Debugging options	80
19.9	javac implementation survival guide	80
19.9.1	Checker access to compiler information	81
19.9.2	How a checker fits in the compiler as an annotation processor	81
20	Integration with external tools	83
20.1	Javac Compiler	83
20.1.1	Unix/Linux/MacOS installation	83
20.1.2	Windows installation	84
20.2	Ant task	85
20.2.1	Explanation	85
20.3	Maven plugin	86
20.4	IntelliJ IDEA	87
20.5	Eclipse	87
20.6	tIDE	88
20.7	Type inference tools	88
20.7.1	Varieties of type inference	88
20.7.2	Type inference to annotate a program	89
21	Frequently Asked Questions (FAQs)	90
21.1	Are type annotations easy to read and write?	90
21.2	Will my code become cluttered with type annotations?	91
21.3	Can a pluggable type-checker give an absolute guarantee of correctness?	91
21.4	I don't make type errors, so would pluggable type checking help me?	91
21.5	Why shouldn't a qualifier apply to both types and declarations?	92
21.6	When should I use type qualifiers, and when should I use subclasses?	92
21.7	How do I get started annotating an existing program?	92
21.8	How do I run a checker on all my source files?	92
21.9	How do I shorten the command line when invoking a checker?	92
21.10	How do I create a new checker?	93
21.11	Why is there no declarative syntax for writing type rules?	93
21.12	Why not just use a bug detector (like FindBugs)?	93
21.13	How does pluggable type-checking compare with JML?	93
21.14	What is the meaning of array annotations such as @NonNull Object @Nullable []?	94
21.15	Why are the type parameters to List and Map annotated as @NonNull?	94

21.16	How can I do run-time monitoring of properties that were not statically checked?	95
22	Troubleshooting and getting help	96
22.1	Common problems and solutions	96
22.1.1	Known problems in the framework	97
22.1.2	Known problems in the Nullness checker	97
22.2	How to report problems	98
22.3	Building from source	98
22.3.1	Obtain the source	98
22.3.2	Build the Type Annotations compiler	98
22.3.3	Build the Checker Framework	99
22.3.4	Build the Checker Framework manual (this document)	99
22.4	Learning more	99
22.5	Comparison to other tools	99
22.6	Credits and changelog	100

Chapter 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 checkers for specific types of errors:

1. Nullness checker for null pointer errors (see Chapter 3, page 18)
2. Interning checker for errors in equality testing and interning (see Chapter 4, page 29)
3. IGJ checker for mutation errors (incorrect side effects), based on the IGJ type system (see Chapter 5, page 32)
4. Javari checker for mutation errors (incorrect side effects), based on the Javari type system (see Chapter 6, page 36)
5. Lock checker for concurrency and lock errors, inspired by the Java Concurrency in Practice (JCIP) annotations (see Chapter 7, page 38)
6. Fake enum checker to allow type-safe fake enum patterns (see Chapter 8, page 41)
7. Tainting checker for trust and security errors (see Chapter 9, page 44)
8. Linear checker to control aliasing and prevent re-use (see Chapter 10, page 46)
9. Regex checker to prevent use of syntactically invalid regular expressions (see Chapter 11, page 48)
10. Property file checker to ensure that valid keys are used for property files and resource bundles (see Chapter 12, page 49). Also includes a checker that code is properly internationalized and a checker for compiler message keys as used in the Checker Framework.
11. Basic checker for customized checking without writing any code (see Chapter 13, page 52)
12. Typestate checker to ensure operations are performed on objects that are in the right state, such as only opened files being read (see Chapter 14, page 55)
13. Units and dimensions checker to prevent mixing variables that measure different quantities (see Chapter 15.1, page 56)

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 Chapters 13 and 19.

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. 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, you can build the latest development version from source (Section 22.3, page 98).

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

The installation process is simple! (For a set of commands that you can copy and paste into your command shell, see Section 20.1.)

1. Download the Checker Framework distribution (<http://types.cs.washington.edu/checker-framework/current/checker-framework-1.7.0-javadoc.jar>).
2. Unzip it to create a `checker-framework` directory.
3. Optionally, update your execution path.

When doing pluggable type-checking, you need to use the “Type Annotations compiler”, which is an updated version of the OpenJDK `javac` compiler. If you add directory `.../checker-framework/checkers/binary` to your path, then whenever you run `javac`, you will use the updated compiler. Or, you can install the compiler (from <http://types.cs.washington.edu/jsr308/>) using its own installation instructions.

There are no negative consequences to using the Type Annotations compiler, but if you choose not to do so, you have alternatives. When this document tells you to run `javac`, you will instead need to run one of the following commands. The command should be all on one line, and followed by the `javac` arguments such as `-processor`.

```
# Unix
/path/to/checker-framework/checkers/binary/javac

# Unix
java -Xbootclasspath/p:$JSR308/checker-framework/checkers/binary/jsr308-all.jar
-jar $JSR308/checker-framework/checkers/binary/jsr308-all.jar -version

# Windows
java -Xbootclasspath/p:C:\Path\To\...\checker-framework\checkers\binary\jsr308-all.jar
-jar C:\Path\To\...\checker-framework\checkers\binary\jsr308-all.jar -version
```

To ensure that it was installed properly, run `javac -version` (using a variant of `javac` if you did not add it to your path).

The output should be:

```
javac 1.7.0-jsr308-1.1.1
```

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 Chapter 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. (You can also use an IDE or other build tool; see Chapter 20.)

For instance, if you usually run the compiler like this:

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

then you will instead use the command line:

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

but take note that the `javac` command must refer to the Type Annotations compiler (see Section 1.2).

If you usually do your coding within an IDE, you will need to configure the IDE. This manual contains instructions for Ant (Section 20.2), Maven (Section 20.3), IntelliJ IDEA (Section 20.4), Eclipse (Section 20.5), and tIDE (Section 20.6). Otherwise, 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.nullnessquals.*;

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

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

```
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
```

Chapter 2

Using a checker

A pluggable type-checker enables you to detect certain bugs in your code, or to prove that they are not present. The verification happens at compile time.

Finding bugs, or verifying their absence, with a checker plugin is a two-step process, whose steps are described in Sections 2.1 and 2.2.

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 3.2.4 and 6.2.2.) It is possible to annotate only part of your code: see Section 17.1.
2. The checker reports whether the program contains any erroneous code — that is, code that is inconsistent with the annotations.

This section is structured as follows:

- Section 2.1: How to write annotations
- Section 2.2: How to run a checker
- Section 2.3: What the checker guarantees
- Section 2.4: Tips about writing annotations

Additional topics that apply to all checkers are covered later in the manual:

- Chapter 16: Advanced type system features
- Chapter 17: Handling warnings and legacy code
- Chapter 18: Annotating libraries
- Chapter 19: How to create a new checker
- Chapter 20: Integration with external tools

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 Type Annotations or Java 7 compiler. For more details, see Section 17.3.

2.1.1 Distributing your annotated project

If your code contains annotations, then your code has a dependency on the annotation declarations. People who want to compile or run your code may need declarations of the annotations on their classpath.

- To perform pluggable type-checking, all of the Checker Framework (which also contains the annotation declarations) is needed.
- To compile the code:
 - If you wrote annotations in comments (see Section 17.3) and used implicit import statements (see Section 17.3.2), then the code can be compiled by any Java compiler, without needing declarations of the annotations.
 - Otherwise, compiling the code requires a declaration of the annotations. These appear in the full Checker Framework. Additionally, the Checker Framework distribution .zip file contains a small jar file, `checkers-quals.jar`, that only contains the definitions of the distributed qualifiers, without any support for type-checking.
- To run the code:
 - If you compiled the code without using the annotation declarations, then no annotation declarations are needed.
 - If you compiled the code using the annotation declarations, then users may need to have the annotation declarations on their classpath.

A simple rule of thumb is as follows. When distributing your source code, you may wish to include either the Checker Framework jar file or the `checkers-quals.jar` file. When distributing compiled binaries, you may wish to compile them without using the annotations, or include the contents of `checkers-quals.jar` in your distribution.

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 can run a checker from within your favorite IDE or build system. See Chapter 20 for details about Ant (Section 20.2), Maven (Section 20.3), IntelliJ IDEA (Section 20.4), Eclipse (Section 20.5), and tIDE (Section 20.6), and about customizing other IDEs and build tools.) 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 -Xbootclasspath/p:checkers/jdk/jdk.jar MyFile.java
```

For a discussion of the `-Xbootclasspath/p` argument, see Section 18.2.

The checker is run only on any Java file that `javac` compiles. This includes all Java files specified on the command line (or created by another annotation processor). It may also include other of your Java files (but not if a more recent `.class` file exists). Even when the checker does not analyze a class (say, the class was already compiled, or source code is not available), 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 may wish to also supply `-Xmaxwarns 10000`, because by default `javac` prints at most 100 warnings.

You can always compile the code without the `-processor` command-line option, but in that case no checking of the type annotations is performed. The annotations are still written to the resulting `.class` files, however.

2.2.1 Summary of command-line options

You can pass command-line arguments to a checker via javac's standard `-A` option ("`A`" stands for "annotation"). All of the distributed checkers support the following command-line options:

- `-Awarns` Treat checker errors as warnings; see Section 2.2
- `-AskipClasses` Suppress all errors and warnings at all uses of a given class; see Section 17.2
- `-Alint` Enable or disable optional checks; see Section 17.2
- `-Astubs` List of stub files or directories; see Section 18.1.2
- `-Afilenames`, `-Anomsgtext`, `-Ashowchecks` Aids for testing or debugging a checker; see Section 19.8

Some checkers support additional options, such as `-Aqual` for the Basic Checker to check; see Chapter 13.

Here are some standard javac command-line options that you may find useful. Many of them contain the word "processor", because in javac jargon, a checker is a type of "annotation processor".

- `-processor` Names the checker to be run; see Section 2.2
- `-processorpath` Indicates where to search for the checker; should also contain any qualifiers used by the Basic Checker; see Section 13.2
- `-proc:{none,only}` Controls whether checking happens; `-proc:none` means to skip checking; `-proc:only` means to do only checking, without any subsequent compilation; see Section 2.2.2
- `-Xbootclasspath/p:` Indicates where to find the annotated JDK classes; see Section 18.2
- `-implicit:class` Suppresses warnings about implicitly compiled files (not named on the command line); see Section 20.2
- `-XDTA:spacesincomments` parse annotation comments even when they contain spaces; applicable only to the Type Annotations compiler; see Section 17.3
- `-J` Supply an argument to the JVM that is running javac; example: `-J-Djsr308_imports=checkers.nullness.quals.*`; see Section 17.3.2

2.2.2 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/javac.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, which disables all annotation processing including all pluggable type-checking.

2.3 What the checker guarantees

A checker can guarantee that a particular property holds throughout the code. For example, the Nullness checker (Chapter 3) guarantees that every expression whose type is a `@NonNull` type never evaluates to null. The Interning checker (Chapter 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.

- 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 running the checker, then there is no guarantee that the entire program satisfies the property being checked. Some examples of un-checked code are:

- Code compiled without the `-processor` switch, including any external library supplied as a `.class` file.
- Code compiled with the `-AskipClasses` property.
- Suppression of warnings, such as via the `@SuppressWarnings` annotation.
- Native methods (because the implementation is not Java code, it cannot be checked).

In each of these cases, any *use* of the code is checked — for example, a call to a native method must be compatible with any annotations on the native method’s signature. However, the annotations on the un-checked code are trusted; there is no verification that the implementation of the native method satisfies the annotations.

- Reflection can violate the Java type system, and the checkers are not sophisticated enough to reason about the possible effects of reflection. Similarly, deserialization and cloning can create objects that could not result from normal constructor calls, and that therefore may violate the property being checked.
- 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.

In order to avoid a flood of unhelpful warnings, many of the checkers avoid issuing the same warning multiple times. For example, in this code:

```
@Nullable Object x = ...;
x.toString();           // warning
x.toString();           // no warning
```

In this case, the second call to `toString` cannot possibly throw a null pointer warning — `x` is non-null if control flows to the second statement. In other cases, a checker avoids issuing later warnings with the same cause even when later code in a method might also fail. This does not affect the soundness guarantee, but a user may need to examine more warnings after fixing the first ones identified. (More often, at least in our experience to date, a single fix corrects all the warnings.)

If you find that a checker fails to issue a warning that it should, then please report a bug (see Section 22.2).

2.4 Tips about writing annotations

2.4.1 How to get started annotating legacy code

Annotating an entire existing program may seem like a daunting task. But, if you approach it systematically and do a little bit at a time, you will find that it is manageable.

You should start with a property that matters to you, to achieve the best benefits. It is easiest to add annotations if you know the code or the code contains documentation; you will find that you spend most of your time understanding the code, and very little time actually writing annotations or running the checker.

It is best to annotate one package at a time, and to annotate the entire package so that you don’t forget any classes, which can lead to unexpected results. Start as close to the leaves of the call tree as possible, because it is easiest to annotate a class if the code it calls has already been annotated.

For each class, read its Javadoc. For instance, if you are adding annotations for the Nullness Checker (Section 3), then you can search the documentation for “null” and then add `@Nullable` anywhere appropriate. Do not annotate the method bodies yet — first, get the signatures and fields annotated. The only reason to even *read* the method bodies yet is to determine signature annotations for undocumented methods — for example, if the method returns null, you know its return type should be annotated `@Nullable`, and a parameter that is compared against null may need to be annotated `@Nullable`. If you are only annotating signatures (say, for a library you do not maintain and do not wish to check), you are now done.

If you wish to check the implementation, then after the signatures are annotated, run the checker. Then, add method body annotations (usually, few are necessary), fix bugs in code, and add annotations to signatures where necessary. If signature annotations are necessary, then you may want to fix the documentation that did not indicate the property; but this isn't strictly necessary, since the annotations that you wrote provide that documentation.

You may wonder about the effect of adding a given annotation — how many other annotations it will require, or whether it conflicts with other code. Suppose you have added an annotation to a method parameter. You could manually examine all callees. A better way can be to save the checker output before adding the annotation, and to compare it to the checker output after adding the annotation. This helps you to focus on the specific consequences of your change.

Also see Chapter 17, which tells you what to do when you are unable to eliminate checker warnings.

2.4.2 Do not annotate local variables unless necessary

The checker infers annotations for local variables (see Section 16.3.2). Usually, you only need to annotate fields and method signatures. After doing those, you can add annotations inside method bodies if the checker is unable to infer the correct annotation, if you need to suppress a warning (see Section 17.2), etc.

2.4.3 Annotations indicate normal behavior

You should use annotations to indicate *normal* behavior. The annotations 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)
```

`@Nullable` (see Section 3.1) might seem like a reasonable annotation for the parameter, for two reasons. First, `null` is a legal argument with a well-defined semantics: throw an exception. Second, `@Nullable` 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. When you use the `@NonNull` annotation, the checker is able to issue compile-time warnings about possible run-time exceptions, which is its purpose. Marking the parameter as `@Nullable` would suppress such warnings, which is undesirable.

2.4.4 Subclasses must respect superclass annotations

An annotation indicates a guarantee that a client can depend upon. A subclass is 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 2.4.3 that annotations should indicate normal behavior.)

If parameter `element` on line 141 were to be annotated, the correct annotation would be `@NonNull`. Suppose a client has a reference to same `Multiset ms`. The only way the client can be sure not to throw an exception is to pass only non-null elements to `ms.add()`. 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 pass any non-null value to `add()`, and may also pass `null`.

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.

It is a pleasant feature of the Checker Framework that in many cases, no annotations at all are needed on type parameters such as `E` in `Multiset`.

2.4.5 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. The “return value” annotation appears before the constructor name, for example:

```

class MyClass {
    @ReadOnly MyClass() { ... }
}

```

In general, you should only use an annotation on a constructor invocation when you know that the cast is guaranteed to succeed. An example from the IGJ checker (Chapter 5) is `new @Immutable MyClass()` or `new @Mutable MyClass()`, where you know that every other reference to the class is annotated `@ReadOnly`.

2.4.6 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 make your code incompatible with existing libraries or clients. Brian Goetz expands on this issue 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 Chapters 5 and 6). More generally, type qualifiers permit creating a new supertype, not just a subtype, of an existing Java type.

A final reason is efficiency. Type qualifiers can be more efficient, since there is 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.

Chapter 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 a program is run. See Section 3.3 for more details about the guarantee and what is checked.

To run the Nullness Checker, supply the `-processor checkers.nullness.NullnessChecker` command-line option to `javac`. For examples, see Section 3.7.

3.1 Nullness annotations

The Nullness checker uses three separate type hierarchies: one for nullness, one for rawness (Section 3.5), and one for map keys (Section 3.6). The Nullness checker has three varieties of annotations: nullness qualifiers, nullness method annotations, rawness qualifiers, and map key qualifiers.

3.1.1 Nullness qualifiers

The nullness hierarchy contains these qualifiers:

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

@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.

The `@NonNull` annotation is rarely written in a program, because it is the default (see Section 3.2.2).

@PolyNull indicates qualifier polymorphism. For a description of `@PolyNull`, see Section 16.1.2.

@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 (Section 16.3.2). 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 3.1 shows part of the type hierarchy for the Nullness type system. (The annotations exist only at compile time; at run time, Java has no multiple inheritance.)

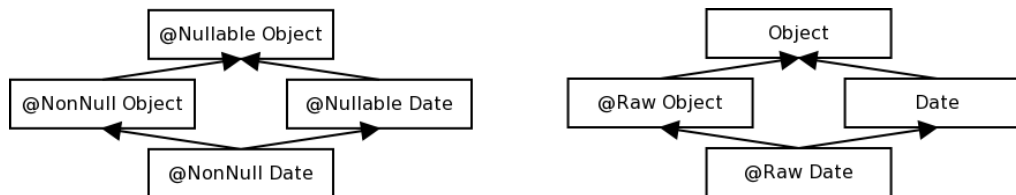


Figure 3.1: Partial type hierarchy for the Nullness type system. Java’s `Object` is expressed as `@Nullable Object`. Programmers can omit most type qualifiers, because the default annotation (Section 3.2.2) is usually correct. Also shown is the type hierarchy for rawness (Section 3.5), which indicates whether initialization has completed. The two type hierarchies are independent but inter-related, and the Nullness Checker verifies them both.

3.1.2 Nullness method annotations

The Nullness checker supports several annotations that specify method behavior.

@NonNullOnEntry indicates a method precondition: The annotated method expects the specified variables (typically field references) to be non-null when the method is invoked.

@AssertNonNullAfter

@AssertNonNullIfTrue

@AssertNonNullIfFalse indicates a method postcondition. With `@AssertNonNullAfter`, the given expressions are non-null after the method returns; this is useful for a method that initializes a field, for example. With `@AssertNonNullIfTrue` and `@AssertNonNullIfFalse`, if the annotated method returns the given boolean value (true or false), then the given expressions are non-null. See Section 3.2.3 and the Javadoc for examples of their use.

@Pure indicates that the method has no (visible) side effects. Furthermore, if the method is called multiple times with the same arguments, then it returns the same result. This property cannot be assumed in general. For example, suppose that the return value of method `m` is nullable. Then this code will pass the type-checker:

```

if (m(arg) != null) {
    m(arg).toString();
}

```

only if method `m` is annotated as `@Pure`.

@AssertParametersNonNull is used for suppressing warnings, in very rare cases. See the Javadoc for details.

3.1.3 Rawness qualifiers

The Nullness Checker supports rawness annotations that indicate whether an object is fully initialized — that is, whether its fields have all been assigned.

@Raw

@NonRaw

@PolyRaw

Use of these annotations can help you to type-check more code. Figure 3.1 shows its type hierarchy. For details, see Section 3.5.

3.1.4 Map key qualifiers

The Nullness Checker supports a map key annotation, `@KeyFor` that indicates whether a value is a key for a given map — that is, whether `map.containsKey(value)` would evaluate to `true`.

@KeyFor

Use of this annotation can help you to type-check more code. For details, see Section 3.6.

3.2 Writing nullness annotations

3.2.1 Implicit qualifiers

As described in Section 16.3, 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 16.3.1.

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("Nullable")` annotation on a class or method declaration.
- `@NonNull`: Unannotated types are treated as non-null. You can activate this default via the `@DefaultQualifier("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`.) This is the standard behavior. You can explicitly activate this default via the `@DefaultQualifier(value="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 Conditional nullness

The Nullness Checker supports a form of conditional nullness types, via the `@AssertNonNullIfTrue` and `@AssertNonNullIfFalse` method annotations. The annotation on a method declares that some expressions are non-null, if the method returns true (false, respectively).

Consider `java.io.File`. Method `File.listFiles()` may return null, but is specified to return a non-null value if `File.isDirectory()` is true. The same holds for method `File.list()`. You could declare this relationship in the following way (this particular example is already done for you in the annotated JDK that comes with the Checker Framework):

```
class File {  
  
    @AssertNonNullIfTrue({"list()", "listFiles()"})  
    public boolean isDirectory() { ... }  
  
    public File @Nullable [] listFiles();  
}
```

A client that checks that a `File` reference is indeed that of a directory, can then de-reference `File.isDirectory` safely without any nullness check.

```
static void analyze(File file) {  
    if (file.isDirectory()) {  
        for (File child : file.listFiles()) { // no possible null dereference
```

```

        analyze(child);
    }
} else {
    ... analyze file ...
}
}

```

3.2.4 Inference of @NonNull and @Nullable annotations

It can be tedious to write annotations in your code. 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 16.3.2), 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 one of these tools:
 - Julia analyzer,
 - Nit: Nullability Inference Tool,
 - Non-null checker and inferencer of the JastAdd Extensible Compiler.

3.3 What the Nullness checker checks

The checker issues a warning in three 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 thrown, 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.
3. When a null check is performed against a value that is guaranteed to be non-null, as in `("m" == null)`, because this might indicate a programmer error or misunderstanding, and is unnecessary. This check is performed only if the `nulltest` lint option is enabled via the `-Alint=nulltest` command-line option. The lint option is disabled by default because sometimes such checks are part of ordinary defensive programming. See Section 17.2 for more details about the `-Alint` command-line option.

This example illustrates the programming errors that the checker detects:

```

        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
if (nnobj == null)   // checker warning:  redundant test

```

Parameter passing and return values are checked analogously to assignments.

The Nullness Checker also checks the correctness, and correct use, of rawness annotations for checking initialization. See Section 3.5.

3.4 Suppressing nullness warnings

The Checker Framework supplies several ways to suppress warnings, most notably the `@SuppressWarnings("nullness")` annotation (see Section 17.2). An example use is

```
// might return null
@Nullable Object getObject() { ... }

void myMethod() {
    // The programmer knows that this particular call never returns null.
    @SuppressWarnings("nullness")
    @NonNull Object o2 = getObject();
}
```

The Nullness Checker supports an additional warning suppression key, `nullness:collection-typeargs`. Use of `@SuppressWarnings("nullness:generic.argument")` causes the Nullness Checker to suppress warnings related to misuse of generic type arguments. One use for this key is when a class is declared to take only `@NonNull` type arguments, but you want to instantiate the class with a `@Nullable` type argument, as in `List<@Nullable Object>`. For a more complete explanation of this example, see Section 21.15, page 94.

The Nullness Checker also permits you to use assertions or method calls to suppress warnings; see below.

3.4.1 Suppressing warnings with assertions and method calls

Occasionally, it is inconvenient or verbose to use the `@SuppressWarnings` annotation. For example, Java does not permit annotations such as `@SuppressWarnings` to appear on statements.

For situations when the `@SuppressWarnings` annotation is inconvenient, the Nullness Checker provides three additional ways to suppress warnings: via an `assert` statement, the `castNonNull` method, and the `@AssertParametersNonNull` annotation. These are appropriate when the Nullness Checker issues a warning, but the programmer knows for sure that the warning is a false positive, because the value cannot ever be null at run time.

1. Use an assertion. If the string “nullness” appears in the message body, then the Nullness Checker treats the assertion as suppressing a warning and assumes that the assertion always succeeds. For example, the checker assumes that no null pointer exception can occur in code such as

```
assert x != null : "@SuppressWarnings(nullness)";
... x.f ...
```

If the string “nullness” does not appear in the assertion message, then the Nullness Checker treats the assertion as being used for defensive programming, and it warns if the method might throw a nullness-related exception. A downside of putting the string in the assertion message is that if the assertion ever fails, then a user might see the string and be confused. But the string should only be used if the programmer has reasoned that the assertion can never fail.

2. Use the `NullnessUtils.castNonNull` method.

The Nullness Checker considers both the return value, and also the argument, to be non-null after the method call. Therefore, the `castNonNull` method can be used either as a cast expression or as a statement. The Nullness Checker issues no warnings in any of the following code:

```
// one way to use as a cast:
@NonNull String s = castNonNull(possiblyNull1);

// another way to use as a cast:
castNonNull(possiblyNull2).toString();

// one way to use as a statement:
castNonNull(possiblyNull3);
possiblyNull3.toString();`
```

The method also throws `AssertionError` if Java assertions are enabled and the argument is `null`. However, it is not intended for general defensive programming; see Section 3.4.2.

A potential disadvantage of using the `castNonNull` method is that your code becomes dependent on the Checker Framework at run time as well as at compile time. You can avoid this by copying the implementation of `castNonNull` into your own code, and possibly renaming it if you do not like the name. Be sure to retain the documentation that indicates that your copy is intended for use only to suppress warnings and not for defensive programming. See Section 3.4.2 for an explanation of the distinction.

3. Use the `@AssertParametersNonNull` annotation. It is used on `castNonNull`, and may be used on other methods with the same semantics; it should probably never be used in any other situation.

3.4.2 Suppressing warnings on nullness-checking routines and defensive programming

One way to suppress warnings in the Nullness Checker is to use method `castNonNull`. (Section 3.4.1 gives other techniques.)

This section explains why the Nullness Checker introduces a new method rather than re-using the `assert` statement (as in `assert x != null`) or an existing method such as:

```
org.junit.Assert.assertNotNull(Object)
com.google.common.base.Preconditions.checkNotNull(Object)
```

In each case, the assertion or method indicates an application invariant — a fact that should always be true. There are two distinct reasons a programmer may have written the invariant, depending on whether the programmer is 100% sure that the application invariant holds.

1. A programmer might write it as **defensive programming**. This causes the program to throw an exception, which is useful for debugging because it gives an earlier run-time indication of the error. A programmer would use an assertion in this way if the programmer is not 100% sure that the application invariant holds.
2. A programmer might write it to **suppress** false positive **warning messages** from a checker. A programmer would use an assertion this way if the programmer is 100% sure that the application invariant holds, and the reference can never be null at run time.

With assertions and existing methods like JUnit's `assertNotNull`, there is no way of knowing the programmer's intent in using the method. Different programmers or codebases may use them in different ways. Guessing wrong would make the Nullness Checker less useful, because it would either miss real errors or issue warnings where there is no real error. Also, different checking tools issue different false warnings that need to be suppressed, so warning suppression needs to be customized for each tool rather than inferred from general-purpose code.

As an example of using assertions for defensive programming, some style guides suggest using assertions or method calls to indicate nullness. A programmer might write

```
String s = ...
assert s != null;    // or: assertNotNull(s);    or: checkNotNull(s);
... Double.valueOf(s) ...
```

A programming error might cause `s` to be null, in which case the code would throw an exception at run time. If the assertion caused the Nullness Checker to assume that `s` is not null, then the Nullness Checker would issue no warning for this code. That would be undesirable, because the whole purpose of the Nullness Checker is to give a compile-time warning about possible run-time exceptions. Furthermore, if the programmer uses assertions for defensive programming systematically throughout the codebase, then many useful Nullness Checker warnings would be suppressed.

Because it is important to distinguish between the two uses of assertions (defensive programming vs. suppressing warnings), the Checker Framework introduces the `NullnessUtils.castNonNull` method. Unlike existing assertions and methods, `castNonNull` is intended only to suppress false warnings that are issued by the Nullness Checker, not for defensive programming.

If you know that a particular codebase uses a nullness-checking method not for defensive programming but to indicate facts that are guaranteed to be true (that is, these assertions will never fail at run time), then you can cause the Nullness Checker to suppress warnings related to them, just as it does for `castNonNull`. Annotate its definition just as `NullnessUtils.castNonNull` is annotated (see the source code for the Checker Framework). Also, be sure to document the intention in the method's Javadoc, so that programmers do not accidentally misuse it for defensive programming.

If you are annotating a codebase that already contains precondition checks, such as:

```
public String get(String key, String def) {
    checkNotNull(key, "key"); //NOI18N
    ...
}
```

then you should mark the appropriate parameter as `@NonNull` (which is the default). This will prevent the checker from issuing a warning about the `checkNotNull` call.

3.5 `@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. Most readers can skip this section on first reading; you can return to it once you have mastered the rest of the nullness checker.

The rawness hierarchy is independent of the nullness hierarchy, and is shown in Figure 3.1. 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 16.1.2).

Suppose a class contains a field “`@NonNull Date d;`”. Java executes the class's constructor by first setting `d` to null. The constructor sets field `d` to its final value, either directly or by calling helper methods. Before the constructor sets field `d`, its initial value null violates its type `@NonNull Date`. In general, code can depend on field `d` not being null, but not in a partially-initialized object. A partially-initialized object (`this` in a constructor) may be passed to a helper method or stored in a variable; if so, the method receiver, or the field, would have to be annotated as `@Raw`.

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 `@NonNull` fields in the referred-to object: might those fields be temporarily null in contravention of their declaration?

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 Within the constructor, `this` starts out with `@Raw` type. As soon as all of the `@NonNull` fields have been initialized, then `this` is treated as 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 (except blank final ones, defined in JLS §4.12.4) because fields have a default value of null.

Currently, the type-checker requires that all fields either have a default in the field declaration, or be initialized in the constructor. If your code initializes (some) fields in a helper method, you will need to suppress some warnings.

Invoking the superclass constructor; rawness of the superclass reference Suppose that class B extends class A. Within the B constructor, until the A superclass constructor is called, `this` has type `@Raw B` and also `@Raw A`. After the superclass constructor has been exited, then `this` has type `@Raw B` and also `@NonRaw A`. By the time that the constructor exits, `this` has type `@NonRaw B` and also `@NonRaw A`.

When you write `@Raw`, the annotation applies only to the given class, not to any superclass. For instance, the checker interprets `@Raw B` as “`@Raw B` and also **`@NonRaw A`**”, rather than “`@Raw B` and also `@Raw A`”, which would be less useful. The only exception is when a method overriding relationship forces the superclass to also be raw. For example:

```
class A extends Object {
    // receiver is "@NonRaw A"
    void nonRawAReceiver() { }
    // annotation forces receiver to be "@Raw A"; also is "@NonRaw Object"
    void rawAReceiver() @Raw { }
}

class B extends A {
    // annotation forces receiver to be "@Raw B", method overriding forces "@Raw A"
    void rawAReceiver() @Raw {
        super.nonRawAReceiver(); // illegal! rawness of A does not match
    }
    // annotation forces receiver to be "@Raw B"; also is "@NonRaw A"
    void rawBReceiver() @Raw {
        super.nonRawAReceiver(); // OK
    }
}
```

A note about the terminology “raw” The name “raw” comes from a research paper that proposed this approach [FL03]. A better name might have been “not yet initialized” or “partially initialized”, but the term “raw” is now well-known. The `@Raw` annotation has nothing to do with the raw types of Java Generics.

3.6 Map key annotations

Java’s `Map.get` method always has the possibility to return null, if the key is not in the map. Thus, to guarantee that the value returned from `Map.get` is non-null, it is necessary that the map contains only non-null values, *and* the key is in the map. The `@KeyFor` annotation states the latter property.

If a type is annotated as `@KeyFor("m")`, then any value `v` with that type is a key in `Map m`. Another way of saying this is that the expression `m.containsKey(v)` evaluates to true.

You usually do not have to write `@KeyFor` explicitly, because the checker infers it based on usage patterns, such as calls to `containsKey` or iteration over a map’s key set.

One usage pattern where you *do* have to write `@KeyFor` is for a user-managed collection that is a subset of the key set:

```
Map<String, Object> m;
Set<@KeyFor("m") String> matchingKeys; // keys that match some criterion
for (@KeyFor("m") String k : matchingKeys) {
```

```
... m.get(k) ... // known to be non-null
}
```

As with any annotation, use of the `@KeyFor` annotation may force you to slightly refactor your code. For example, this would be illegal:

```
Map<K,V> m;
Collection<@KeyFor("m") K> coll;
coll.add(x);
...           // at this point, the @KeyFor annotation is violated
m.put(x, ...);
```

but this would be OK:

```
Collection<@KeyFor("m") K> coll;
m.put(x, ...);
coll.add(x);
```

3.7 Examples

3.7.1 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`.

3.7.2 Annotated library

Some libraries that are annotated with nullness qualifiers are:

- The Nullness checker itself.
- The Plume-lib library. Run the command `make check-nullness`.
- The Daikon invariant detector. Run the command `make check-nullness`.

3.8 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 22.5 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 3.2.

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

Figure 3.2: Refactoring for converting nullness annotations from FindBugs, the JSR 305 proposal, and IntelliJ to the Checker Framework.

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 3.2, so that they can be used as type qualifiers. The Checker Framework interprets them according to the right side of Figure 3.2.

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: it aims to never miss 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 3.8.2).

3.8.1 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.

Checker Framework has a more powerful nullness analysis; FindBugs misses some real 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 22.2) so that we can enhance the Checker Framework.

3.8.2 Incompatibility note about FindBugs `@Nullable`

FindBugs has a non-standard definition of `@Nullable`. FindBugs’s treatment is not documented in its own Javadoc; it is different from the definition of `@Nullable` in every other tool for nullness analysis; it means the same thing as `@NonNull` when applied to a formal parameter; and it invariably surprises programmers. Thus, FindBugs’s `@Nullable` is detrimental rather than useful as documentation. In practice, your best bet is to not rely on FindBugs for nullness analysis, even if you find FindBugs useful for other purposes.

You can skip the rest of this section unless you wish to learn more details.

FindBugs suppresses all warnings at uses of a `Nullable` variable. (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() {
    @Nullable Object o = getObject();
    // FindBugs issues no warning about calling toString on a possibly-null reference!
    o.toString();
}
```

The Checker Framework does not emulate this non-standard behavior of FindBugs, even if the code uses FindBugs annotations.

FindBugs takes the approach of annotating a declaration, and thus suppressing checking at *all* client uses, even the places that you want to check. It is better to suppress warnings at only the specific client uses where the value is known to be non-null; the Checker Framework supports this, if you write `SuppressWarnings` at the client uses. The Checker Framework also supports suppressing checking at all client uses, by writing a `SuppressWarnings` annotation at the declaration site.

In general, the Checker Framework will issue more warnings than FindBugs, and some of them may be about real bugs in your program. See Section 3.4 for information about suppressing nullness warnings.

(FindBugs made a poor choice of names. The choice of names should make a clear distinction between annotations that specify whether a reference is null, and annotations that suppress false warnings. The choice of names should also have been consistent for other tools, and intuitively clear to programmers. The FindBugs choices make the FindBugs annotations less helpful to people, and much less useful for other tools. The FindBugs analysis is also very imprecise. For type-related analyses, it is best to stay away from the FindBugs nullness annotations, and use a more capable tool like the Checker Framework.)

Chapter 4

Interning checker

If the Interning checker issues no warnings for a given program, then all reference equality tests (i.e., all uses of “==”) are proper; == is not misused where equals() should have been used instead.

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 related to the flyweight design pattern. Interning can save memory and can speed up testing for equality by permitting use of ==. However, == should be used only on interned values; using == 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 2.3 for caveats to the checker’s guarantees.)

Interning is such an important design pattern that Java builds it in for strings. Every string literal in the program is guaranteed to be interned (JLS §3.10.5), and the String.intern() method performs interning for strings that are computed at run time. Users can also write their own interning methods for other types.

It is a proper optimization to use ==, rather than equals(), whenever the comparison is guaranteed to produce the same result — that is, whenever the comparison is never provided with two different objects for which equals() would return true. Here are three reasons that this property could hold:

1. Interning. A factory method ensures that, globally, no two different interned objects are equals() to one another. (In some cases other, non-interned objects of the class might be equals() to one another; in other cases, every object of the class is interned.) Interned objects should always be immutable.
2. Global control flow. The program’s control flow is such that the constructor for class *C* is called a limited number of times, and with specific values that ensure the results are not equals() to one another. Objects of class *C* can always be compared with ==. Such objects may be mutable or immutable.
3. Local control flow. Even though not all objects of the given type may be compared with ==, the specific objects that can reach a given comparison may be. For example, suppose that an array contains no duplicates. Then testing to find the index of a given element that is known to be in the array can use ==.

To eliminate Interning Checker warnings, you will need to annotate your code regarding all legal uses of ==. Thus, the Interning Checker should perhaps have been called the Reference Equality Checker. In the future, the checker will include annotations that target the non-interning cases above, but for now you need to use @Interned and/or @SuppressWarnings.

To run the Interning Checker, supply the -processor checkers.interning.InterningChecker command-line option to javac. For examples, see Section 4.4.

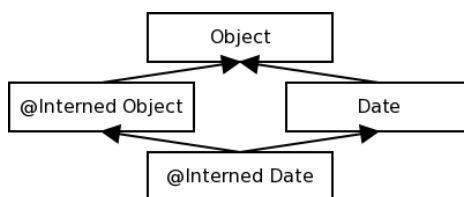


Figure 4.1: Type hierarchy for the Interning type system.

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 16.1.2.

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 16.3, 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`.

You can also restrict which types the checker should examine and type-check, using the `-Acheckclass` option. So if you want to find all the interning errors related to uses of `String`, you can pass `-Acheckclass=java.lang.String`.

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`.

Chapter 5

IGJ immutability 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 2.3 for caveats to the guarantee.)

To run the IGJ Checker, supply the `-processor checkers.igj.IGJChecker` command-line option to `javac`. For examples, see Section 5.6.

5.1 IGJ and Mutability

IGJ [ZPA⁺07] 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 more 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 a slightly different dialect of IGJ than that described in the ESEC/FSE paper.

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.

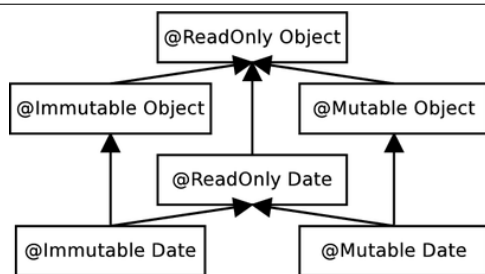


Figure 5.1: Type hierarchy for three of IGJ’s type qualifiers.

@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.

@Assignable The annotated field may be re-assigned regardless of the immutability of the enclosing class or object instance.

@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 reference is assigned to a mutable variable. The checker also emits a warning when casts increase the mutability access of a reference.

5.4 Implicit and default qualifiers

As described in Section 16.3, the IGJ checker adds implicit qualifiers, reducing the number of annotations that must appear in your code.

For a complete description of all implicit IGJ qualifiers, see the Javadoc for `IGJAnnotatedTypeFactory`.

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 16.3.2).
- **@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>
```

is valid in IGJ, but in Annotation IGJ, only

```
@Mutable Vector<Integer> <: @ReadOnly Vector<Integer>
```

holds and the other two subtype relations do not hold

```
@ReadOnly Vector<Integer> </: @ReadOnly Vector<Number>
                        </: @ReadOnly Vector<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 declaration annotated with `@I` can then be used 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> coll,
                                              @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.

Chapter 6

Javari immutability 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 2.3 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 Javariifier tool infers Javari types for an existing program; see Section 6.2.2.

Also consider the IGJ checker (Chapter 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 Javariifier.

To run the Javari Checker, supply the `-processor checkers.javari.JavariChecker` command-line option to `javac`. For examples, see Section 6.4.

6.1 Javari annotations

The following six annotations make up the Javari type system.

@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 16.1.2 and the `@PolyRead` Javadoc for more details.

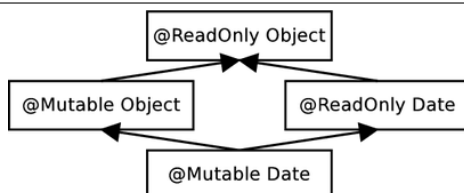


Figure 6.1: Type hierarchy for Javari’s ReadOnly type qualifier.

@ThisMutable means that the mutability of the field is the same as that of the reference that contains it. **@ThisMutable** is the default on fields, and does not make sense to write elsewhere. Therefore, **@ThisMutable** should never appear in a program.

6.2 Writing Javari annotations

6.2.1 Implicit qualifiers

As described in Section 16.3, the Javari checker adds implicit qualifiers, reducing the number of annotations that must appear in your code.

For a complete description of all implicit Javari 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 `.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.

Chapter 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 release 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 obeys its locking discipline is an easy and effective way to eliminate a common and important class of errors.

To run the Lock Checker, supply the `-processor checkers.lock.LockChecker` command-line option to `javac`.

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 `GuardedBy` Javadoc for an explanation of the argument. The lock acquisition and the value access may be arbitrarily far in the future; or, if the value is never accessed, the lock never need be held.

@Holding is a method annotation (not a qualifier). It indicates that when the method is called, the given lock must be held by the caller. In other words, the given lock is already held at the time the method is called.

7.1.1 Examples

Most often, field values are annotated with `@GuardedBy`, but other uses are possible.

A return value may be annotated with `@GuardedBy`:

```
@GuardedBy("MyClass.myLock") Object myMethod() { ... }

// reassignments without holding the lock are OK.
@GuardedBy("MyClass.myLock") Object x = myMethod();
@GuardedBy("MyClass.myLock") Object y = x;
Object z = x; // ILLEGAL (assuming no lock inference),
              // because z can be freely accessed.
x.toString() // ILLEGAL because the lock is not held
synchronized(MyClass.myLock) {
    y.toString(); // OK: the lock is held
}
```

A parameter may be annotated with `@GuardedBy`:

```

void helper1(@GuardedBy("MyClass.myLock") Object a) {
    a.toString(); // ILLEGAL: the lock is not held
    synchronized(MyClass.myLock) {
        a.toString(); // OK: the lock is held
    }
}
@Holding("MyClass.myLock")
void helper2(@GuardedBy("MyClass.myLock") Object b) {
    b.toString(); // OK: the lock is held
}
void helper3(Object c) {
    c.toString(); // OK: no lock constraints
}
void helper4(@GuardedBy("MyClass.myLock") Object d) {
    d.toString(); // ILLEGAL: the lock is not held
}
void myMethod2(@GuardedBy("MyClass.myLock") Object e) {
    helper1(e); // OK to pass to another routine without holding the lock
    e.toString(); // ILLEGAL: the lock is not held
    synchronized (MyClass.myLock) {
        helper2(e);
        helper3(e);
        helper4(e); // OK, but helper4's body still does not type-check
    }
}

```

7.1.2 Discussion of @Holding

A programmer might choose to use the `@Holding` method annotation in two different ways: to specify a higher-level protocol, or to summarize intended usage. Both of these approaches are useful, and the Lock checker supports both.

Higher-level synchronization protocol `@Holding` can specify a higher-level synchronization protocol that is not expressible as locks over Java objects. By requiring locks to be held, you can create higher-level protocol primitives without giving up the benefits of the annotations and checking of them.

Method summary that simplifies reasoning `@Holding` can be a method summary that simplifies reasoning. In this case, the `@Holding` doesn't necessarily introduce a new correctness constraint; the program might be correct even if the lock were acquired later in the body of the method or in a method it calls, so long as the lock is acquired before accessing the data it protects.

Rather, here `@Holding` expresses a fact about execution: when execution reaches this point, the following locks are already held. This fact enables people and tools to reason intra- rather than inter-procedurally.

In Java, it is always legal to re-acquire a lock that is already held, and the re-acquisition always works. Thus, whenever you write

```

@Holding("myLock")
void myMethod() {
    ...
}

```

it would be equivalent, from the point of view of which locks are held during the body, to write

```

void myMethod() {
    synchronized (myLock) {    // no-op:  re-acquire a lock that is already held
        ...
    }
}

```

The advantages of the `@Holding` annotation include:

- The annotation documents the fact that the lock is intended to already be held.
- The Lock Checker enforces that the lock is held when the method is called, rather than masking a programmer error by silently re-acquiring the lock.
- The `synchronized` statement can deadlock if, due to a programmer error, the lock is not already held. The Lock Checker prevents this type of error.
- The annotation has no run-time overhead. Even if the lock re-acquisition succeeds, it still consumes time.

7.1.3 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 that 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. This informal intuition is inappropriate for a tool that requires precise semantics.

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) meanings of the two annotations.

Chapter 8

Fake Enum checker

Java's `enum` keyword lets you define an enumeration type: a finite set of distinct values that are related to one another but are disjoint from all other types, including other enumerations. Before enums were added to Java, there were two ways to encode an enumeration, both of which are error-prone:

the fake enum pattern a set of `int` or `String` constants (as often found in older C code).

the typesafe enum pattern a class with private constructor.

Sometimes you need to use the fake enum pattern, rather than a real enum or the typesafe enum pattern. One reason is backward-compatibility. A public API that predates Java's `enum` keyword may use `int` constants; it cannot be changed, because doing so would break existing clients. For example, Java's JDK still uses `int` constants in the AWT and Swing frameworks. Another reason is performance, especially in environments with limited resources. For example, the Android mobile phone platform recommends use of fake enums when only an integer value is needed, in order to reduce code size and run time.

In cases when code has to use the fake enum pattern, the fake enum (Fenum) checker gives the same safety guarantees as a true enumeration type. The developer can introduce new types that are distinct from all values of the base type and from all other fake enums. Fenums can be introduced for primitive types as well as for reference types.

8.1 Fake enum annotations

The checker supports two ways to introduce a new fake enum (fenum):

1. Introduce your own specialized fenum annotation with code like this in file *MyFenum.java*:

```
package myprojectquals;

import java.lang.annotation.*;
import checkers.quals.SubtypeOf;
import checkers.quals.TypeQualifier;

@Documented
@Retention(RetentionPolicy.RUNTIME)
@TypeQualifier
@SubtypeOf( { FenumTop.class } )
public @interface MyFenum {}
```

You only need to adapt the package, annotation, and file names in the example.

2. Use the provided `@Fenum` annotation, that takes a `String` argument to distinguish different fenums. For example, `@Fenum("A")` and `@Fenum("B")` are two distinct fenums.

The first approach allows you to define a short, meaningful name suitable for your project, whereas the second approach allows quick prototyping.

8.2 What the Fenum checker checks

The Fenum checker ensures that unrelated types are not mixed. All types with a particular fenum annotation, or `@Fenum(...)` with a particular `String` argument, are disjoint from all unannotated types and all types with a different fenum annotation or `String` argument.

The checker forbids method calls on fenum types and ensures that only compatible fenum types are used in comparisons and arithmetic operations (if applicable to the annotated type).

It is the programmer's responsibility to ensure that fields with a fenum type are properly initialized before use. Otherwise, one might observe a `null` reference or zero value in the field of a fenum type. (The Nullness checker (Chapter 3, page 18) can prevent failure to initialize a reference variable.)

8.3 Running the Fenum checker

The Fenum checker can be invoked by running the following commands.

- If you define your own annotation, provide the name of the annotation using the `-Aquals` option:

```
javac -processor checkers.fenum.FenumChecker  
      -Aquals=myprojectquals.MyFenum MyFile.java ...
```
- If your code uses the `@Fenum` annotation, you do not need the `-Aquals` option:

```
javac -processor checkers.fenum.FenumChecker MyFile.java ...
```

8.4 Suppressing warnings

One example of when you need to suppress warnings is when you initialize the fenum constants to literal values. To remove this warning message, add the corresponding `@SuppressWarnings` to either the field or class declaration, for example:

```
@SuppressWarnings("fenum:assignment.type.incompatible")  
class MyConsts {  
    public static final @Fenum("A") int ACONST1 = 1;  
    public static final @Fenum("A") int ACONST2 = 2;  
}
```

8.5 Example

The following example introduces two fenums in class `TestStatic` and then performs a few typical operations.

```
@SuppressWarnings("fenum:assignment.type.incompatible") // for initialization  
public class TestStatic {  
    public static final @Fenum("A") int ACONST1 = 1;  
    public static final @Fenum("A") int ACONST2 = 2;  
  
    public static final @Fenum("B") int BCONST1 = 4;  
    public static final @Fenum("B") int BCONST2 = 5;  
}
```

```

class FenumUser {
    @Fenum("A") int state1 = TestStatic.ACONST1;    // ok
    @Fenum("B") int state2 = TestStatic.ACONST1;    // Incompatible fenums forbidden!

    void fenumArg(@Fenum("A") int p) {}

    void foo() {
        state1 = 4;                                // Direct use of value forbidden!
        state1 = TestStatic.BCONST1;               // Incompatible fenums forbidden!
        state1 = TestStatic.ACONST2;               // ok

        fenumArg(5);                                // Direct use of value forbidden!
        fenumArg(TestStatic.BCONST1);              // Incompatible fenums forbidden!
        fenumArg(TestStatic.ACONST1);              // ok
    }
}

```

8.6 References

- **Java Language Specification on enums:**
<http://java.sun.com/docs/books/jls/third-edition/html/classes.html#8.9>
- **Tutorial trail on enums:**
<http://java.sun.com/docs/books/tutorial/java/java00/enum.html>
- **Typesafe enum pattern:**
<http://java.sun.com/developer/Books/shiftintojava/page1.html>
- **Avoiding enums for performance:**
<http://developer.android.com/guide/practices/design/performance.html#avoid.enums>
- **Java Tip 122: Beware of Java typesafe enumerations:**
<http://www.javaworld.com/javaworld/jwatips/jw-javatip122.html>

Chapter 9

Tainting checker

The tainting checker prevents certain kinds of trust errors. A *tainted*, or untrusted, value is one that comes from an arbitrary, possibly malicious source, such as user input or unvalidated data. In certain parts of your application, using a tainted value can compromise the application’s integrity, causing it to crash, corrupt data, leak private data, etc.

For example, a user-supplied pointer, handle, or map key should be validated before being dereferenced. As another example, a user-supplied string should not be concatenated into a SQL query, lest the program be subject to a SQL injection attack. A location in your program where malicious data could do damage is called a *sensitive sink*.

A program must “sanitize” or “untaint” an untrusted value before using it at a sensitive sink. There are two general ways to untaint a value: by checking that it is innocuous/legal (e.g., it contains no characters that can be interpreted as SQL commands when pasted into a string context), or by transforming the value to be legal (e.g., quoting all the characters that can be interpreted as SQL commands). A correct program must use one of these two techniques so that tainted values never flow to a sensitive sink. The Tainting Checker ensures that your program does so.

If the Tainting Checker issues no warning for a given program, then no tainted value ever flows to a sensitive sink. However, your program is not necessarily free from all trust errors. As a simple example, you might have forgotten to annotate a sensitive sink as requiring an untainted type, or you might have forgotten to annotate untrusted data as having a tainted type.

To run the Tainting Checker, supply the `-processor checkers.tainting.TaintingChecker` command-line option to `javac`.

9.1 Tainting annotations

The Tainting type system uses the following annotations:

- `@Untainted` indicates a type that includes only untainted, trusted values.
- `@Tainted` indicates a type that may include only tainted, untrusted values. `@Tainted` is a supertype of `@Untainted`.
- `@PolyTainted` is a qualifier that is polymorphic over tainting (see Section 16.1.2).

9.2 Tips on writing `@Untainted` annotations

Most programs are designed with a boundary that surrounds sensitive computations, separating them from untrusted values. Outside this boundary, the program may manipulate malicious values, but no malicious values ever pass the boundary to be operated upon by sensitive computations.

In some programs, the area outside the boundary is very small: values are sanitized as soon as they are received from an external source. In other programs, the area inside the boundary is very small: values are sanitized only immediately before being used at a sensitive sink. Either approach can work, so long as every possibly-tainted value is sanitized before it reaches a sensitive sink.

Once you determine the boundary, annotating your program is easy: put `@Tainted` outside the boundary, `@Untainted` inside, and `@SuppressWarnings("tainting")` at the validation or sanitization routines that are used at the boundary.

The Tainting Checker's standard default qualifier is `@Tainted` (see Section 16.3.1 for overriding this default). This is the safest default, and the one that should be used for all code outside the boundary (for example, code that reads user input). You can set the default qualifier to `@Untainted` in code that may contain sensitive sinks.

The Tainting Checker does not know the intended semantics of your program, so it cannot warn you if you mis-annotate a sensitive sink as taking `@Tainted` data, or if you mis-annotate external data as `@Untainted`. So long as you correctly annotate the sensitive sinks and the places that untrusted data is read, the Tainting Checker will ensure that all your other annotations are correct and that no undesired information flows exist.

As an example, suppose that you wish to prevent SQL injection attacks. You would start by annotating the `Statement` class to indicate that the `execute` operations may only operate on untainted queries (Chapter 18 describes how to annotate external libraries):

```
public boolean execute(@Untainted String sql) throws SQLException;
public boolean executeUpdate(@Untainted String sql) throws SQLException;
```

9.3 `@Tainted` and `@Untainted` can be used for many purposes

The `@Tainted` and `@Untainted` annotations have only minimal built-in semantics. In fact, the Tainting Checker provides only a small amount of functionality beyond the Basic Checker (Section 13). This lack of hard-coded behavior means that the annotations can serve many different purposes. Here are just a few examples:

- Prevent SQL injection attacks: `@Tainted` is external input, `@Untainted` has been checked for SQL syntax.
- Prevent cross-site scripting attacks: `@Tainted` is external input, `@Untainted` has been checked for JavaScript syntax.
- Prevent information leakage: `@Tainted` is secret data, `@Untainted` may be displayed to a user.

In each case, you need to annotate the appropriate untainting/sanitization routines. This is similar to the `@Encrypted` annotation (Section 13.2), where the cryptographic functions are beyond the reasoning abilities of the type system. In each case, the type system verifies most of your code, and the `@SuppressWarnings` annotations indicate the few places where human attention is needed.

If you want more specialized semantics, or you want to annotate multiple types of tainting in a single program, then you can copy the definition of the Tainting Checker to create a new annotation and checker with a more specific name and semantics. See Chapter 19 for more details.

Chapter 10

Linear checker for preventing aliasing

The Linear Checker implements type-checking for a linear type system. A linear type system prevents aliasing: there is only one (usable) reference to a given object at any time. Once a reference appears on the right-hand side of an assignment, it may not be used any more. The same rule applies for pseudo-assignments such as procedure argument-passing (including as the receiver) or return.

One way of thinking about this is that a reference can only be used once, after which it is “used up”. This property is checked statically at compile time. The single-use property only applies to use in an assignment, which makes a new reference to the object; ordinary field dereferencing does not use up a reference.

By forbidding aliasing, a linear type system can prevent problems such as unexpected modification (by an alias), or ineffectual modification (after a reference has already been passed to, and used by, other code).

To run the Linear Checker, supply the `-processor checkers.Linear.LinearChecker` command-line option to `javac`.

Figure 10.1 gives an example of the Linear Checker’s rules.

10.1 Linear annotations

The linear type system uses one user-visible annotation: `@Linear`. The annotation indicates a type for which each value may only have a single reference — equivalently, may only be used once on the right-hand side of an assignment.

The full qualifier hierarchy for the linear type system includes three types:

- `@UsedUp` is the type of references whose object has been assigned to another reference. The reference may not be used in any way, including having its fields dereferenced, being tested for equality with `==`, or being assigned to another reference. Users never need to write this qualifier.
- `@Linear` is the type of references that have no aliases, and that may be dereferenced at most once in the future. The type of `new T()` is `@Linear T` (the analysis does not account for the slim possibility that an alias to `this` escapes the constructor).
- `@NonLinear` is the type of references that may be dereferenced, and aliases made, as many times as desired. This is the default, so users only need to write `@NonLinear` if they change the default.

`@UsedUp` is a supertype of `@NonLinear`, which is a supertype of `@Linear`.

This hierarchy makes an assignment like

```
@Linear Object l = new Object();
@NonLinear Object nl = l;
@NonLinear Object nl2 = nl;
```

legal. In other words, the fact that an object is referenced by a `@Linear` type means that there is only one usable reference to it *now*, not that there will *never* be multiple usable references to it. (The latter guarantee would be possible to enforce, but it is not what the Linear Checker does.)

```

class Pair {
    Object a;
    Object b;
    public String toString() {
        return "<" + String.valueOf(a) + "," + String.valueOf(b) + ">";
    }
}

void print(@Linear Object arg) {
    System.out.println(arg);
}

@Linear Pair printAndReturn(@Linear Pair arg) {
    System.out.println(arg.a);
    System.out.println(arg.b);    // OK: field dereferencing does not use up the reference arg
    return arg;
}

@Linear Object m(Object o, @Linear Pair lp) {
    @Linear Object lo2 = o;        // ERROR: aliases may exist
    @Linear Pair lp3 = lp;
    @Linear Pair lp4 = lp;        // ERROR: reference lp was already used
    lp3.a;
    lp3.b;                        // OK: field dereferencing does not use up the reference
    print(lp3);
    print(lp3);                  // ERROR: reference lp3 was already used
    lp3.a;                       // ERROR: reference lp3 was already used
    @Linear Pair lp4 = new Pair(...);
    lp4.toString();
    lp4.toString();              // ERROR: reference lp4 was already used
    lp4 = new Pair();            // OK to reassign to a used-up reference
    // If you need a value back after passing it to a procedure, that
    // procedure must return it to you.
    lp4 = printAndReturn(lp4);
    if (...) {
        print(lp4);
    }
    if (...) {
        return lp4;              // ERROR: reference lp4 may have been used
    } else {
        return new Object();
    }
}

```

Figure 10.1: Example of Linear Checker rules.

10.2 Limitations

The `@Linear` annotation is supported and checked only on method parameters (including the receiver), return types, and local variables. Supporting `@Linear` on fields would require a sophisticated alias analysis or type system, and is future work.

No annotated libraries are provided for linear types. Most libraries would not be able to use linear types in their purest form. For example, you cannot put a linearly-typed object in a hashtable, because hashtable insertion calls `hashCode`; `hashCode` uses up the reference and does not return the object, even though it does not retain any pointers to the object. For similar reasons, a collection of linearly-typed objects could not be sorted or searched.

Our lightweight implementation is intended for use in the parts of your program where errors relating to aliasing and object reuse are most likely. You can use manual reasoning (and possibly an unchecked cast or warning suppression) when objects enter or exit those portions of your program, or when that portion of your program uses an unannotated library.

Chapter 11

Regex checker for regular expression syntax

The Regex Checker prevents, at compile-time, use of syntactically invalid regular expressions.

A regular expression, or regex, is a pattern for matching certain strings of text. In Java, a programmer writes a regular expression as a string. At run time, the string is “compiled” into an efficient internal form (`Pattern`) that is used for text-matching.

The syntax of regular expressions is complex, so it is easy to make a mistake. It is also easy to accidentally use a regex feature from another language that is not supported by Java (see section “Comparison to Perl 5” in the `Pattern` Javadoc). Ordinarily, the programmer does not learn of these errors until run time. The Regex checker warns about these problems at compile time.

To run the Regex Checker, supply the `-processor checkers.regex.RegexChecker` command-line option to `javac`.

11.1 Regex annotations

The Regex Checker uses one annotation only: `@Regex`, to indicate valid regular expression `Strings`.

The checker implicitly adds the `Regex` qualifier to any `String` literal that is a valid regex.

Chapter 12

Property file checker

The property file checker ensures that a property file or resource bundle (both of which act like maps from keys to values) is only accessed with valid keys. Accesses without a valid key either return `null` or a default value, which can lead to a `NullPointerException` or hard-to-trace behavior. The property file checker (Section 12.1, page 49) ensures that the used keys are found in the corresponding property file or resource bundle.

We also provide two specialized checkers. An internationalization checker (Section 12.2, page 50) verifies that code is properly internationalized. A compiler message key checker (Section 12.3, page 50) verifies that compiler message keys used in the Checker Framework are declared in a property file; This is an example of a simple specialization of the property file checker, and the Checker Framework source code shows how it is used.

It is easy to customize the property key checker for other related purposes. Take a look at the source code of the compiler message key checker and adapt it for your purposes.

12.1 Generic property file checker

The generic property file checker ensures that a resource key is located in a specified property file or resource bundle.

The annotation `@PropertyKey` indicates that the qualified `String` is a valid key found in the property file or resource bundle. You do not need to annotate `String` literals. The checker looks up every `String` literal in the specified property file or resource bundle, and adds annotations as appropriate.

If you pass a `String` variable to be eventually used as a key, you also need to annotate all these variables with `@PropertyKey`.

The checker can be invoked by running the following command:

```
javac -processor checkers.propkey.PropertyKeyChecker
      -Abundlenames=MyResource MyFile.java ...
```

You must specify the resources, which map keys to strings. The checker supports two types of resource: resource bundles and property files. You can specify one or both of the following two command-line options:

1. `-Abundlenames=resource_name`
resource_name is the name of the resource to be used with `ResourceBundle.getBundle()`. The checker uses the default `Locale` and `ClassLoader` in the compilation system. (For a tutorial about `ResourceBundles`, see <http://java.sun.com/developer/technicalArticles/Intl/ResourceBundles/>.) Multiple resource bundle names are separated by colons `':'`.
2. `-Apropfiles=prop_file`
prop_file is the name of a properties file that maps keys to values. The file format is described in the Javadoc for `Properties.load()`. Multiple files are separated by colons `':'`.

12.2 Internationalization checker

The Internationalization Checker verifies that your code is properly internationalized. Internationalization is the process of adapting software to different languages and locales. Internationalization is sometimes called localization (though the terms are not identical), and is sometimes called *i18n* (because the word starts with “i”, ends with “n”, and has 18 characters in between; localization is similarly sometimes abbreviated as *l10n*).

The checker focuses on one aspect of internationalization: user-visible strings should be presented in the user’s own language, such as English, French, or German. This is achieved by looking up keys in a localization resource, which maps keys to user-visible strings. For instance, one version of a resource might map “CANCEL_STRING” to “Cancel”, and another version of the same resource might map “CANCEL_STRING” to “Abbrechen”.

There are other aspects to localization, such as formatting of dates (3/5 vs. 5/3 for March 5), that the checker does not check.

The Internationalization Checker verifies these two properties:

1. Any user-visible text should be obtained from a localization resource. For example, `String` literals should not be output to the user.
2. When looking up keys in a localization resource, the key should exist in that resource. This check catches incorrect or misspelled localization keys.

12.2.1 Internationalization annotations

The Internationalization Checker supports two annotations:

1. `@Localized`: indicates that the qualified `String` is a message that has been localized and/or formatted with respect to the used locale.
2. `@LocalizableKey`: indicates that the qualified `String` or `Object` is a valid key found in the localization resource. This annotation is a specialization of the `@PropertyKey` annotation, that gets checked by the generic property key checker.

You may need to add the `@Localized` annotation to more methods in the JDK or other libraries, or in your own code.

12.2.2 Running the Internationalization Checker

The Internationalization Checker can be invoked by running the following command:

```
javac -processor checkers.i18n.I18nChecker -Abundlenames=MyResource MyFile.java ...
```

You must specify the localization resource, which maps keys to user-visible strings. Like the generic property key checker, the internationalization checker supports two types of localization resource: `ResourceBundles` using the `-Abundlenames=resource_name` option or property files using the `-Apropfiles=prop_file` option.

12.3 Compiler Message Key checker

The Checker Framework uses compiler message keys to output error messages. These keys are substituted by localized strings for user-visible error messages. Using keys instead of the localized strings in the source code enables easier testing, as the expected error keys can stay unchanged while the localized strings can still be modified. We use the compiler message key checker to ensure that all internal keys are correctly localized. Instead of using the property file checker, we use a specialized checker, giving us more precise documentation of the intended use of `Strings`.

The single annotation used by this checker is `@CompilerMessageKey`. The Checker Framework is completely annotated; for example, class `checkers.source.Result` uses `@CompilerMessageKey` in methods `failure` and `warning`. For most users of the Checker Framework there will be no need to annotate any `Strings`, as the checker looks up all `String` literals and adds annotations as appropriate.

The compiler message key checker can be invoked by running the following command:

```
javac -processor checkers.compilermsgs.CompilerMessagesChecker  
      -Apropfiles=messages.properties MyFile.java ...
```

You must specify the resource, which maps compiler message keys to user-visible strings. The checker supports the same options as the generic property key checker. Within the Checker Framework we only use property files, so the `-Apropfiles=prop_file` option should be used.

Chapter 13

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 Chapter 19). 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 16.3.2) and qualifier polymorphism (Section 16.1.2).

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.

If you need typestate analysis, then you can extend a typestate checker, much as you would extend the Basic Checker if you do not need typestate analysis. For more details (including a definition of “typestate”), see Chapter 14.

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 Chapter 19.

13.1 Using the Basic checker

The Basic checker is used in the same way as other checkers (using the `-processor checkers.basic.BasicChecker` option; see Chapter 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 19.6).

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`.

13.2 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;

import checkersquals.*;

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

```

Don't forget to compile this class:

```
$ javac myquals/Encrypted.java
```

The resulting `.class` file should either be on your classpath, or on the processor path (set via the `-processorpath` command-line option to `javac`).

2. Write `@Encrypted` annotations in your program (`YourProgram.java`):

```

import myquals.Encrypted;

...

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. Analyzing them is beyond the capability of any realistic type system.

3. Invoke the compiler with the Basic checker, specifying the `@Encrypted` annotation using the `-Aquals` option. You should add the `Encrypted` classfile to the processor classpath:

```
$ javac -processorpath myqualspath -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);  
    ^
```

Chapter 14

Typestate checker

In a regular type system, a variable has the same type throughout its scope. In a typestate system, a variable's type can change as operations are performed on it.

The most common example of typestate is for a `File` object. Assume a file can be in two states, `@Open` and `@Closed`. Calling the `close()` method changes the file's state. Any subsequent attempt to read, write, or close the file will lead to a run-time error. It would be better for the type system to warn about such problems, or guarantee their absence, at compile time.

Just as you can extend the Basic Checker to create a type checker, you can extend a typestate checker to create a type checker that supports typestate analysis. Two extensible typestate analyses that build on the Checker Framework are available. One is by Adam Warski: <http://www.warski.org/typestate.html>. The other is by Daniel Wand: <http://typestate.ewand.de/>.

14.1 Comparison to flow-sensitive type refinement

The Checker Framework's flow-sensitive type refinement (Section 16.3.2) implements a form of typestate analysis. For example, after code that tests a variable against null, the Nullness Checker (Chapter 3) treats the variable's type as `@NonNull T`, for some `T`.

For many type systems, flow-sensitive type refinement is sufficient. But sometimes, you need full typestate analysis. This section compares the two. (Dependent types and unused variables (Section 16.2) also have similarities with typestate analysis and can occasionally substitute for it. For brevity, this discussion omits them.)

A typestate analysis is easier for a user to create or extend. Flow-sensitive type refinement is built into the Checker Framework and is optionally extended by each checker. Modifying the rules requires writing Java code in your checker. By contrast, it is possible to write a simple typestate checker declaratively, by writing annotations on the methods (such as `close()`) that change a reference's typestate.

A typestate analysis can change a reference's type to something that is not consistent with its original definition. For example, suppose that a programmer decides that the `@Open` and `@Closed` qualifiers are incomparable — neither is a subtype of the other. A typestate analysis can specify that the `close()` operation converts an `@Open File` into a `@Closed File`. By contrast, flow-sensitive type refinement can only give a new type that is a subtype of the declared type — for flow-sensitive type refinement to be effective, `@Closed` would need to be a child of `@Open` in the qualifier hierarchy (and `close()` would need to be treated specially by the checker).

Chapter 15

External checkers

The checker framework has been used to build other checkers that are not distributed together with the framework.

If you want a reference to your checker included in this chapter, send us a link and short description of your checker,

15.1 Units and dimensions checker

A checker for units and dimensions is available at <http://www.lexspoon.org/expannots/>.

Chapter 16

Advanced type system features

This section describes features that are automatically supported by every checker written with the Checker Framework. You may wish to skim or skip this section on first reading. After you have used a checker for a little while and want to be able to express more sophisticated and useful types, or to understand more about how the Checker Framework works, you can return to it.

16.1 Polymorphism and generics

16.1.1 Generics (parametric polymorphism or type polymorphism)

The Checker Framework fully supports type-qualified Java generic types (also known in the research literature as “parametric polymorphism”). Before running any checker, we recommend that you eliminate raw types from your code (e.g., your code should use `List<...>` as opposed to `List`). 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>`.

Restricting instantiation of a generic class There are two ways to restrict the type qualifiers that may be used on the actual type argument when instantiating a generic class.

The first technique is the standard Java approach of using the `extends` or `super` clause to supply an upper or lower bound. For example:

```
MyClass<T extends @NonNull Object> { ... }

MyClass<@NonNull String> m1;      // OK
MyClass<@Nullable String> m2;    // error
```

The second technique is to write a type annotation on the declaration of a generic type parameter, which specifies the exact annotation that is required on the actual type argument, rather than just a bound. For example:

```
class MyClassNN<@NonNull T> { ... }
class MyClassNble<@Nullable T> { ... }

MyClassNN<@NonNull Number> v1;    // OK
MyClassNN<@Nullable Number> v2;  // error
MyClassNble<@NonNull Number> v4;  // error
MyClassNble<@Nullable Number> v3; // OK
```

A way to view a type annotation on a generic type parameter declaration is as syntactic sugar for the annotation on both the extends and the super clauses of the declaration. For example, these two declarations have the same effect:

```
class MyClassNN<@NonNull T> { ... }
class MyClassNN<T extends @NonNull Object super @NonNull void> { ... }
```

except that the latter is not legal Java syntax. The syntactic sugar is necessary for two reasons: it is illegal to specify both the upper and the lower bound, and it is impossible to specify a type annotation for a lower bound without also specifying a type (use of `void` is illegal).

If a type parameter declaration is annotated with `@A`, and a bound is also given, then the annotation applies everywhere that there is no explicit annotation. For example, the following pairs of declarations are identical.

```
class MyClassNN<@A T> { ... }
class MyClassNN<T extends @A Object super @A void> { ... }

class MyClassNN<@A T extends Number> { ... }
class MyClassNN<T extends @A Number super @A void> { ... }

class MyClassNN<@A T extends @B Number> { ... }
class MyClassNN<T extends @B Number super @A void> { ... }

class MyClassNN<@A T super Number> { ... }
class MyClassNN<T extends @A Object super @A Number> { ... }

class MyClassNN<@A T super @B Number> { ... }
class MyClassNN<T extends @A Object super @B Number> { ... }
```

Note that these types mean different things:

```
class MyList1<T extends @Nullable Object> { ... }
class MyList2<@NonNull T> { ... }
```

Consider the implementation of the list (in the ellipsis). The implementation of `MyList2` may only place non-null objects in the list and may assume that retrieved elements are non-null. The implementation of `MyList1` is similar in that it may only place non-null objects in the list, because it might be instantiated as, say, `MyList1<@NonNull Date>`. When retrieving elements from the list, the implementation of `MyList1` must account for the fact that elements of `MyList1` may be null, because it might be instantiated as, say, `MyList1<@Nullable Date>`. The differences are more significant when the qualifier hierarchy is more complicated than just `@Nullable` and `@NonNull`.

Defaults for bounds Ordinarily, a type parameter declaration with no extends clause means the type parameter can be instantiated with any type argument at all. For example:

```
class C<T> { ... }
class C<T extends Object> { ... } // identical to previous line
```

However, instantiation may be restricted if a default qualifier is in effect (see Section 16.3.1). For example, the Nullness checker (Chapter 3) uses a (configurable) default of `@NonNull` (see Section 3.2.2). That means that either declaration above is interpreted as

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

and an instantiation such as `C<@Nullable Number>` is illegal. In such a case, to permit all type arguments, the programmer would write

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

It is possible to set the default qualifier for upper bounds separately from other default qualifiers, by writing an annotation such as `@DefaultQualifier(value="Nullable", locations=DefaultLocation.UPPER_BOUNDS)`.

Type annotations on a use of a generic type variable 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, suppose that `T` is a formal type parameter. Then using `@Nullable T` within the scope of `T` applies the type qualifier `@Nullable` to the (unqualified) Java type of `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`.

16.1.2 Qualifier polymorphism

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

To *define* a polymorphic qualifier, mark the definition with `@PolymorphicQualifier`. For example, `@PolyNull` is a polymorphic type qualifier for the Nullness type system:

```
@PolymorphicQualifier  
@Target(ElementType.TYPE_USE)  
public @interface PolyNull { }
```

To *use* a polymorphic qualifier, just write it on a type. For example, you can write `@PolyNull` anywhere that you would write `@NonNull` or `@Nullable`.

A method written using a polymorphic qualifier conceptually has multiple versions, somewhat like a template in C++ or the generics feature of Java. In each version, each instance of the polymorphic qualifier has been replaced by the same other qualifier from the hierarchy. See the examples below in Section 16.1.2.

The method body must type-check with all signatures. A method call is type-correct if it type-checks under any one of the signatures. If a call matches multiple signatures, then the compiler uses the most specific matching signature for the purpose of type-checking. This is just like Java's rule for resolving overriding methods, though there is no effect on run-time dispatch or behavior.

Polymorphic qualifiers can be used on a method signature or 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);
```

Another 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 supertype of both, also known as the *least upper bound* or *lub*. If both arguments are `@NonNull`, their unification (lub) is `@NonNull`, and the method return type is `@NonNull`. But if even one of the arguments is `@Nullable`, then the unification (lub) is `@Nullable`, and so is the return type.

Use multiple polymorphic qualifiers in a method signature Usually, it does not make sense to write only a single instance of a polymorphic qualifier in a method definition: if you write one instance of (say) `@PolyNull`, then you should use at least two. (An exception is a polymorphic qualifier on an array element type; this section ignores that case, but see below for further details.)

For example, there is no point to writing

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

which expands to

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

This is no different (in terms of which calls to the method will type-check) 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 at least one formal parameter and also on the return type. It can also be useful to have polymorphic qualifiers on (only) multiple formal parameters, especially if the method side-effects one of its arguments. For example, consider

```
void moveBetweenStacks(Stack<@PolyNull Object> s1, Stack<@PolyNull Object> s2) {
    s1.push(s2.pop());
}
```

In this example, if it is acceptable to rewrite your code to use Java generics, the code can be even cleaner:

```
<T> void moveBetweenStacks(Stack<T> s1, Stack<T> s2) {
    s1.push(s2.pop());
}
```

Using a single polymorphic qualifier on an element type There is an exception to the general rule that a polymorphic qualifier should be used multiple times in a signature. It can make sense to use a polymorphic qualifier just once, if it is on an array or generic element type.

For example, consider a routine that returns the first index, in an array or collection, of a given element:

```
public static int indexOf(@PolyNull Object[] a, Object elt) { ... }

public static int indexOf(Collection<@PolyNull Object> a, Object elt) { ... }
```

If `@PolyNull` were replaced with either `@Nullable` or `@NonNull`, then some safe client calls would be rejected.

Of course, it would be better style to use a generic method, as in either of these signatures (and likewise for the `Collection` version):

```
public static <T> int indexOf(T[] a, /*@Nullable*/ Object elt) { ... }
public static <T> int indexOf(T[] a, T elt) { ... }
```

In conclusion, use of a single polymorphic qualifier may be necessary in legacy code, but can be avoided by use of better code style.

16.2 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: a field that may not be used if the receiver has a given qualifier, and a fields whose qualifier changes based on the qualifier of the receiver. (Also see the discussion of `typestate` checkers, in Chapter 14.)

16.2.1 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).

16.2.2 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 16.1). 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`.

16.2.3 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 for `Person` 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 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
...
```

16.3 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. In particular, qualifiers in method bodies are extremely rare.

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 19.4), 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 annotation 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 16.3.1.

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 16.3.2.

16.3.1 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 19.3.3).
- 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, but type system implementers can rely on the fact that some qualifier is present.

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 `String` name of an annotation. It may be a short name like `"NonNull"`, if an appropriate import statement exists. Otherwise, it should be fully-qualified, like `"checkers.nullnessquals.NonNull"`. The 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.

For example, using the Nullness type system (Chapter 3):

```
import checkers.quals.*;          // for DefaultQualifier[s]
import checkers.nullnessquals.NonNull;

@DefaultQualifier("NonNull"),
class MyClass {

    public boolean compile(File myFile) { // myFile has type "@NonNull 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("Mutable")
    public boolean isJavaFile(File myfile) { // myFile has type "@Mutable File"
        ...
    }
}
```

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

```
@DefaultQualifiers({
    @DefaultQualifier("NonNull"),
    @DefaultQualifier("Mutable")
})
```

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

Recall that an annotation on a class definition indicates an implicit qualifier (Section 16.3) that can only be strengthened, not weakened. This can lead to unexpected results if the default qualifier applies to a class definition. Thus, you may want to put explicit qualifiers on class declarations (which prevents the default from taking effect), or exclude class declarations from defaulting.

When a programmer omits an `extends` clause at a declaration of a type parameter, the default still applies to the implicit upper bound. For example, consider these two declarations:

```
class C<T> { ... }
class C<T extends Object> { ... } // identical to previous line
```

The two declarations are treated identically by Java, and the default qualifier applies to the `Object` upper bound whether it is implicit or explicit. (The `@NonNull` default annotation applies only to the upper bound in the `extends` clause, not to the lower bound in the inexpressible implicit `super void` clause.)

16.3.2 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 16.3.1) 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 programmer 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 (Chapter 3) can automatically determine that certain variables are non-null, even if they were explicitly or by default annotated as nullable. The checker treats a variable or expression as `@NonNull`

- starting at 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 non-null types, without compiler warnings.

Consider this code, along with comments indicating whether the Nullness checker (Chapter 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(@Nullable 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 examples 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 22.2) 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).

16.3.3 Fields and flow sensitivity analysis

Flow sensitivity analysis infers the type of fields in some restricted cases:

- A final initialized field: Type inference is performed for final fields that are initialized to a compile-time constant at the declaration site; so the type of `protocol` is `@NonNull String` in the following declaration:

```
public final String protocol = "https";
```

Please note that such inferred type may leak to the public interface of the class. To override such behavior, you can explicitly insert the desired annotation, e.g.

```
public final @Nullable String protocol = "https";
```

- Within method bodies: Type inference is performed for fields in the context of method bodies, like local variables, but method invocations invalidate any inferred information. Consider the following example, where `name` is a nullable field:

```
class DBObject {
    @Nullable Date updatedAt;

    void update() {
        if (updatedAt == null)
            updatedAt = new Date();
        // updatedAt is nonnull
        log("Updating object at " + updatedAt.getTime());

        persistData();
        // updatedAt is nullable again
        log.debug("Saved object updated at " + updatedAt.getTime()); // invalid!
    }
}
```

Here the call to `persistData()` invalidates the inferred non-null type of `updatedAt`.

When methods do not modify any object state or have any identity side-effects (e.g. `log()` method here), you can annotate these methods as `Pure`. Annotating them as `Pure`, would cause the flow analyzer to carry the inferred types across the method invocation boundary.

16.4 Inexpressible types

The Type Annotations syntax [Ern08] is designed to be easy to read. As a result, there are types that it cannot express. An example is the type of `Collection.toArray()`, which returns an array of objects, where the objects have the same annotation as the elements of the receiver.

A possible annotation would be

```
public @Polynull Object [] toArray() ArrayList<@PolyNull E> { ... }
```

except that this is illegal syntax: “`ArrayList<@PolyNull E>`” is not legal in the receiver position. (This is a motivation for extending the Type Annotations syntax.)

The annotated libraries (Section 18) contain a less-precise annotation for `toArray`. The Nullness Checker special-cases `toArray` to act as if it had the above annotation. The cases that are currently being handled are described in `CollectionToArrayHeuristics`. This approach would be possible for other checkers and other methods as the need arises.

Chapter 17

Handling warnings and legacy code

Section 2.4.1 describes a methodology for applying annotations to legacy code. This chapter tells you what to do if, for some reason, you cannot change your code in such a way as to eliminate a checker warning.

17.1 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 enough knowledge to annotate poorly-documented libraries that your program uses.

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

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

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 strongest guarantees, but may require you to annotate additional methods that `unannotated_method` calls. See Chapter 18 for a discussion of how to annotate libraries for which you have no source code.
2. Annotate only the signature of `unannotated_method`, and suppress warnings in its body. Two ways to suppress the warnings are via a `@SuppressWarnings` annotation or by not running the checker on that file (see Section 17.2).
3. Suppress all warnings related to uses of `unannotated_method` via the `skipClasses` processor option (see Section 17.2). 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 limited number of classes/methods (e.g., the public interface to a library or package).

Chapter 18 discusses adding annotations to signatures when you do not have source code available. Section 17.2 discusses suppressing warnings.

If you annotate a third-party library, please share it with us so that we can distribute the annotations with the Checker Framework; see Section 22.2.

17.2 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 limita-

tions, 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,
- not using the `-processor` switch to `javac`, or
- checker-specific mechanisms.

We now explain these mechanisms in turn.

You can suppress specific errors and warnings by use of the `@SuppressWarnings("checkername")` annotation, for example `@SuppressWarnings("interning")` or `@SuppressWarnings("nullness")`. This may be placed on program elements such as a local variable declaration, a method, or a class. For instance, one common use is to suppress warnings at a cast that you know is safe. Here is an example that uses the Tainting Checker (Section 9):

```
@SuppressWarnings("tainting")
String myvar = (@Untainted String) expr; // expr has type: @Tainted String
```

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 (but the class itself is still type-checked). 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 classes whose fully-qualified name starts with `java.`, all warnings relating to invalid arguments, and all warnings relating to incorrect use of the return value.

To suppress all errors and warnings related to multiple classes, you can use the regular expression alternative operator `|`, as in `"-AskipClasses="java\.lang\.|java\.util\."` to suppress all warnings related to classes belong to the `java.lang` or `java.util` packages.

The `-Alint` option enables or disables optional checks, analogously to `javac`'s `-Xlint` option. Each of the distributed checkers supports at least the following lint options:

- `cast:unsafe` (default: on) warn about unsafe casts that are not checked at run time, as in `((@NonNull String) myref)`. Such casts are generally not necessary when flow-sensitive local type refinement is enabled.
- `cast:redundant` (default: on) warn about redundant casts that are guaranteed to succeed at run time, as in `((@NonNull String) "m")`. Such casts are not necessary, because the target expression of the cast already has the given type qualifier.
- `cast` Enable or disable all cast-related warnings.
- `all` Enable or disable all lint warnings, including checker-specific ones if any. Examples include `nulltest` for the Nullness Checker (see Section 3) and `dotequals` for the Interning Checker (see Section 4.3). This option does not enable/disable the checker's standard checks, just its optional ones.
- `none` The inverse of `all`: disable or enable all lint warnings, including checker-specific ones if any.

To activate a lint option, write `-Alint=` followed by a comma-delimited list of check names. If the option is preceded by a hyphen (`-`), the warning is disabled. For example, to disable all lint options except redundant casts, you can pass `-Alint=-all,cast:redundant` on the command line.

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 (Chapter 3) uses `assert` statements that contain null checks, and the special `castNonNull` method, to suppress warnings (Section 3.4.1).

17.3 Writing annotations in comments for backward compatibility

Sometimes, your code needs to be compilable by people who are not using a Java 7 compiler.

17.3.1 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, extends clauses, method receiver, etc.).

So that your code can be compiled by any Java compiler (for any version of the Java language), 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.

This feature only works if you provide no `-source` command-line argument to `javac`, or if the `-source` argument is 1.7 or 7.

(**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 ignores annotations written in comments. This is the key 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.

17.3.2 Implicit 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`.

The traditional way to do this is to write an import statement like `"import checkers.nullness.quals.*;"`. This works, but everyone who compiles the code (no matter what compiler they use, and even if the annotations are in comments) 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.

An alternative is to set the shell environment variable `jsr308_imports` when you compile the code. The Type Annotations compiler treats this as if the given packages were imported, but other compilers ignore the `jsr308_imports` environment variable — they do not need it, since they do not support annotations in comments. Thus, your code can compile whether or not the Type Annotations compiler is being used.

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.*`.

If you issue the `javac` command from the command line or in a Makefile, you may need to add quotes, to prevent your shell from expanding the `*` character. 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 environment variable via the `javac` command-line argument `-J-Djsr308_imports='checkers.nullness.quals.*'`. If you supply the `-J-Djsr308_imports` argument via an Ant buildfile, you do not need the extra quoting.

17.3.3 Migrating away from annotations in comments

Suppose that your codebase currently uses annotations in comments, but you wish to remove the comment characters around your annotations, because in the future you will use only Java 7 compilers. This Unix command removes the comment characters, for all Java files in the current working directory or any subdirectory.

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

You can customize this command:

- To process comments with embedded spaces and asterisks, change two instances of “[^ \s/]” to “[^ \s/]*”.
- 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 17.3.2), you may also need to introduce explicit import statements into your code.

Chapter 18

Annotating libraries

When annotated code uses an unannotated library, a checker may issue warnings. As described in Section 17.1, the best way to correct this problem is to add annotations to the library. (Alternately, you can instead suppress all warnings related to an unannotated library by use of the `-AskipClasses` command-line option; see Section 17.2.) If you have source code for the library, you can easily add the annotations. This section tells you how to add annotations to a library for which you have no source code, because the library is distributed only in binary (`.class` or `.jar`) form. This section is also useful if you do not wish to edit the library's source code.

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 22.2.

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 3.2.4) 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” 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 18.1 describes how to use the stub class generation tools.
- You can annotate the compiled `.jar` or `.class` files using the Annotation File Utilities (<http://types.cs.washington.edu/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.

18.1 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 18.1.1 describes how to create stub classes. Section 18.1.2 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 18.1.3.)

18.1.1 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.interningquals.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 18.1.3.

18.1.2 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 ...
```

Checker and library writers are encouraged to distribute stub files for the JDK and most commonly used libraries. Section 18.1.4 explains how to bundle a stub file, such that they get imported automatically. Programmers should only explicitly specify the stub files they create themselves.

18.1.3 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 (Chapter 4) is as follows. This file appears as `checkers/src/checkers/interning/jdk.astub` in the Checker Framework distribution.

```
import checkers.interningquals.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. Every valid Java file is a valid 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 is allowed to differ 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.)

Declaration specifiers: Declaration specifiers (e.g., `public`, `final`, `volatile`) may be omitted, since they have nothing to do with types.

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.

18.1.4 Distributing stub files

A distributed stub file doesn't need to be explicitly specified by the programmer; the checker automatically uses it.

A checker writer should include the JDK stub file as `jdk.astub` in the same directory level as the Checker class (i.e., the subclass of `BaseTypeVisitor`). The Checker Framework imports the `jdk.astub` automatically.

18.1.5 Known problems

The Checker Framework stub file reader has several limitations:

- It does not handle `enums`.
- It only handles type annotations, not declaration annotations (e.g. IJ's `Assignable`).

18.1.6 Style tips for stub files

Every Java file is a stub file. If you have access to the Java file, then it is usually best to use the Java file as the stub file, without removing any of the parts that the stub file format permits you to. Just add annotations to the full source code. This approach retains the original documentation and source code, making it easier for a programmer to double-check the annotations. It also enables creation of diffs, easing the process of upgrading when a library adds new methods. And, the annotations are in a format that the library maintainers can even incorporate.

The downside of this approach is that the stub files are larger. This can slow down parsing. Furthermore, a programmer must search the stub file for a given method rather than just skimming one or two pages of signatures.

If you do not have access to the library source code, then you can create a stub file from the Javadoc or the class file, and then annotate it.

18.2 Using distributed annotated JDKs

The Checker Framework distribution contains annotated JDKs at the path `checkers/jdk/jdk.jar`.

1. When you run `javac`, add a `-bootclasspath/p:` argument to indicate where to find the annotated JDK classes. Supply `-bootclasspath/p` in addition to whatever other arguments you usually use, including `-classpath`. The `-bootclasspath/p:` argument causes the compiler to read annotations from annotated JDK classes in preference to the unannotated original library classes.

```
javac -processor checkers.nullness.NullnessChecker -Xbootclasspath/p:checkers/jdk/jdk.jar my_source_files
```

2. Run the compiled code as usual. The annotated JDK does not need to be in your classpath at run time.

Please note that so far, only three checkers require passing the annotated JDK explicitly, and they are the Nullness, the Javari, and the IGJ checkers.

Chapter 19

How to create a new checker

This section describes how 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)
@Target({ElementType.TYPE_PARAMETER, ElementType.TYPE_USE})
public @interface Encrypted {}
```

This checker is so short because it builds on the Basic Checker (Chapter 13). See Section 13.2 for more details about this particular checker. When you wish to create a new checker, it is sometimes easiest to begin by building it declaratively on top of the Basic Checker, and then return to this chapter when you need more expressiveness or power than the Basic Checker affords.

You can also customize a typestate checker, which enables a variable’s type to change — for instance, a file might transition from the `@Open` to the `@Closed` state after the `close()` method is called. For more details, see Chapter 14.

The rest of this section contains many details for people who want to write more powerful checkers. You do not 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. You can even create your checker by modifying one of those. The Javadoc documentation of the framework and the checkers is in the distribution and is also available online at <http://types.cs.washington.edu/checker-framework/current/doc/>.

If you write a new checker and wish to advertise it to the world, let us know so we can mention it in the Checker Framework manual, link to it from the webpages, or include it in the Checker Framework distribution.

19.1 Relationship of the Checker Framework to other tools

This table shows the relationship among various tools. All of the tools use the Type Annotations (JSR 308) syntax. You use the Checker Framework to build pluggable type systems, and the Annotation File Utilities to manipulate `.java` and `.class` files.

Nullness Checker	Mutation Checker	Tainting Checker	...	Your Checker	Type inference	Other tools
Checker Framework (enables creation of pluggable type-checkers)					Annotation File Utilities (.java ↔ .class files)	
Type Annotations syntax and classfile format (“JSR 308”) (no built-in semantics)						

(Strictly speaking, the specific checkers, such as the Nullness Checker, are built on top of the Basic Checker, which is built on top of the Checker Framework. The Basic Checker can also be used directly by users.)

19.2 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 19.3–19.6 describe the components of a type system as written using the Checker Framework:

19.3 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`).

19.4 Type introduction rules. For some types and expressions, a qualifier should be treated as implicitly 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`.

19.5 Type rules. You specify the type system semantics (type rules), violation of which yields a type error. There are two types of rules.

- Subtyping rules related to the type hierarchy, such as that every assignment and pseudo-assignment satisfies a subtyping relationship. Your checker automatically inherits these subtyping rules from the Basic Checker (Chapter 13).
- Additional rules that are specific to your particular checker. For example, in the Nullness type system, only references with a `@NonNull` type may be dereferenced. You write these additional rules yourself.

19.6 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.

19.3 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` meta-annotation on the annotation definition to indicate that the annotation represents a type qualifier and should be processed by the checker. Also write a `@Target` meta-annotation to indicate where the annotation may be written. For example:

```
// Define an annotation for the @NonNull type qualifier.
@TypeQualifier
@Target({ElementType.TYPE_PARAMETER, ElementType.TYPE_USE})
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 19.3.1), or procedurally through subclassing `QualifierHierarchy` or `TypeHierarchy` (Section 19.3.2).

19.3.1 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 a 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
@Target({ElementType.TYPE_PARAMETER, ElementType.TYPE_USE})
@SubtypeOf( { Nullable.class } )
public @interface NonNull { }

```

`@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.

All type qualifiers, except for polymorphic qualifiers (see below and also Section 16.1.2), need to be properly annotated with `SubtypeOf`.

The root qualifier should be annotated with `@SubtypeOf({ })`. The root qualifier is the qualifier that is a supertype of all other qualifiers. For example, `@Nullable` is the root of the Nullness type system, hence is defined as:

```

@TypeQualifier
@Target({ElementType.TYPE_PARAMETER, ElementType.TYPE_USE})
@SubtypeOf( { } )
public @interface Nullable { }

```

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 13.2.

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

```

@TypeQualifier
@Target({ElementType.TYPE_PARAMETER, ElementType.TYPE_USE})
@PolymorphicQualifier
public @interface PolyNull { }

```

For a description of polymorphic qualifiers, see Section 16.1.2. 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 19.3.2) by overriding the `isSubtype` method in the checker class (Section 19.6). However, the declarative mechanism is sufficient for most type systems.

19.3.2 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 the type hierarchy — that is, 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 wildcards 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” or “novariant”.) 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<@ReadOnly Date>`).

19.3.3 Defining a default annotation

A type system designer may set a default annotation. A user may override the default; see Section 16.3.1.

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 16.3 and 19.4.

19.3.4 Completeness of the type hierarchy

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. For instance, the hierarchy of Figure 5.1 lacks a bottom qualifier, because there is no qualifier that is a subtype of both `@Immutable` and `@Mutable`. The bottom qualifier is the natural type for the `null` value, which can be viewed as having any type at all. Without a bottom qualifier, type-checking becomes less precise. Users should never write the bottom qualifier explicitly; it is merely used for the `null` value.

The actual IJG hierarchy contains a (non-user-visible) bottom qualifier, defined like this:

```
@TypeQualifier
@SubtypeOf({Mutable.class, Immutable.class, I.class})
@Target({}) // forbids a programmer from writing it in a program
@ImplicitFor(trees = { Kind.NULL_LITERAL, Kind.CLASS, Kind.NEW_ARRAY },
             typeClasses = { AnnotatedPrimitiveType.class })
@interface IGJBottom { }
```

Top qualifier Similarly, it is a good idea to have a top qualifier in your type hierarchy — a qualifier that is a (direct or indirect) supertype of every other qualifier. For instance, the `@Encrypted` type system of Section 19.3.4 lacks a top qualifier:

```
@TypeQualifier
@SubtypeOf(Unqualified.class)
@Target({ElementType.TYPE_PARAMETER, ElementType.TYPE_USE})
public @interface Encrypted { }
```

The interning type system of Section 4 also lacks a top qualifier; there is no `@Uninterned` qualifier that is a supertype of `@Interned`.

When a type system lacks a top qualifier (or any other qualifier), then users lose flexibility in expressing defaults. In the encryption example there is no top qualifier, and every type is either unqualified or has the `@Encrypted` qualifier. Another way of saying this is that the default is `@Unqualified`. In such a case, it is not sensible for a user to specify a default for unannotated types via the `@DefaultQualifier` meta-annotation (Section 16.3.1), because there is no argument to pass to it. `@Unqualified` is not appropriate, because it is not clear which type system it is intended to refer to.

The ability to omit the top qualifier is a convenience when writing a type system, because it reduces the number of qualifiers that must be defined; this is especially convenient when using the Basic Checker (Section 13). Omitting the top qualifier also restricts the user in ways that the type system designer may have intended.

However, a type system designer should not frequently omit the top qualifier. It is better if 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.

19.4 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.

19.4.1 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 `javac.lang.model.type.TypeMirror`, e.g., `javac.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
    })
@Target({ElementType.TYPE_PARAMETER, ElementType.TYPE_USE})
public @interface NonNull { }

@TypeQualifier
@SubtypeOf({})
@ImplicitFor(trees={Tree.Kind.NULL_LITERAL})
@Target({ElementType.TYPE_PARAMETER, ElementType.TYPE_USE})
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 19.9 can help you get started.)

19.4.2 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.

19.5 Visitor: Type rules

A type system's rules define which operations on values of a particular type are forbidden. These rules must be defined procedurally, not declaratively.

The Checker 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 it issues a warning whenever the type system 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 Chapter 3 contains 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`) for 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

19.5.1 AST traversal

The Checker Framework needs to do its own traversal of the AST even though it operates as an ordinary annotation processor [Dar06]. Annotation processors can utilize a visitor for Java code, but that visitor only visits the public elements of Java code, such as classes, fields, methods, and method arguments — it does not visit code bodies or various other locations. The Checker Framework hardly uses the built-in visitor — as soon as the built-in visitor starts to visit a class, then the Checker Framework's visitor takes over and visits all of the class's source code.

Because there is no standard API for the AST of Java code, the Checker Framework uses the `javac` implementation. This is why the Checker Framework is not deeply integrated with Eclipse, but runs as an external tool (see Section 20.5). Actually, there is standard API for Java ASTs — JSR 198 (Extension API for Integrated Development

Environments) [Cro06]. If tools were to implement it (which would just require writing wrappers or adapters), then the Checker Framework and similar tools could be portable among different compilers and IDEs.

19.5.2 Avoid hardcoding

It may be tempting to write a type-checking rule for method invocation, where your rule checks the name of the method being called and then treats the method in a special way. This is usually the wrong approach. It is better to write annotations, in a stub file (Chapter 18), and leave the work to the standard type-checking rules.

19.6 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 compiled, 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.

19.6.1 Bundling multiple checkers

To run a checker, a users supplies the `-processor` command-line option. When multiple related checkers need to be run together as a unit, users can pass multiple `-processor` arguments, like:

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

This is verbose, and it is also error-prone, since a user might omit one of several related checkers that are designed to be run together.

Alternatively, you can define an aggregate checker class that combines multiple checkers. Extend `AggregateChecker` and override the `getSupportedTypeCheckers` method, like the following:

```

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

```

Now, users can pass a single `-processor` argument on the command line:

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

19.7 Testing framework

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

19.8 Debugging options

The Checker Framework provides debugging options that can be helpful when writing a 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 19.5 above)
- `-Afilenames`: print 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.interningquals.Interned java.lang.String
  expected: DECLARED @checkers.interningquals.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.interningquals.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 execution main class to `com.sun.tools.javac.Main`, and insert the `JSR308 javac.jar` (resides in `$jsr308-langtools/dist/lib/javac.jar`). If using an IDE, it is recommended that you add `$jsr308-langtools` as a project, so you can step into its source code if needed.

19.9 javac implementation survival guide

A checker built using the Checker Framework makes use of a few interfaces from the underlying compiler. This section describes those interfaces.

19.9.1 Checker access to compiler information

The compiler uses and exposes three hierarchies to model the Java source code and classfiles.

Types - Java Language Model API

A `TypeMirror` represents a Java type.

There is a `TypeMirror` interface to represent each type kind, e.g., `PrimitiveType` for primitive types, `ExecutableType` for method types, and `NullType` for the type of the null literal.

`TypeMirror` does not represent annotated types though. Checkers should use the Framework types API `AnnotatedTypeMirror` instead. `AnnotatedTypeMirror` parallels the `TypeMirror` API, but also present the type annotations associated with the type.

The Checker Framework and the Checkers use the types API extensively.

Elements - Java Language Model API

An `Element` represents a potentially-public declaration that can be accessed from elsewhere: classes, interfaces, methods, constructors, and fields. `Element` represents elements found in both source code and bytecode.

There is an `Element` interface to represent each construct, e.g. `TypeElement` for class/interfaces, `ExecutableElement` for methods/constructors, `VariableElement` for local variables and method parameters.

If you need to operate on the declaration level, always use elements rather than trees (Section 19.9.1). This allows the code to work on both source and bytecode elements.

Example: retrieve declaration annotations, check variable modifiers (e.g. `strictfp`, `synchronized`)

Trees - Compiler Tree API

A `Tree` represents a syntactic units in the source code, like method declarations, statements, blocks, for loop etc. Trees only represent source code to be compiled (or found in `-sourcepath`); no tree is available for classes read from bytecode.

There is a `Tree` interface for each Java source structure, e.g. `ClassTree` for class declaration, `MethodInvocationTree` for method invocations, `ForEachTree` for enhanced-for-loop statement.

You should limit your use of trees. Checkers use Trees mainly to traverse the source code, retrieve the types/elements corresponding to them, and perform any needed checks on the types/elements instead.

Using the APIs

The three APIs use some common idioms and conventions; knowing them will help you to create your checker.

Type-checking: Do not use `instanceof` to determining the class of the object, because you cannot necessarily predict the run-time type of the object that implements an interface. Instead, use the `getKind()` method. The method returns `TypeKind`, `ElementKind`, and `Tree.Kind` for the three interfaces, respectively.

Visitors and Scanners: The compiler and the Checker Framework use the visitor pattern extensively. For example, visitors are used to traverse the source tree (`BaseTypeVisitor` extends `TreePathScanner`) and for type checking (`TreeAnnotator` implements `TreeVisitor`).

Utility classes: Some useful methods appear in a utility class. The Sun convention is that the utility class for a Foo hierarchy is `Foos` (e.g., `Types`, `Elements`, and `Trees`). The Checker Framework uses a common `Utils` post-fix instead (e.g. `TypesUtils`, `TreeUtils`, `ElementUtils`), with one notable exception: `AnnotatedTypes`.

19.9.2 How a checker fits in the compiler as an annotation processor

The Checker Framework builds on the Annotation Processing API introduced in Java 6. A type annotation processors is one that extends `AbstractTypeProcessor`; these get run on each class source file after the compiler confirms that the class is valid Java code.

The most important methods of `AbstractTypeProcessor` are `typeProcess` and `getSupportedSourceVersion`. The former class is where you would insert any sort of method call to walk the AST, and the latter just returns a constant indicating that we are targeting version 7 of the compiler. Implementing these two methods should be enough for a basic plugin; see the Javadoc for the class for other methods that you may find useful later on.

The Checker Framework uses Sun's Tree API to access a program's AST. The Tree API is specific to the Sun JDK, so the Checker Framework only works with Sun's `javac`, not with Eclipse's compiler `ecj` or with `gcj`. This also limits the tightness of the integration of the Checker Framework into other IDEs such as IntelliJ IDEA. An implementation-neutral API would be preferable. 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 source code of a method. But, at present no tools implement JSR 198. Also see Section 19.5.1.

Learning more about `javac`

Sun's `javac` compiler interfaces can be daunting to a newcomer, and its documentation is a bit sparse. The Checker Framework aims to abstract a lot of these complexities. You do not have to understand the implementation of `javac` to build powerful and useful checkers. Beyond this section, 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>).

Chapter 20

Integration with external tools

This section discusses how to run a checker from your favorite IDE.

Or, if your favorite isn't here, you should customize how it runs the `javac` command on your behalf. See the IDE documentation to learn how to customize it, adapting the instructions for `javac` in Section 2.2. If you make another tool support running a checker, please inform us via the mailing list or issue tracker so we can add it to this manual.

This section also discusses type inference tools (see Section 20.7).

20.1 Javac Compiler

If you use `javac` compiler from the command line, then you can use the Java 7 `javac` that is bundled with the Checker Framework. The bundled `javac` recognizes type annotations, and annotations in comments (see Section 17.3). (Eventually, you will be able to use any Java 7 compiler, but Oracle has been slow to incorporate all the patches, so the bundled `javac` is superior, for the purpose of pluggable type-checking.)

This section describes how you can install and use the bundled `javac`, using either Unix/Linux/MacOS (see Section 20.1.1) or Windows (see Section 20.1.2). The instructions are identical to those of Section 1.2, but are given as commands that you can cut and paste into your command shell.

20.1.1 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 by changing the definition of environment variable `JSR308` below; a standard place is in a new directory named `jsr308`.

```
export JSR308=$HOME/jsr308
mkdir -p ${JSR308}
cd ${JSR308}
# or: wget http://types.cs.washington.edu/checker-framework/current/checkers.zip
curl -O http://types.cs.washington.edu/checker-framework/current/checkers.zip
unzip checkers.zip
chmod +x checker-framework/checkers/binary/javac
checker-framework/checkers/binary/javac -version
```

The output of the last command should be:

```
javac 1.7.0-jsr308-1.1.1
```

2. Place the following commands in your `.bashrc` file:

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

Also execute them on the command line, or log out and back in. Then, verify that the installation works. From the command line, run:

```
javac -version
```

The output should be:

```
javac 1.7.0-jsr308-1.1.1
```

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

20.1.2 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://types.cs.washington.edu/checker-framework/current/checkers.zip> to your Desktop.
 - (b) Double-click the `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.
 - (c) Verify that the installation works. From a Windows command prompt, run (all on one line):

```
java -Xbootclasspath/p:C:\Path\To\...\checker-framework\checkers\binary\jsr308-all.jar
-jar C:\Program Files\checkers\binary\jsr308-all.jar -version
```

The output should be:

```
javac 1.7.0-jsr308-1.1.1
```

2. In order to use the updated compiler when you type `javac`, add the directory `C:\Program Files\checkers\binary` to the beginning of 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. You can set the variable as a “System Variable” (visible to all users) or as a “User Variable” (visible to just this user). Both work; the instructions below show how to set as a “System Variable”. 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.

Now, verify that the installation works. From the command line, run:

```
javac -version
```

The output should be:

```
javac 1.7.0-jsr308-1.1.1
```

20.2 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.

1. Set the `jsr308javac` property:

```
<property environment="env"/>

<presetdef name="jsr308.javac">
  <javac fork="yes">
    <!-- JSR308 related compiler arguments -->
    <compilerarg value="-version"/>
    <compilerarg line="-target 5"/>
    <compilerarg value="-implicit:class"/>
    <compilerarg line="-Awarns -Xmaxwarns 10000"/>
    <compilerarg value="-J-Xbootclasspath/p:${env.CHECKERS}/binary/jsr308-all.jar"/>

    <classpath>
      <pathelement location="${env.CHECKERS}/checkers.jar"/>
    </classpath>
  </javac>
</presetdef>
```

2. Duplicate the compilation target, then modify it slightly as indicated in this example, filling in each ellipsis (...

```
<target name="check-nullness"
  description="Check for nullness errors"
  depends="clean,...">
  <!-- use jsr308.javac instead of javac -->
  <jsr308.javac ... >
    <compilerarg line="-processor checkers.nullness.NullnessChecker"/>
    <compilerarg value="-Xbootclasspath/p:${env.CHECKERS}/jdk/jdk.jar"/>
    <!-- optional, for implicit imports: <compilerarg value="-J-Djsr308_imports=checkers.nullnessquals.*"/> -->
    <!-- optional, to not check library bodies: <compilerarg value="-AskipClasses:^(java\.awt\.|javax\.swing\.)/> -->
    ...
  </jsr308.javac>
</target>
```

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

The target should not include a `-source` argument, such as `-source 1.5`, because doing so will disable the annotations in comments feature (see Section 17.3, page 68).

20.2.1 Explanation

This section explains each part of the Ant task.

1. Definition of `jsr308.javac`:

The `fork` field of the `javac` task ensures 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 `-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.

The `-Awarns ...` compiler argument is optional, and causes the checker to treat errors as warnings so that you can see all errors in all files rather than only the errors in the first file; see Section 2.2.

2. The `check-nullness` target:

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

20.3 Maven plugin

If you use the Maven project tool, then you can specify the distributed checkers as part of your build process.

1. First, you need to add the repositories in your `pom.xml` file:

```
<repositories>
  <repository>
    <id>checker-framework-repo</id>
    <url>http://types.cs.washington.edu/m2-repo</url>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>checker-framework-repo</id>
    <url>http://types.cs.washington.edu/m2-repo</url>
  </pluginRepository>
</pluginRepositories>
```

2. Then, to use the annotations used by the distributed checkers, you'll have to declared as a dependency:

```
<dependencies>
  <!-- annotations for the standard checkers: nullness, internning, mutability -->
  <dependency>
    <groupId>com.jolira.types.checkers</groupId>
    <artifactId>checkers-quals</artifactId>
    <version>1.0.6</version>
  </dependency>

  <!-- other dependencies -->
</dependencies>
```

3. And finally, you need to attach the plugin to your build lifecycle:

```
<build>
  <plugins>
    <plugin>
      <groupId>com.jolira.types.checkers</groupId>
      <artifactId>checkersplugin</artifactId>
      <version>0.1</version>
      <executions>
        <execution>
          <!-- run the checkers after compilation; this can also be any later phase -->
          <phase>process-classes</phase>
          <goals>
            <goal>check</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

```

<configuration>
  <!-- required configuration options -->
  <!-- a list of processors to run -->
  <processors>
    <processor>checkers.nullness.NullnessChecker</processor>
    <processor>checkers.interning.InterningChecker</processor>
  </processors>

  <!-- other optional configuration -->
  <!-- full path to a java executable that should be used to create the forked JVM -->
  <executable>/opt/java1.6/bin/java</executable>
  <!-- should an error reported by a checker cause a build failure, or only be logged -->
  <failOnError>true|false</failOnError>
  <!-- a list of patterns to include, in the standard maven syntax; defaults to **/* -->
  <includes>
    <include>org/company/important/**/*.java</include>
  </includes>
  <!-- a list of patterns to exclude, in the standard maven syntax; defaults to an empty list -->
  <excludes>
    <exclude>org/company/notimportant/**/*.java</exclude>
  </excludes>
  <!-- additional parameters passed to the JSR308 java compiler -->
  <javacParams>-Xlint</javacParams>
  <!-- additional parameters to pass to the forked JVM -->
  <javaParams>-Xdebug</javaParams>
  <!-- versions of checkers to use; defaults to the current newest version: 1.0.6 -->
  <checkersVersion>0.8.8</checkersVersion>
</configuration>
</plugin>
</plugins>
</build>

```

The plugin was contributed by Adam Warski.

20.4 IntelliJ IDEA

IntelliJ IDEA (Maia release) supports the Type Annotations (JSR-308) syntax. See <http://blogs.jetbrains.com/idea/2009/07/type-annotations-jsr-308-support/>.

20.5 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 20.2. 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 for the Checker Framework The Checker Plugin is an Eclipse plugin that enables the use of the Checker Framework. Its website (<http://types.cs.washington.edu/checker-framework/eclipse/>). The website contains instructions for installing and using the plugin.

Eclipse plug-in for Type Annotations A prototype version of Type Annotations support for Eclipse is available from the Eclipse project. The goal is to enable full support for writing type annotations outside of comments. You do not need this to run the Checker Framework, whether or not you write your type annotations in comments.

(Update: this apparently needs a username and password, so it may not be publicly available.) Use the following information to check out the CVS repository:

Host: dev.eclipse.org

Repository path: /cvsroot/eclipse

Module name: org.eclipse.jdt.core

Branch: JSR_308

20.6 tIDE

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

20.7 Type inference tools

20.7.1 Varieties of type inference

There are two different tasks that are commonly called “type inference”.

1. Type inference during type checking (Section 16.3.2): During type checking, if certain variables have no type qualifier, the type-checker determines whether there is some type qualifier that would permit the program to type check. If so, the type checker uses that type qualifier, but never tells the programmer what it was. Each time the type checker runs, it re-infers the type qualifier for that variable. If no type qualifier exists that permits the program to type-check, the type-checker issues a type warning.

This variety of type inference is built into the Checker Framework. Every checker can take advantage of it at no extra effort. However, it only works within a method, not across method boundaries.

Advantages of this variety of type inference include:

- If the type qualifier is obvious to the programmer, then omitting it can reduce annotation clutter in the program.
 - The type inference can take advantage of only the code currently being compiled, rather than having to be correct for all possible calls. Additionally, if the code changes, then there is no old annotation to update.
2. Type inference to annotate a program (Section 20.7.2): As a separate step before type checking, a type inference tool takes the program as input, and outputs a set of type qualifiers that would type-check. These qualifiers are inserted into the source code or the class file. They can be viewed and adjusted by the programmer, and can be used by tools such as the type checker.

This variety of type inference must be provided by a separate tool. It is not built into the Checker Framework.

Advantages of this variety of type inference include:

- The program contains documentation in the form of type qualifiers, which can aid programmer understanding.

- Error messages may be more comprehensible. With type inference during type checking, error messages can be obscure, because the compiler has already inferred (possibly incorrect) types for a number of variables.
- A minor advantage is speed: type-checking can be modular, which can be faster than re-doing type inference every time the program is type-checked.

Advantages of both varieties of inference include:

- Less work for the programmer.
- The tool chooses the most general type, whereas a programmer might accidentally write a more specific, less generally-useful annotation.

Each variety of type inference has its place. When using the Checker Framework, type inference during type checking is performed only *within* a method (Section 16.3.2). Every method signature (arguments and return values) and field must be explicitly annotated, either by the programmer or by a separate type checking tool (Section 20.7.2). This choice reduces programmer effort (typically, a programmer does not have to write any qualifiers inside the body of a method) while still retaining modular checking and documentation benefits.

20.7.2 Type inference to annotate a program

This section lists tools that take a program and output a set of annotations for it.

Section 3.2.4 lists several tools that infer annotations for the Nullness Checker.

Section 6.2.2 lists a tool that infers annotations for the Javari Checker, which detects mutation errors.

Chapter 21

Frequently Asked Questions (FAQs)

These are some common questions about the Checker Framework and about pluggable type-checking in general. Feel free to suggest improvements to the answers, or other questions to include here.

There is a separate FAQ for the type annotations syntax (<http://types.cs.washington.edu/jsr308/jsr308-faq.html>).

Contents:

- 21.1: Are type annotations easy to read and write?
- 21.2: Will my code become cluttered with type annotations?
- 21.3: Can a pluggable type-checker give an absolute guarantee of correctness?
- 21.4: I don't make type errors, so would pluggable type checking help me?
- 21.5: Why shouldn't a qualifier apply to both types and declarations?
- 21.6: When should I use type qualifiers, and when should I use subclasses?
- 21.7: How do I get started annotating an existing program?
- 21.8: How do I run a checker on all my source files?
- 21.9: How do I shorten the command line when invoking a checker?
- 21.10: How do I create a new checker?
- 21.11: Why is there no declarative syntax for writing type rules?
- 21.12: Why not just use a bug detector (like FindBugs)?
- 21.13: How does pluggable type-checking compare with JML?
- 21.14: What is the meaning of array annotations such as `@NonNull Object @Nullable []`?
- 21.15: Why are the type parameters to `List` and `Map` annotated as `@NonNull`?
- 21.16: How can I do run-time monitoring of properties that were not statically checked?

21.1 Are type annotations easy to read and write?

The paper “Practical pluggable types for Java” [PAC⁺08] discusses case studies in which programmers found type annotations to be natural to read and write. The code continued to feel like Java, and the type-checking errors were easy to comprehend and often led to real bugs.

You don't have to take our word for it, though. You can try the Checker Framework for yourself.

The difficulty of adding and verifying annotations depends on your program. If your program is well-designed and -documented, then skimming the existing documentation and writing type annotations is extremely easy. Otherwise, you may find yourself spending a lot of time trying to understand, reverse-engineer, or fix bugs in your program, and then just a moment writing a type annotation that describes what you discovered. This process inevitably improves your code. You must decide whether it is a good use of your time. For code that is not causing trouble now and is unlikely to do so in the future (the code is bug-free, and you do not anticipate changing it or using it in new contexts), then the effort of writing type annotations for it may not be justified.

21.2 Will my code become cluttered with type annotations?

As with any language feature, it is possible to write ugly code that over-uses annotations. However, in normal use, very few annotations need to be written. Figure 1 of the paper Practical pluggable types for Java [PAC⁺08] reports data for over 350,000 lines of type-annotated code:

- 1 annotation per 62 lines for nullness annotations (@NonNull, @Nullable, etc.)
- 1 annotation per 1736 lines for interned annotations (@Interned)
- 1 annotation per 27 lines for immutability annotations (IGJ type system)

Furthermore, these numbers are for annotating existing code. New code that is written with the type annotation system in mind is cleaner and more correct, so it requires even fewer annotations.

In other words, annotations do not clutter code, and they are used much less frequently than generic types, which Java programmers find acceptable.

21.3 Can a pluggable type-checker give an absolute guarantee of correctness?

Each checker looks for certain errors. You can use multiple checkers, but even then your program might still contain other kinds of errors.

If you run a pluggable checker on only part of the code of a program, then you do not get a guarantee that all parts of the program satisfy the type system (that is, are error-free). An example is a framework that clients are intended to extend. In this case, you should recommend that clients run the pluggable checker. There is no way to force users to do so, so you may want to retain dynamic checks or use other mechanisms to detect errors.

There are other circumstances in which a static type-checker may fail to detect a possible type error. In Java, these include arrays, casts, raw types, reflection, separate compilation (bytecodes from unverified sources), native code, etc. (For details, see section 2.3.) Java uses dynamic checks for most of these, so that the type error cannot cause a security vulnerability or a crash. The pluggable type-checkers inherit many (not all) of these weaknesses of Java type-checking, but do not currently have built-in dynamic checkers. Writing dynamic checkers would be an interesting and valuable project.

Even if a tool such as a pluggable checker cannot give an ironclad guarantee of correctness, it is still useful. It can find errors, excluding certain types of possible problems (e.g., restricting the possible class of problems), and increasing confidence in a piece of software.

21.4 I don't make type errors, so would pluggable type checking help me?

Occasionally, a developer says that he makes no errors that typechecking could catch, or that any such errors are unimportant because they have low impact and are easy to fix. When I investigate the claim, I invariably find that the developer is mistaken.

Very frequently, the developer has underestimated what typechecking can discover. Not every type error leads to an exception being thrown; and even if an exception is thrown, it may not seem related to classical types. Remember that a type system can discover null pointer dereferences, incorrect side effects, security errors such as information leakage or SQL injection, partially-initialized data, and many other errors. Even where type-checking does not discover a problem directly, it can indicate code with bad smells, thus revealing problems, improving documentation, and making future maintenance easier.

There are other ways to discover errors, including extensive testing and debugging. But type-checking is a good complement to these. It is more effective for some problems, and less effective for other problems. It can reduce (but not eliminate) the time and effort that you spend on other approaches. There are many important errors that type checking and other automated approaches cannot find; pluggable typechecking gives you more time to focus on those.

21.5 Why shouldn't a qualifier apply to both types and declarations?

It is bad style for an annotation to apply to both types and declarations. In other words, every annotation should have a `@Target` meta-annotation, and the `@Target` meta-annotation should list either only declaration locations or only type annotations. (It's OK for an annotation to target both `ElementType.TYPE_PARAMETER` and `ElementType.TYPE_USE`, but no other declaration location along with `ElementType.TYPE_USE`.)

Sometimes, it may seem tempting for an annotation to apply to both type uses and (say) method declarations. Here is a hypothetical example:

"Each `Widget` type may have a `@Version` annotation. I wish to prove that versions of widgets don't get assigned to incompatible variables, and that older code does not call newer code (to avoid problems when backporting).

A `@Version` annotation could be written like so:

```
@Version("2.0") Widget createWidget(String value) { ... }
```

`@Version("2.0")` on the method could mean that the `createWidget` method only appears in the 2.0 version. `@Version("2.0")` on the return type could mean that the returned `Widget` should only be used by code that uses the 2.0 API of `Widget`. It should be possible to specify these independently, such as a 2.0 method that returns a value that allows the 1.0 API method invocations."

Both of these are type properties and should be specified with type annotations. No method annotation is necessary or desirable. The best way to require that the receiver has a certain property is to use a type annotation on the receiver of the method. (Slightly more formally, the property being checked is compatibility between the annotation on the type of the formal parameter receiver and the annotation on the type of the actual receiver.)

Another example of a type-and-declaration annotation that represents poor design is JCIP's `@GuardedBy` annotation [GPB⁺06]. As discussed in Section 7.1.3, it means two different things when applied to a field or a method. To reduce confusion and increase expressiveness, the Lock Checker (see Chapter 7) uses the `@Holding` annotation for one of these meanings, rather than overloading `@GuardedBy` with two distinct meanings.

21.6 When should I use type qualifiers, and when should I use subclasses?

In brief, use subtypes when you can, and use type qualifiers when you cannot use subtypes. For more details, see section 2.4.6.

21.7 How do I get started annotating an existing program?

See Section 2.4.1.

21.8 How do I run a checker on all my source files?

The `javac` compiler halts compilation as soon as it processes a source file with an error, including an error issued by a pluggable type-checker. Section 2.2 describes the `-Awarns` command-line option that turns checker errors into warnings, permitting `javac` to continue past the first erroneous source file.

21.9 How do I shorten the command line when invoking a checker?

The compile options to `javac` can be a pain to type; for example, `javac -processor checkers.nullness.NullnessChecker ...`. See Section 2.2.2 for a way to avoid the need for the `-processor` command-line option.

21.10 How do I create a new checker?

In addition to using the checkers that are distributed with the Checker Framework, you can write your own checker to check specific properties that you care about. Thus, you can find and prevent the bugs that are most important to you.

Chapter 19 gives complete details regarding how to write a checker. It also suggests places to look for more help, such as the Checker Framework API documentation (Javadoc) and the source code of the distributed checkers.

To whet your interest and demonstrate how easy it is to get started, here is an example of a complete, useful type checker.

```
@TypeQualifier
@SubtypeOf(Unqualified.class)
@Target({ElementType.TYPE_PARAMETER, ElementType.TYPE_USE})
public @interface Encrypted { }
```

Section 13.2 explains this checker and tells you how to run it.

21.11 Why is there no declarative syntax for writing type rules?

A type system implementer can declaratively specify the type qualifier hierarchy (Section 19.3.1) and the type introduction rules (Section 19.4.1). However, the Checker Framework uses a procedural syntax for specifying type-checking rules (Section 19.5). A declarative syntax might be more concise, more readable, and more verifiable than a procedural syntax.

We have not found the procedural syntax to be the most important impediment to writing a checker.

Previous attempts to devise a declarative syntax for realistic type systems have failed; see a technical paper [PAC⁺08] for a discussion. When an adequate syntax exists, then the Checker Framework can be extended to support it.

21.12 Why not just use a bug detector (like FindBugs)?

Pluggable type-checking finds more bugs than a bug detector does, for any given variety of bug.

A bug detector like FindBugs [HP04, HSP05], JLint [Art01], or PMD [Cop05] aims to find *some* of the most obvious bugs in your program. It uses a lightweight analysis, then uses heuristics to discard some of its warnings. Thus, even if the tool prints no warnings, your code might still have errors — maybe the analysis was too weak to find them, or the tool’s heuristics classified the warnings as likely false positives and discarded them.

A type checker aims to find *all* the bugs (of certain varieties). It requires you to write type qualifiers in your program, or to use a tool that infers types. Thus, it requires more work from the programmer, and in return it gives stronger guarantees.

Each tool is useful in different circumstances, depending on how important your code is and your desired level of confidence in your code. For more details on the comparison, see section 22.5. For a case study that compared the nullness analysis of FindBugs, JLint, PMD, and the Checker Framework, see section 6 of the paper “Practical pluggable types for Java” [PAC⁺08].

21.13 How does pluggable type-checking compare with JML?

JML, the Java Modeling Language [LBR06], is a language for writing formal specifications. JML aims to be more expressive than pluggable type-checking. JML is not as practical as pluggable type-checking.

A programmer can write a JML specification that describes arbitrary facts about program behavior. Then, the programmer can use formal reasoning or a theorem-proving tool to verify that the code meets the specification. Run-time checking is also possible. By contrast, pluggable type-checking can express a more limited set of properties about your program.

The JML toolset is less mature. For instance, if your code uses generics or other features of Java 5, then you cannot use JML. However, JML has a run-time checker, which the Checker Framework currently lacks.

21.14 What is the meaning of array annotations such as `@NonNull Object @Nullable []`?

`@NonNull Object @Nullable []` is a possibly-null array of non-null objects. Note that even though the first token in the type is “`@NonNull`”, that annotation applies to the element type `Object`. The annotation `@Nullable` applies to the array (`[]`).

Similarly, `@Nullable Object @NonNull []` is a non-null array of possibly-null objects.

21.15 Why are the type parameters to `List` and `Map` annotated as `@NonNull`?

The annotation on `java.util.Collection` only allows non-null elements:

```
public interface Collection<E extends @NonNull Object> {  
    ...  
}
```

Thus, you will get a type error if you write code like `Collection<@Nullable Object>`. A nullable type parameter is also forbidden for certain other collections, including `AbstractCollection`, `List`, `Map`, and `Queue`.

The `extends @NonNull Object` bound is a direct consequence of the design of the collections classes; it merely formalizes the Javadoc specification. The Javadoc for `Collection` states:

Some list implementations have restrictions on the elements that they may contain. For example, some implementations prohibit null elements, ...

Here are some consequences of the requirement to detect all nullness errors at compile time. If even one subclass of a given collection class may prohibit null, then the collection class and all its subclasses must prohibit null. Conversely, if a collection class is specified to accept null, then all its subclasses must honor that specification.

The Checker Framework’s annotations make apparent a flaw in the JDK design, and helps you to avoid problems that might be caused by that flaw.

Justification from type theory Suppose `B` is a subtype of `A`. Then an overriding method in `B` must have a stronger (or equal) signature than the overridden method in `A`. In a stronger signature, the formal parameter types may be supertypes, and the return type may be a subtype. Here are examples:

```
class A { @NonNull Object Number m1( @NonNull Object arg) { ... } }  
class B extends A { @Nullable Object Number m1( @NonNull Object arg) { ... } } // error!  
class C extends A { @NonNull Object Number m1( @Nullable Object arg) { ... } } // OK  
class D { @Nullable Object Number m2( @Nullable Object arg) { ... } }  
class E extends D { @NonNull Object Number m2( @Nullable Object arg) { ... } } // OK  
class F extends D { @Nullable Object Number m2( @NonNull Object arg) { ... } } // error!
```

According to these rules, since some subclasses of `Collection` do not permit nulls, then `Collection` cannot either:

```
// does not permit null elements  
class PriorityQueue<E> implements Collection<E> {  
    boolean add(E);  
    ...  
}  
// must not permit null elements, or PriorityQueue would not be a subtype of Collection  
interface Collection<E> {  
    boolean add(E);  
    ...  
}
```

Justification from checker behavior Suppose that you changed the bound in the `Collection` declaration to `extends @Nullable Object`. Then, the checker would issue no warning for this method:

```
static void addNull(Collection l) {
    l.add(null);
}
```

However, calling this method *can* result in a null pointer exception, for instance caused by the following code:

```
addNull(new PriorityQueue());
```

Therefore, the bound must remain as `extends @NonNull Object`.

By contrast, this code is OK because `ArrayList` is documented to support null elements:

```
static void addNull(ArrayList l) {
    l.add(null);
}
```

Therefore, the upper bound in `ArrayList` is `extends @Nullable Object`. Any subclass of `ArrayList` must also support null elements.

Suppressing warnings Suppose your program has a list variable, and you know that any list referenced by that variable will definitely support null. Then, you can suppress the warning:

```
@SuppressWarnings("nullness:generic.argument")
static void addNull(List l) {
    l.add(null);
}
```

You need to use `@SuppressWarnings("nullness:generic.argument")` whenever you use a collection that may contain null elements in contradiction to its documentation. Fortunately, such uses are relatively rare.

For more details on suppressing nullness warnings, see Section 3.4.

21.16 How can I do run-time monitoring of properties that were not statically checked?

Currently, the Checker Framework has no support for adding code to check, at run time, code that was not checked (see Chapter 17 for reasons that code might not be checked). An exception is the Nullness Checker, which has ways to dynamically check nullness via assertions and casts (the `NullnessUtils.castNonNull` method); see Section 3.4.1.

More general support would be an interesting and valuable project. If you are able to add run-time verification functionality, we would gladly welcome it as a contribution to the Checker Framework.

Chapter 22

Troubleshooting and getting help

Please read the entire manual, including this chapter and the FAQ (Chapter 21), because the manual might already answer your question. If not, you can use the mailing list, checker-framework-discuss@googlegroups.com, to ask other users for help. For archives and to subscribe, see <http://groups.google.com/group/checker-framework-discuss>. To report bugs, use the issue tracker at <http://code.google.com/p/checker-framework/issues/list>. If you want to help out, you can choose a bug and fix it, or select a project from the ideas list at <http://code.google.com/p/checker-framework/wiki/Ideas>.

22.1 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 17.3.2). You can solve this by unsetting the variable/option, or by ensuring that the `checkers.jar` file is on your classpath. If the error is

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

(note the extra apostrophe!), then you have probably mis-used quoting when supplying the `jsr308_imports` environment variable.

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

- 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 Type Annotations 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 from the JDK or another external library, then maybe the stub/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)
```

- If you get an error like the following when using the Ant task (Section 20.2),

```
...\build.xml:59: Error running ${env.CHECKERS}\binary\javac.bat compiler
```

then the problem may be that you have not set the CHECKERS environment variable, as described in Section 20.1.2. Or, maybe you made it a user variable instead of a system variable.

- If you get one of these errors:

```
The hierarchy of the type ClassName is inconsistent
```

```
The type com.sun.source.util.AbstractTypeProcessor cannot be resolved.
```

```
It is indirectly referenced from required .class files",
then you are missing jsr308-all.jar from your classpath.
```

- If you get the error

```
java.lang.ArrayStoreException: sun.reflect.annotation.TypeNotPresentExceptionProxy
```

then an annotation is not present at run time that was present at compile time. For example, maybe when you compiled the code, the `@NonNull` annotation was available, but it was not available at run time. You can use JDK 7 at run time, or compile with a Java 6 compiler that will ignore the annotations in comments.

22.1.1 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.

22.1.2 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.

22.2 How to report problems

If you have a problem with any checker, or with the Checker Framework, please file a bug at <http://code.google.com/p/checker-framework/issues/list>. (First, check whether there is an existing bug report for that issue.)

Alternately (especially if your communication is not a bug report), you can send mail to checker-framework-dev@googlegroups.com. 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 `-version -verbose` to the `javac` options. This causes the compiler to output debugging information, including its version number.
- Indicate exactly what you did. Don't skip any steps, and don't merely describe your actions in words. Show the exact commands by attaching a file or using cut-and-paste from your command shell;
- 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 by attaching a file or using cut-and-paste from your command shell (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.

A particularly useful format for a test case is as a diff, or a new file, for the existing checker test cases. For instance, for the Nullness Checker, see directory `checker-framework/checkers/tests/nullness/`.

22.3 Building from source

The Checker Framework 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 re-build its binaries from source. You will be using the latest development version of the Checker Framework, rather than an official release.

22.3.1 Obtain the source

Obtain the latest source code from the version control repository:

```
export JSR308=$HOME/jsr308
mkdir -p $JSR308
cd $JSR308
hg clone https://jsr308-langtools.googlecode.com/hg/ jsr308-langtools
hg clone https://checker-framework.googlecode.com/hg/ checker-framework
```

(Alternately, you could use the version of the source code that is packaged in the Checker Framework release.)

22.3.2 Build the Type Annotations compiler

1. Set the `JAVA_HOME` environment variable to the location of your JDK 6 or 7 installation (not the JRE installation. (It may already be set for Ant to work.)
2. Compile the Type Annotations `javac` compiler and the `javap` tool:

```
cd $JSR308/jsr308-langtools/make
ant clean build-javac build-javap
```

3. Add the `jsr308-langtools/dist/bin` directory to the front of your `PATH` environment variable. Example command:

```
export PATH=$JSR308/jsr308-langtools/dist/bin:${PATH}
```

22.3.3 Build the Checker Framework

1. Run ant to create checkers.jar:

```
cd $JSR308/checker-framework/checkers
ant
```

2. Add tools.jar and checkers.jar to your classpath (If you do not do this, you will have to supply the -cp option whenever you run javac and use a checker plugin.) Example command:

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

3. Test that everything works:

- Run ant all-tests in the checkers directory:

```
cd $JSR308/checker-framework/checkers
ant all-tests
```
- Run the Nullness checker examples (see Section 3.7, page 26).

22.3.4 Build the Checker Framework manual (this document)

1. To build the manual you will need plume-bib (<http://code.google.com/p/plume-bib/>) and HEVEA (<http://hevea.inria.fr/>) installed.
2. Run make in the checkers/manual directory to build both the PDF and HTML versions of the manual.

22.4 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 case studies in which each of the checkers found previously-unknown errors in real software.

22.5 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 2.3.)
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 17.2.
- 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 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.

Also see the JSR 308 [Ern08] documentation for a detailed discussion of related work.

22.6 Credits and changelog

The key developers of the Checker Framework are 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 Checker Framework distribution and is also available on the web at <http://types.cs.washington.edu/checker-framework/current/changelog-checkers.txt>.

Bibliography

- [Art01] Cyrille Artho. Finding faults in multi-threaded programs. Master's thesis, Swiss Federal Institute of Technology, March 15, 2001.
- [Cop05] Tom Copeland. *PMD Applied*. Centennial Books, November 2005.
- [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://types.cs.washington.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.
- [HP04] David Hovemeyer and William Pugh. Finding bugs is easy. In *Companion to Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 132–136, Vancouver, BC, Canada, October 26–28, 2004.
- [HSP05] David Hovemeyer, Jaime Spacco, and William Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2005)*, pages 13–19, Lisbon, Portugal, September 5–6, 2005.
- [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 Kiezun, 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.