The Checker Framework Manual: Custom pluggable types for Java

http://checkerframework.org/

Version 1.9.4 (4 Aug 2015)

For the impatient: Section 1.3 (page 13) describes how to **install and use** pluggable type-checkers.

Contents

1 Introduction **12**
- 1.1 How to read this manual . . . 13
- 1.2 How it works: Pluggable types . . . 13
- 1.3 Installation . . . 13
- 1.4 Example use: detecting a null pointer bug . . . 14
- 1.5 What comes with the Checker Framework distribution . . . 14

2 Using a checker **16**
- 2.1 Writing annotations . . . 16
- 2.2 Running a checker . . . 17
  - 2.2.1 Distributing your annotated project . . . 17
  - 2.2.2 Summary of command-line options . . . 18
  - 2.2.3 Checker auto-discovery . . . 19
  - 2.2.4 Shorthand for built-in checkers . . . 19
- 2.3 What the checker guarantees . . . 20
- 2.4 Tips about writing annotations . . . 20
  - 2.4.1 How to get started annotating legacy code . . . 20
  - 2.4.2 Do not annotate local variables unless necessary . . . 21
  - 2.4.3 Annotations indicate normal behavior . . . 21
  - 2.4.4 Subclasses must respect superclass annotations . . . 22
  - 2.4.5 Annotations on constructor invocations . . . 22
  - 2.4.6 What to do if a checker issues a warning about your code . . . 23

3 Nullness Checker **24**
- 3.1 What the Nullness Checker checks . . . 24
- 3.2 Nullness annotations . . . 25
  - 3.2.1 Nullness qualifiers . . . 25
  - 3.2.2 Nullness method annotations . . . 26
  - 3.2.3 Initialization qualifiers . . . 26
  - 3.2.4 Map key qualifiers . . . 26
- 3.3 Writing nullness annotations . . . 27
  - 3.3.1 Implicit qualifiers . . . 27
  - 3.3.2 Default annotation . . . 27
  - 3.3.3 Conditional nullness . . . 27
  - 3.3.4 Nullness and arrays . . . 28
  - 3.3.5 Run-time checks for nullness . . . 28
  - 3.3.6 Additional details . . . 28
  - 3.3.7 Inference of `@NonNull` and `@Nullable` annotations . . . 28
- 3.4 Suppressing nullness warnings . . . 29
  - 3.4.1 Suppressing warnings with assertions and method calls . . . 29

3.5 Examples . . . 30
3.5.1 Tiny examples . . . 30
3.5.2 Example annotated source code . . . 30
3.6 Tips for getting started . . . 30
3.7 Other tools for nullness checking . . . 31
3.7.1 Which tool is right for you? . . . 32
3.7.2 Incompatibility note about FindBugs `@Nullable` . . . 32
3.7.3 Relationship to `Optional<T>` . . . 33
3.8 Initialization Checker . . . 33
3.8.1 Initialization qualifiers . . . 35
3.8.2 How an object becomes initialized . . . 36
3.8.3 Partial initialization . . . 36
3.8.4 How to handle warnings . . . 37
3.8.5 More details about initialization checking . . . 39
3.8.6 Rawness Initialization Checker . . . 39

4 Map Key Checker **44**
4.1 Map key annotations . . . 44
4.2 Examples . . . 45
4.3 Inference of @KeyFor annotations . . . 45

5 Interning Checker **47**
5.1 Interning annotations . . . 48
5.2 Annotating your code with `@Interned` . . . 48
5.2.1 Implicit qualifiers . . . 48
5.3 What the Interning Checker checks . . . 48
5.3.1 Limitations of the Interning Checker . . . 49
5.4 Examples . . . 49
5.5 Other interning annotations . . . 49

6 Lock Checker **50**
6.1 Lock annotations . . . 50
6.1.1 Type annotations for objects protected by locks . . . 50
6.1.2 Lock method annotations . . . 50
6.1.3 Discussion of `@Holding` . . . 51
6.2 Examples . . . 52
6.2.1 Examples of @GuardedBy and @Holding . . . 52
6.2.2 Examples of @EnsuresLockHeld and @EnsuresLockHeldIf . . . 52
6.2.3 Example of @LockingFree . . . 53
6.3 Other lock annotations . . . 54
6.3.1 Relationship to annotations in *Java Concurrency in Practice* . . . 54
6.4 Possible extensions . . . 55
6.5 A note on Lock Checker internals . . . 55

7 Fake Enum Checker **56**
7.1 Fake enum annotations . . . 56
7.2 What the Fenum Checker checks . . . 57
7.3 Running the Fenum Checker . . . 57
7.4 Suppressing warnings . . . 57
7.5 Example . . . 58
7.6 References . . . 58

8 Tainting Checker **59**
8.1 Tainting annotations . . . 59
8.2 Tips on writing `@Untainted` annotations . . . 59
8.3 `@Tainted` and `@Untainted` can be used for many purposes . . . 60
8.3.1 Qualifier Parameters . . . 60

9 Regex Checker for regular expression syntax **61**
9.1 Regex annotations . . . 61
9.2 Annotating your code with `@Regex` . . . 61
9.2.1 Implicit qualifiers . . . 61
9.2.2 Capturing groups . . . 62
9.2.3 Concatenation of partial regular expressions . . . 62
9.2.4 Testing whether a string is a regular expression . . . 63
9.2.5 Qualifier Parameters . . . 63
9.2.6 Suppressing warnings . . . 63

10 Format String Checker **65**
10.1 Formatting terminology . . . 65
10.2 Format String Checker annotations . . . 65
10.2.1 Conversion Categories . . . 66
10.3 What the Format String Checker checks . . . 68
10.3.1 Possible false alarms . . . 68
10.3.2 Possible missed alarms . . . 69
10.4 Implicit qualifiers . . . 69
10.5 FormatMethod . . . 70
10.6 Testing whether a format string is valid . . . 70

11 Internationalization Format String Checker (I18n Format String Checker) **71**
11.1 Internationalization Format String Checker annotations . . . 71
11.2 Conversion categories . . . 72
11.3 What the Internationalization Format String Checker checks . . . 73
11.4 Resource files . . . 74
11.5 Running the Internationalization Format Checker . . . 75
11.6 Testing whether a string has an i18n format type . . . 75
11.7 Examples of using the Internationalization Format Checker . . . 75

12 Property File Checker **77**
12.1 General Property File Checker . . . 77
12.2 Internationalization Checker . . . 78
12.2.1 Internationalization annotations . . . 78
12.2.2 Running the Internationalization Checker . . . 78
12.3 Compiler Message Key Checker . . . 78

13 Signature Checker for string representations of types **80**
13.1 Signature annotations . . . 80
13.2 What the Signature Checker checks . . . 81

14 GUI Effect Checker **82**
14.1 GUI effect annotations . . . 83
14.2 What the GUI Effect Checker checks . . . 83
14.3 Running the GUI Effect Checker . . . 83
14.4 Annotation defaults . . . 83

14.5 Polymorphic effects . . . 84
14.5.1 Defining an effect-polymorphic type . . . 84
14.5.2 Using an effect-polymorphic type . . . 84
14.5.3 Subclassing a specific instantiation of an effect-polymorphic type . . . 84
14.5.4 Subtyping with polymorphic effects . . . 85
14.6 References . . . 85

15 Units Checker **86**
15.1 Units annotations . . . 86
15.2 Extending the Units Checker . . . 87
15.3 What the Units Checker checks . . . 87
15.4 Running the Units Checker . . . 88
15.5 Suppressing warnings . . . 88
15.6 References . . . 88

16 Constant Value Checker **89**
16.1 Annotations . . . 89
16.1.1 Type Annotations . . . 89
16.1.2 Compile-time execution of expressions . . . 89
16.2 Warnings . . . 91

17 Aliasing Checker **92**
17.1 Aliasing annotations . . . 92
17.2 Leaking contexts . . . 93
17.3 Restrictions on where `@Unique` may be written . . . 94
17.4 Aliasing type refinement . . . 94

18 Linear Checker for preventing aliasing **96**
18.1 Linear annotations . . . 96
18.2 Limitations . . . 97

19 IGJ immutability checker **98**
19.1 IGJ and mutability . . . 98
19.2 IGJ Annotations . . . 99
19.3 What the IGJ Checker checks . . . 99
19.4 Implicit and default qualifiers . . . 99
19.5 Annotation IGJ dialect . . . 100
19.5.1 Semantic Changes . . . 100
19.5.2 Syntax Changes . . . 100
19.5.3 Templating over immutability: `@I` . . . 100
19.6 Iterators and their abstract state . . . 101
19.7 Examples . . . 101

20 Javari immutability checker **102**
20.1 Javari annotations . . . 102
20.2 Writing Javari annotations . . . 103
20.2.1 Implicit qualifiers . . . 103
20.2.2 Inference of Javari annotations . . . 103
20.3 What the Javari Checker checks . . . 103
20.4 Iterators and their abstract state . . . 103
20.5 Examples . . . 103

21 Reflection resolution **104**
21.1 MethodVal and ClassVal Checkers . . . 104
21.1.1 ClassVal Checker . . . 104
21.1.2 MethodVal Checker . . . 105
21.1.3 MethodVal and ClassVal inference . . . 106
21.2 Reflection resolution example . . . 107

22 Subtyping Checker **108**
22.1 Using the Subtyping Checker . . . 108
22.2 Subtyping Checker example . . . 109

23 Third-party checkers **111**
23.1 Typestate checkers . . . 111
23.1.1 Comparison to flow-sensitive type refinement . . . 111
23.2 Units and dimensions checker . . . 112
23.3 Thread locality checker . . . 112
23.4 Safety-Critical Java checker . . . 112
23.5 Generic Universe Types checker . . . 112
23.6 EnerJ checker . . . 112
23.7 CheckLT taint checker . . . 112
23.8 SPARTA information flow type-checker for Android . . . 112

24 Generics and polymorphism **113**
24.1 Generics (parametric polymorphism or type polymorphism) . . . 113
24.1.1 Raw types . . . 113
24.1.2 Restricting instantiation of a generic class . . . 113
24.1.3 Type annotations on a use of a generic type variable . . . 115
24.1.4 Annotations on wildcards . . . 115
24.1.5 Examples of qualifiers on a type parameter . . . 116
24.1.6 Covariant type parameters . . . 116
24.1.7 Method type argument inference and type qualifiers . . . 117
24.2 Qualifier polymorphism . . . 117
24.2.1 Examples of using polymorphic qualifiers . . . 117
24.2.2 Relationship to subtyping and generics . . . 118
24.2.3 Using multiple polymorphic qualifiers in a method signature . . . 118
24.2.4 Using a single polymorphic qualifier on an element type . . . 119
24.2.5 The `@PolyAll` qualifier applies to every type system . . . 119
24.3 Qualifier parameters . . . 120
24.3.1 Motivation for qualifier parameters . . . 121
24.3.2 Overview of qualifier parameters . . . 121
24.3.3 Qualifier parameter wildcards . . . 122
24.3.4 Syntax of qualifier parameters . . . 122
24.3.5 Primary qualifiers . . . 126

25 Advanced type system features **127**
25.1 Invariant array types . . . 127
25.2 Context-sensitive type inference for array constructors . . . 127
25.3 The effective qualifier on a type (defaults and inference) . . . 128
25.3.1 Default qualifier for unannotated types . . . 129
25.3.2 Defaulting rules and CLIMB-to-top . . . 130
25.3.3 Inherited defaults . . . 131
25.3.4 Inherited wildcard annotations . . . 131

25.3.5 Default qualifiers for `.class` files (conservative library defaults) . . . 132
25.4 Automatic type refinement (flow-sensitive type qualifier inference) . . . 132
25.4.1 Run-time tests and type refinement . . . 134
25.4.2 Fields and flow-sensitive analysis . . . 135
25.4.3 Side effects, determinism, purity, and flow-sensitive analysis . . . 135
25.4.4 Assertions . . . 137
25.5 Writing Java expressions as annotation arguments . . . 138
25.6 Unused fields . . . 138
25.6.1 `@Unused` annotation . . . 139

26 Suppressing warnings **140**
26.1 `@SuppressWarnings` annotation . . . 140
26.1.1 `@SuppressWarnings` syntax . . . 141
26.1.2 Where `@SuppressWarnings` can be written . . . 141
26.1.3 Good practices when suppressing warnings . . . 141
26.2 `@AssumeAssertion` string in an `assert` message . . . 142
26.2.1 Suppressing warnings and defensive programming . . . 142
26.3 `-AsuppressWarnings` command-line option . . . 143
26.4 `-AskipUses` and `-AonlyUses` command-line options . . . 143
26.5 `-AskipDefs` and `-AonlyDefs` command-line options . . . 144
26.6 `-Alint` command-line option . . . 144
26.7 No `-processor` command-line option . . . 144
26.8 Checker-specific mechanisms . . . 145

27 Handling legacy code **146**
27.1 Checking partially-annotated programs: handling unannotated code . . . 146
27.2 Backward compatibility with earlier versions of Java . . . 146
27.2.1 Annotations in comments . . . 147
27.2.2 Import statements and receiver parameters in comments . . . 147
27.2.3 Migrating away from annotations in comments . . . 148
27.2.4 No modular type-checking when targeting Java 5/6/7 . . . 148
27.2.5 Distributing declaration annotations instead of type annotations . . . 149

28 Annotating libraries **151**
28.1 Compiling partially-annotated libraries . . . 152
28.1.1 The `-AuseSafeDefaultsForUnannotatedSourceCode` command-line argument . . . 152
28.1.2 Workflow for creating or augmenting a partially-annotated library . . . 152
28.2 Using stub classes . . . 153
28.2.1 Using a stub file . . . 153
28.2.2 Stub file format . . . 153
28.2.3 Creating a stub file . . . 154
28.2.4 Troubleshooting stub libraries . . . 155
28.2.5 Limitations . . . 155
28.3 Troubleshooting/debugging annotated libraries . . . 156

29 How to create a new checker **157**
29.1 Relationship of the Checker Framework to other tools . . . 157
29.2 The parts of a checker . . . 158
29.3 Annotations: Type qualifiers and hierarchy . . . 158
29.3.1 Defining the type qualifiers . . . 158
29.3.2 Declaratively defining the qualifier hierarchy . . . 159
29.3.3 Procedurally defining the qualifier hierarchy . . . 159

29.3.4 Defining a default annotation . . . 160
29.3.5 Completeness of the type hierarchy . . . 160
29.4 Type factory: Implicit annotations . . . 161
29.4.1 Declaratively specifying implicit annotations . . . 161
29.4.2 Procedurally specifying implicit annotations . . . 162
29.5 Dataflow: enhancing flow-sensitive type qualifier inference . . . 162
29.5.1 Create required classes and configure their use . . . 163
29.5.2 Override methods that handle Nodes of interest . . . 163
29.5.3 Determine the expressions to refine the types of . . . 164
29.5.4 Implement the refinement . . . 164
29.6 Visitor: Type rules . . . 165
29.6.1 AST traversal . . . 166
29.6.2 Avoid hardcoding . . . 166
29.7 The checker class: Compiler interface . . . 166
29.7.1 Bundling multiple checkers . . . 167
29.7.2 Providing command-line options . . . 168
29.8 Testing framework . . . 168
29.9 Debugging options . . . 169
29.9.1 Amount of detail in messages . . . 169
29.9.2 Stub and JDK libraries . . . 169
29.9.3 Progress tracing . . . 169
29.9.4 Saving the command-line arguments to a file . . . 169
29.9.5 Miscellaneous debugging options . . . 169
29.9.6 Examples . . . 169
29.10 Documenting the checker . . . 170
29.11 javac implementation survival guide . . . 171
29.11.1 Checker access to compiler information . . . 171
29.11.2 How a checker fits in the compiler as an annotation processor . . . 172
29.12 Integrating a checker with the Checker Framework . . . 172

30 Integration with external tools **173**
30.1 Javac compiler . . . 173
30.2 Ant task . . . 174
30.2.1 Explanation . . . 175
30.3 Maven . . . 175
30.3.1 Debugging the Maven compiler command-line arguments . . . 176
30.4 Gradle . . . 176
30.5 IntelliJ IDEA . . . 177
30.6 Eclipse . . . 177
30.6.1 Using an Ant task . . . 177
30.6.2 Eclipse plugin for the Checker Framework . . . 178
30.7 tIDE . . . 178
30.8 Type inference tools . . . 178
30.8.1 Varieties of type inference . . . 178
30.8.2 Type inference to annotate a program . . . 179

31 Frequently Asked Questions (FAQs) **180**
31.1 Motivation for pluggable type-checking . . . 181
31.1.1 I don't make type errors, so would pluggable type-checking help me? . . . 181
31.1.2 When should I use type qualifiers, and when should I use subclasses? . . . 181
31.2 Getting started . . . 181
31.2.1 How do I get started annotating an existing program? . . . 181

31.2.2 Which checker should I start with? . . . 182
31.2.3 Should I use pluggable types or Java subtypes? . . . 182
31.3 Usability of pluggable type-checking . . . 183
31.3.1 Are type annotations easy to read and write? . . . 183
31.3.2 Will my code become cluttered with type annotations? . . . 183
31.3.3 Will using the Checker Framework slow down my program? Will it slow down the compiler? 184
31.3.4 How do I shorten the command line when invoking a checker? . . . 184
31.4 How to handle warnings and errors . . . 184
31.4.1 What should I do if a checker issues a warning about my code? . . . 184
31.4.2 What does a certain Checker Framework warning message mean? . . . 184
31.4.3 Can a pluggable type-checker guarantee that my code is correct? . . . 184
31.4.4 What guarantee does the Checker Framework give for concurrent code? . . . 185
31.4.5 How do I make compilation succeed even if a checker issues errors? . . . 185
31.4.6 Why does the checker always say there are 100 errors or warnings? . . . 185
31.4.7 Why does the Checker Framework report an error regarding a type I have not written in my program? . . . 185
31.4.8 How can I do run-time monitoring of properties that were not statically checked? . . . 185
31.5 Syntax of type annotations . . . 186
31.5.1 What is a "receiver"? . . . 186
31.5.2 What is the meaning of an annotation after a type, such as `@NonNull Object @Nullable`? . 186
31.5.3 What is the meaning of array annotations such as `@NonNull Object @Nullable []`? . . . 186
31.5.4 What is the meaning of a type qualifier at a class declaration? . . . 187
31.5.5 Why shouldn't a qualifier apply to both types and declarations? . . . 187
31.6 Semantics of type annotations . . . 187
31.6.1 Why are the type parameters to `List` and `Map` annotated as `@NonNull`? . . . 187
31.6.2 How can I handle typestate, or phases of my program with different data properties? . . . 189
31.6.3 Why are explicit and implicit bounds defaulted differently? . . . 189
31.7 Creating a new checker . . . 190
31.7.1 How do I create a new checker? . . . 190
31.7.2 Why is there no declarative syntax for writing type rules? . . . 190
31.8 Relationship to other tools . . . 190
31.8.1 Why not just use a bug detector (like FindBugs)? . . . 190
31.8.2 How does the Checker Framework compare with Eclipse's null analysis? . . . 191
31.8.3 How does pluggable type-checking compare with JML? . . . 191
31.8.4 Is the Checker Framework an official part of Java? . . . 191
31.8.5 What is the relationship between the Checker Framework and JSR 305? . . . 192
31.8.6 What is the relationship between the Checker Framework and JSR 308? . . . 192

32 Troubleshooting and getting help **193**
32.1 Common problems and solutions . . . 193
32.1.1 Unable to run the checker, or checker crashes . . . 193
32.1.2 Unexpected type-checking results . . . 195
32.1.3 Unable to build the checker, or to run programs . . . 197
32.1.4 Classfile version warning . . . 198
32.2 How to report problems (bug reporting) . . . 198
32.3 Building from source . . . 198
32.3.1 Obtain the source . . . 198
32.3.2 Build the Type Annotations compiler . . . 199
32.3.3 Build the Annotation File Utilities . . . 199
32.3.4 Build the Checker Framework . . . 199
32.3.5 Build the Checker Framework Manual (this document) . . . 200
32.4 Publications . . . 200

32.5 Comparison to other tools 201
32.6 Credits, changelog, and license 202

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.

A "checker" is a tool that warns you about certain errors or gives you a guarantee that those errors do not occur. The Checker Framework comes with checkers for specific types of errors:

1. Nullness Checker for null pointer errors (see Chapter 3, page 24)
2. Initialization Checker to ensure all fields are set in the constructor (see Chapter 3.8, page 33)
3. Map Key Checker to track which values are keys in a map (see Chapter 4, page 44)
4. Interning Checker for errors in equality testing and interning (see Chapter 5, page 47)
5. Lock Checker for concurrency and lock errors (see Chapter 6, page 50)
6. Fake Enum Checker to allow type-safe fake enum patterns (see Chapter 7, page 56)
7. Tainting Checker for trust and security errors (see Chapter 8, page 59)
8. Regex Checker to prevent use of syntactically invalid regular expressions (see Chapter 9, page 61)
9. Format String Checker to ensure that format strings have the right number and type of `%` directives (see Chapter 10, page 65)
10. Internationalization Format String Checker to ensure that i18n format strings have the right number and type of `{}` directives (see Chapter 11, page 71)
11. Property File Checker to ensure that valid keys are used for property files and resource bundles (see Chapter 12, page 77)
12. Internationalization Checker to ensure that code is properly internationalized (see Chapter 12.2, page 78)
13. Signature String Checker to ensure that the string representation of a type is properly used, for example in `Class.forName` (see Chapter 13, page 80)
14. GUI Effect Checker to ensure that non-GUI threads do not access the UI, which would crash the application (see Chapter 14, page 82)
15. Units Checker to ensure operations are performed on correct units of measurement (see Chapter 15, page 86)
16. Constant Value Checker to determine whether an expression's value can be known at compile time (see Chapter 16, page 89)
17. Aliasing Checker to identify whether expressions have aliases (see Chapter 17, page 92)
18. Linear Checker to control aliasing and prevent re-use (see Chapter 18, page 96)
19. IGJ Checker for mutation errors (incorrect side effects), based on the IGJ type system (see Chapter 19, page 98)
20. Javari Checker for mutation errors (incorrect side effects), based on the Javari type system (see Chapter 20, page 102)
21. Subtyping Checker for customized checking without writing any code (see Chapter 22, page 108)
22. Third-party checkers that are distributed separately from the Checker Framework (see Chapter 23, page 111)

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 22 and 29.

1.1 How to read this manual

If you wish to get started using some particular type system from the list above, then the most effective way to read this manual is:

- Read all of the introductory material (Chapters 1–2).
- Read just one of the descriptions of a particular type system and its checker (Chapters 3–23).
- Skim the advanced material that will enable you to make more effective use of a type system (Chapters 24–32), so that you will know what is available and can find it later. Skip Chapter 29 on creating a new checker.

1.2 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 type-checker finds and prevents many errors — but it doesn't find and prevent *enough* errors. The Checker Framework lets you run an additional type-checker 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.

1.3 Installation

This section describes how to install the Checker Framework. (If you wish to use the Checker Framework from Eclipse, see the Checker Framework Eclipse Plugin webpage instead: `http://types.cs.washington.edu/checker-framework/eclipse/`.)

The Checker Framework 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 32.3, page 198).

Requirement: You must have **JDK 7** or later installed. You can get JDK 7 from Oracle or elsewhere. If you are using Apple Mac OS X, you can use Apple's implementation, SoyLatte, or the OpenJDK.

The installation process is simple! It has two required steps and one optional step.

1. Download the Checker Framework distribution:
   `http://types.cs.washington.edu/checker-framework/current/checker-framework.zip`
2. Unzip it to create a `checker-framework` directory.
3. Configure your IDE, build system, or command shell to use the Checker Framework compiler. Choose the appropriate section of Chapter 30 for javac (Section 30.1), Ant (Section 30.2), Maven (Section 30.3), Gradle (Section 30.4), IntelliJ IDEA (Section 30.5), Eclipse (Section 30.6), or tIDE (Section 30.7).

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

We recommend that you work through the Checker Framework tutorial (`http://types.cs.washington.edu/checker-framework/tutorial/`), which walks you through how to use the Checker Framework in Eclipse or on the command line.

Section 1.4 walks you through a simple example. More detailed instructions for using a checker appear in Chapter 2.

1.4 Example use: detecting a null pointer bug

This section gives a very simple example of running the Checker Framework. There is also a tutorial (`http://types.cs.washington.edu/checker-framework/tutorial/`) that gives more extensive instructions for using the Checker Framework in Eclipse or on the command line.

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

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

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

2. Run the Nullness Checker on the class. You can do that from the command line or from an IDE:
   (a) From the command line, run this command:

```
javac -processor org.checkerframework.checker.nullness.NullnessChecker GetStarted.java
```

   where the `javac` command refers to the Checker Framework compiler (see Section 30.1).
   (b) To compile within your IDE, you must have customized it to use the Checker Framework compiler and to pass the extra arguments (see Chapter 30).

   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 you can write a type, including generics and casts; see Section 2.1. Here are some examples:

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

1.5 What comes with the Checker Framework distribution

The Checker Framework distribution contains the following notable directories and files:

- `changelog.txt` The changelog.
- `checker/bin/javac` A replacement for the `javac` compiler that enables use of the Checker Framework.
- `checker/manual/` A local copy of this manual in PDF and HTML formats.
- `tutorial/` The Checker Framework tutorial.
- `checker/dist/` Contains jar files for use by advanced users:
  - `javac.jar` A Java 9 javac with additional support for Checker Framework extensions.

- `checker.jar` The Checker Framework classes.
- `jdk7.jar` and `jdk8.jar` Annotations for the JDK classes for Java 7 and Java 8 (but no class bodies).
- `checker-qual.jar` The annotation types defined by the Checker Framework. This jar file is useful to distribute with code that uses Checker Framework annotations. See Section 2.2.1.
- `checker-source.jar` The Checker Framework source code for use by IDEs.
- `checker-javadoc.jar` The Checker Framework Javadoc for use by IDEs.

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 3.3.7 and 20.2.2.) It is possible to annotate only part of your code: see Section 27.1.
2. The checker reports whether the program contains any erroneous code — that is, code that is inconsistent with the annotations.

This chapter 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 25: Advanced type system features
- Chapter 26: Suppressing warnings
- Chapter 27: Handling legacy code
- Chapter 28: Annotating libraries
- Chapter 29: How to create a new checker
- Chapter 30: Integration with external tools

Finally, there is a tutorial (`http://types.cs.washington.edu/checker-framework/tutorial/`) that walks you through using the Checker Framework in Eclipse or on the command line.

2.1 Writing annotations

The syntax of type annotations in Java is specified by the Java Language Specification (Java SE 8 edition). Java 5 permitted annotations on declarations. Java 8 also 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 MyClass this) { ... } // 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 Checker Framework 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 Checker Framework compiler. For more details, see Section 27.2.1.

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 30 for details about Ant (Section 30.2), Maven (Section 30.3), Gradle (Section 30.4), IntelliJ IDEA (Section 30.5), Eclipse (Section 30.6), and tIDE (Section 30.7), 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 1.3).

A concrete example (using the Nullness Checker) is:

```
javac -processor org.checkerframework.checker.nullness.NullnessChecker MyFile.java
```

Since the `-processor` expects a fully-qualified class name, its argument can often be verbose. There is shorthand for invoking checkers that are built into the Checker Framework. See Section 2.2.4.

The checker is run on only the Java files 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.

You can always compile the code without the `-processor` command-line option, but in that case no checking of the type annotations is performed. Furthermore, only explicitly-written annotations are written to the `.class` file; defaulted annotations are not, and this will interfere with type-checking of clients that use your code. Therefore, it is strongly recommended that whenever you are creating `.class` files that will be distributed or compiled against, you run the type-checkers for all the annotations that your have written.

2.2.1 Distributing your annotated project

You have two main options for distributing your compiled code (`.jar` files).

- Option 1: no annotations appear in the `.jar` files. There is no run-time dependence on the Checker Framework, and the distributed `.jar` files are not useful for pluggable type-checking of client code.
  Write annotations in comments (see Section 27.2.1). Developers perform pluggable type-checking in-house to detect errors and verify their absence. To create the distributed `.jar` files, use a normal Java compiler, which ignores the annotations.
- Option 2: annotations appear in the `.jar` files. The distributed `.jar` files can be used for pluggable type-checking of client code. The `.jar` files are only compatible with a Java 8 JVM, unless you do extra work (see Section 27.2.5).
  Write annotations in comments or not in comments (it doesn't matter which). Developers perform pluggable type-checking in-house to detect errors and verify their absence. When you create `.class` files, use the Checker Framework compiler (Section 30) and running each relevant type system. Create the distributed `.jar` files from those `.class` files, and also include the contents of `checker-framework/checker/dist/checker-qual.jar` from the Checker Framework distribution, to define the annotations.

2.2.2 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.

Unsound checking: ignore some errors

- `-AskipUses`, `-AonlyUses` Suppress all errors and warnings at all uses of a given class — or at all uses except those of a given class. See Section 26.4
- `-AskipDefs`, `-AonlyDefs` Suppress all errors and warnings within the definition of a given class — or everywhere except within the definition of a given class. See Section 26.5
- `-AsuppressWarnings` Suppress all warnings matching the given key; see Section 26.3
- `-AignoreRawTypeArguments` Ignore subtype tests for type arguments that were inferred for a raw type. If possible, it is better to write the type arguments. See Section 24.1.1.
- `-AassumeSideEffectFree` Unsoundly assume that every method is side-effect-free; see Section 25.4.3.
- `-AassumeAssertionsAreEnabled`, `-AassumeAssertionsAreDisabled` Whether to assume that assertions are enabled or disabled; see Section 25.4.4.
- `-AsafeDefaultsForUnannotatedBytecode` Whether the checker should use conservative defaults for unannotated bytecode; see Section 25.3.5.
- `-AuseSafeDefaultsForUnannotatedSourceCode` Outside the scope of any relevant `@AnnotatedFor` annotation, use conservative default annotations and suppress all type-checking warnings; see Section 28.1.

More sound (strict) checking: enable errors that are disabled by default

- `-AcheckPurityAnnotations` Check the bodies of methods marked `@SideEffectFree`, `@Deterministic`, and `@Pure` to ensure the method satisfies the annotation. By default, the Checker Framework unsoundly trusts the method annotation. See Section 25.4.3.
- `-AinvariantArrays` Make array subtyping invariant; that is, two arrays are subtypes of one another only if they have exactly the same element type. By default, the Checker Framework unsoundly permits covariant array subtyping, just as Java does. See Section 25.1.
- `-AconcurrentSemantics` Whether to assume concurrent semantics (field values may change at any time) or sequential semantics; see Section 31.4.4.

Type-checking modes: enable/disable functionality

- `-Alint` Enable or disable optional checks; see Section 26.6.
- `-AshowSuppressWarningKeys` With each warning, show all possible keys to suppress that warning; see Section 26.3
- `-AsuggestPureMethods` Suggest methods that could be marked `@SideEffectFree`, `@Deterministic`, or `@Pure`; see Section 25.4.3.
- `-AcheckCastElementType` In a cast, require that parameterized type arguments and array elements are the same. By default, the Checker Framework unsoundly permits them to differ, just as Java does. See Section 24.1.6 and Section 25.1.
- `-Awarns` Treat checker errors as warnings. If you use this, you may wish to also supply `-Xmaxwarns 10000`, because by default `javac` prints at most 100 warnings.

Partially-annotated libraries

- `-Astubs` List of stub files or directories; see Section 28.2.1.
- `-AstubWarnIfNotFound` Warn if a stub file entry could not be found; see Section 28.2.1.
- `-AuseSafeDefaultsForUnannotatedSourceCode` Outside the scope of any relevant `@AnnotatedFor` annotation, use conservative default annotations and suppress all type-checking warnings; see Section 28.1.

Debugging

- `-AprintAllQualifiers`, `-Adetailedmsgtext`, `-AprintErrorStack`, `-Anomsgtext` Amount of detail in messages; see Section 29.9.1.

- `-Aignorejdkastub`, `-Anocheckjdk -AstubDebug`, Stub and JDK libraries; see Section 29.9.2
- `-Afilenames`, `-Ashowchecks` Progress tracing; see Section 29.9.3
- `-AoutputArgsToFile` Output the compiler command-line arguments to a file. Useful when this is not fully in your control, such as when the Checker Framework is run from Maven. See Section 29.9.4
- `-Aflowdotdir`, `-AresourceStats` Miscellaneous debugging options; see Section 29.9.5

Some checkers support additional options, which are described in that checker's manual section. For example, `-Aquals` tells the Subtyping Checker (see Chapter 22) and the Fenum Checker (see Chapter 7) which annotations to check.

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 Subtyping Checker; see Section 22.2
- `-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 2.2.3
- `-implicit:class` Suppresses warnings about implicitly compiled files (not named on the command line); see Section 30.2
- `-XDTA:noannotationsincomments` and `-XDTA:spacesincomments` to turn off parsing annotation comments and to turn on parsing annotation comments even when they contain spaces; applicable only to the Checker Framework compiler; see Section 27.2.1
- `-J` Supply an argument to the JVM that is running javac
- `-doe` To "dump on error", that is, output a stack trace whenever a compiler warning/error is produced. Useful when debugging the compiler or a checker.

2.2.3 Checker auto-discovery

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

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

```
org.checkerframework.checker.nullness.NullnessChecker
org.checkerframework.checker.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.

2.2.4 Shorthand for built-in checkers

The `-processor` flag expects fully-qualified class names. For checkers that are packaged with the Checker Framework, the fully-qualified name can be quite long. Therefore, when running a built-in checker, you may omit the package name and the `Checker` suffix. The following three commands are equivalent:

```
javac -processor org.checkerframework.checker.nullness.NullnessChecker MyFile.java
javac -processor NullnessChecker MyFile.java
javac -processor nullness MyFile.java
```

This feature will work when multiple checkers are specified. For example:

```
javac -processor NullnessChecker,RegexChecker MyFile.java
javac -processor nullness,regex MyFile.java
```

This feature does not apply to Javac @argfiles.

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 5) 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 `-AskipUses`, `-AonlyUses`, `-AskipDefs` or `-AonlyDefs` properties (see Section 26).
  - Suppression of warnings, such as via the `@SuppressWarnings` annotation (see Section 26).
  - 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.
- The Checker Framework is, by default, unsound in a few places where a conservative analysis would issue too many false positive warnings. These are listed in Section 2.2.2. You can supply a command-line argument to make the Checker Framework sound for each of these cases.
- 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 32.2).

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.

Start small, focusing on some specific property that matters to you and on the most mission-critical or error-prone part of your code. 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.

Start by annotating just part of your program. Be systematic, such as annotating an entire class at a time (not just some of the methods) so that you don't lose track of your work. You may find it helpful to start annotating the leaves of the call tree — that is, start with methods/classes/packages that have few dependencies on other code or, equivalently, start with code that a lot of your other code depends on. The reason for this is that 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. Then annotate signatures and fields; there is no need to annotate method bodies. 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`.

After you have annotated all the signatures, run the checker. Then, fix bugs in code and add/modify annotations as necessary. Don't get discouraged if you see many type-checker warnings at first. Often, adding just a few missing annotations will eliminate many warnings, and you'll be surprised how fast the process goes overall.

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 26, which tells you what to do when you are unable to eliminate checker warnings, and Chapter 28, which tells you how to annotate libraries that your code uses.

2.4.2 Do not annotate local variables unless necessary

The checker infers annotations for local variables (see Section 25.4). 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 26), etc.

2.4.3 Annotations indicate normal behavior

You should use annotations to specify *normal* behavior. The annotations indicate all the values that you *want* to flow to a 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 3.2) 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 (whether or not it throws a NullPointerException).

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.

If a method can possibly throw exception because its parameter is `null`, then that parameter's type should be `@NonNull`, which guarantees that the type-checker will issue a warning for every client use that has the potential to cause an exception. Don't write `@Nullable` on the parameter just because there exist some executions that don't necessarily throw an exception.

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 19) is `new @Immutable MyClass()` or `new @Mutable MyClass()`, where you know that every other reference to the class is annotated `@ReadOnly`.

2.4.6 What to do if a checker issues a warning about your code

When you first run a type-checker on your code, it is likely to issue warnings or errors. For each warning, try to understand why the checker issues it. (If you think the warning is wrong, then formulate an argument about why your code is actually correct; also see Section 32.1.2.) For example, if you are using the Nullness Checker (Chapter 3, page 24), try to understand why it cannot prove that no null pointer exception ever occurs. There are three general reasons, listed below. You will need to examine your code, and possibly write test cases, to understand the reason.

1. There is a bug in your code, such as an actual possible null dereference. Fix your code to prevent that crash.
2. There is a weakness in the annotations. Improve the annotations. For example, continuing the Nullness Checker example, if a particular variable is annotated as `@Nullable` but it actually never contains `null` at run time, then change the annotation to `@NonNull`. The weakness might be in the annotations in your code, or in the annotations in a library that your code calls. Another possible problem is that a library is unannotated (see Chapter 28, page 151).
3. There is a weakness in the type-checker. Then your code is safe — it never suffers the error at run time — but the checker cannot prove this fact. The checker is not omniscient, and some tricky coding paradigms are beyond its analysis capabilities. In this case, you should suppress the warning; see Chapter 26, page 140. (Alternatively, if the weakness is a bug in the checker, then please report the bug; see Chapter 32.2, page 198.)

If you have trouble understanding a Checker Framework warning message, you can search for its text in this manual. Oftentimes there is an explanation of what to do.

Also see Chapter 32, Troubleshooting.

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 3.1 for more details about the guarantee and what is checked.

The most important annotations supported by the Nullness Checker are `@NonNull` and `@Nullable`. `@NonNull` is rarely written, because it is the default. All of the annotations are explained in Section 3.2.

To run the Nullness Checker, supply the `-processor org.checkerframework.checker.nullness.NullnessChecker` command-line option to javac. For examples, see Section 3.5.

The NullnessChecker is actually an ensemble of three pluggable type-checkers that work together: the Nullness Checker proper (which is the main focus of this chapter), the Initialization Checker (Section 3.8), and the Map Key Checker (Chapter 4, page 44). Their type hierarchies are completely independent, but they work together to provide precise nullness checking.

3.1 What the Nullness Checker checks

The checker issues a warning in these 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.
   As a special case of an of `@NonNull` type becoming null, the checker also warns whenever a field of `@NonNull` type is not initialized in a constructor. Also see the discussion of the `-Alint=uninitialized` command-line option below.

This example illustrates the programming errors that the checker detects:

```
@Nullable 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 3.8.6) and of map key annotations (see Chapter 4, page 44).

The checker performs additional checks if certain `-Alint` command-line options are provided. (See Section 26.6 for more details about the `-Alint` command-line option.)

1. If you supply the `-Alint=redundantNullComparison` command-line option, then the checker warns when a null check is performed against a value that is guaranteed to be non-null, as in `("m" == null)`. Such a check is unnecessary and might indicate a programmer error or misunderstanding. The lint option is disabled by default because sometimes such checks are part of ordinary defensive programming.
2. If you supply the `-Alint=uninitialized` command-line option, then the checker warns if a constructor fails to initialize any field, including `@Nullable` types and primitive types. Such a warning is unrelated to whether your code might throw a null pointer exception. However, you might want to enable this warning because it is better code style to supply an explicit initializer, even if there is a default value such as `0` or `false`. This command-line option does not affect the Nullness Checker's tests that fields of `@NonNull` type are initialized — such initialization is mandatory, not optional.
3. If you supply the `-Alint=forbidnonnullarraycomponents` command-line option, then the checker warns if it encounters an array creation with a non-null component type. See Section 3.3.4 for a discussion.

3.2 Nullness annotations

The Nullness Checker uses three separate type hierarchies: one for nullness, one for rawness (Section 3.8.6), and one for map keys (Chapter 4, page 44) The Nullness Checker has four varieties of annotations: nullness type qualifiers, nullness method annotations, rawness type qualifiers, and map key type qualifiers.

3.2.1 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 3.3.2).

`@PolyNull` indicates qualifier polymorphism. For a description of `@PolyNull`, see Section 24.2.

`@MonotonicNonNull` 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 `MonotonicNonNull` field is first read within a method, the field cannot be assumed to be non-null. The benefit of MonotonicNonNull over Nullable is its different interaction with flow-sensitive type qualifier refinement (Section 25.4). After a check of a MonotonicNonNull 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.

It is permitted to initialize a MonotonicNonNull field to null, but the field may not be assigned to null anywhere else in the program. If you supply the `noInitForMonotonicNonNull` lint flag (for example, supply `-Alint=noInitForMonotonicNonNull` on the command line), then @MonotonicNonNull fields are not allowed to have initializers.

Use of `@MonotonicNonNull` on a static field is a code smell: it may indicate poor design. You should consider whether it is possible to make the field a member field that is set in the constructor.

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.)

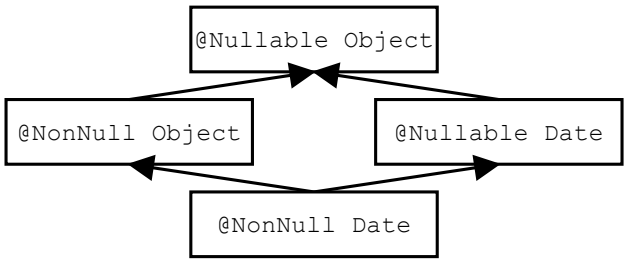


Figure 3.1: Partial type hierarchy for the Nullness type system. Java's `Object` is expressed as `@Nullable Object`. Programmers can omit most type qualifiers, because the default annotation (Section 3.3.2) is usually correct. The Nullness Checker verifies three type hierarchies: this one for nullness, one for initialization (Section 3.8), and one for map keys (Chapter 4, page 44).

3.2.2 Nullness method annotations

The Nullness Checker supports several annotations that specify method behavior. These are declaration annotations, not type annotations: they apply to the method itself rather than to some particular type.

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

`@EnsuresNonNull`

`@EnsuresNonNullIf` indicates a method postcondition. With `@EnsuresNonNull`, the given expressions are non-null after the method returns; this is useful for a method that initializes a field, for example. With `@EnsuresNonNullIf`, if the annotated method returns the given boolean value (true or false), then the given expressions are non-null. See Section 3.3.3 and the Javadoc for examples of their use.

3.2.3 Initialization qualifiers

The Nullness Checker invokes an Initialization Checker, whose annotations indicate whether an object is fully initialized — that is, whether all of its fields have been assigned.

`@Initialized`
`@UnknownInitialization`
`@UnderInitialization`

Use of these annotations can help you to type-check more code. Figure 3.3 shows its type hierarchy. For details, see Section 3.8.

A slightly simpler variant, called the Rawness Initialization Checker, is also available:

`@Raw`
`@NonRaw`
`@PolyRaw`

Figure 3.5 shows its type hierarchy. For details, see Section 3.8.6.

3.2.4 Map key qualifiers

`@KeyFor`

indicates that a value is a key for a given map — that is, indicates whether `map.containsKey(value)` would evaluate to `true`.

This annotation is checked by a Map Key Checker (Chapter 4, page 44) that the Nullness Checker invokes. The `@KeyFor` annotation enables the Nullness Checker to treat calls to `Map.get` precisely rather than assuming it may always return null. In particular, a call `mymap.get(mykey)` returns a non-`null` value if two conditions are satisfied:

1. mymap's values are all non-null; that is, mymap was declared as Map<*KeyType*, @NonNull *ValueType*>. Note that @NonNull is the default type, so it need not be written explicitly.
2. mykey is a key in mymap; that is, mymap.containsKey(mykey) returns true. You express this fact to the Nullness Checker by declaring mykey as @KeyFor("mymap") *KeyType* mykey. For a local variable, you generally do not need to write the @KeyFor("mymap") type qualifier, because it can be inferred.

If either of these two conditions is violated, then mymap.get(mykey) has the possibility of returning null.

3.3 Writing nullness annotations

3.3.1 Implicit qualifiers

As described in Section 25.3, 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.

3.3.2 Default annotation

Unannotated references are treated as if they had a default annotation. The standard defaulting rule is CLIMB-to-top, described in Section 25.3.2. Its effect is to default all types to @NonNull, except that @Nullable is used for casts, locals, instanceof, and implicit bounds. A user can choose a different defaulting rule.

3.3.3 Conditional nullness

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

Consider java.lang.Class. Method Class.getComponentType() may return null, but it is specified to return a non-null value if Class.isArray() is true. 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 Class {
  @EnsuresNonNullIf(expression="getComponentType()", result=true)
  public native boolean isArray();

  public native @Nullable Class<?> getComponentType();
}
```

A client that checks that a Class reference is indeed that of an array, can then de-reference the result of Class.getComponentType safely without any nullness check. The Checker Framework source code itself uses such a pattern:

```
if (clazz.isArray()) {
  // no possible null dereference on the following line
  TypeMirror componentType = typeFromClass(clazz.getComponentType());
  ...
}
```

Another example is Queue.peek and Queue.poll, which return non-null if isEmpty returns false.

The argument to @EnsuresNonNullIf is a Java expression, including method calls (as shown above), method formal parameters, fields, etc.; for details, see Section 25.5. More examples of the use of these annotations appear in the Javadoc for @EnsuresNonNullIf.

3.3.4 Nullness and arrays

The components of a newly created object of reference type are all null. Only after initialization can the array actually be considered to contain non-null components. Therefore, the following is not allowed:

```
@NonNull Object [] oa = new @NonNull Object[10]; // error
```

Instead, one creates a nullable or lazy-nonnull array, initializes each component, and then assigns the result to a non-null array:

```
@MonotonicNonNull Object [] temp = new @MonotonicNonNull Object[10];
for (int i = 0; i < temp.length; ++i) {
  temp[i] = new Object();
}
@SuppressWarnings("nullness") // temp array is now fully initialized
@NonNull Object [] oa = temp;
```

Note that the checker is currently not powerful enough to ensure that each array component was initialized. Therefore, the last assignment needs to be trusted: that is, a programmer must verify that it is safe, then write a `@SuppressWarnings` annotation.

You need to supply the `-Alint=forbidnonnullarraycomponents` command-line option to enable this behavior. For backwards-compatibility reasons, the default behavior is currently to unsoundly allow non-null array components.

3.3.5 Run-time checks for nullness

When you perform a run-time check for nullness, such as `if (x != null) ...`, then the Nullness Checker refines the type of `x` to `@NonNull` within the scope of the test. For more details, see Section 25.4.

3.3.6 Additional details

The Nullness Checker does some special checks in certain circumstances, in order to soundly reduce the number of warnings that it produces.

For example, a call to `System.getProperty(String)` can return null in general, but it will not return null if the argument is one of the built-in-keys listed in the documentation of `System.getProperties()`. The Nullness Checker is aware of this fact, so you do not have to suppress a warning for a call like `System.getProperty("line.separator")`. The warning is still issued for code like this:

```
final String s = "line.separator";
nonNullvar = System.getProperty(s);
```

though that case could be handled as well, if desired. (Suppression of the warning is, strictly speaking, not sound, because a library that your code calls, or your code itself, could perversely change the system properties; the Nullness Checker assumes this bizarre coding pattern does not happen.)

3.3.7 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 25.4), 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 3.3.2) your code uses. You only need one of these tools.

- Inference of `@Nullable`: If your code uses the standard CLIMB-to-top default (Section 25.3.2) 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.

3.4 Suppressing nullness warnings

When the Nullness Checker reports a warning, it's best to change the code or its annotations, to eliminate the warning. Alternately, you can suppress the warning, which does not change the code but prevents the Nullness Checker from reporting this particular warning to you.

The Checker Framework supplies several ways to suppress warnings, most notably the `@SuppressWarnings("nullness")` annotation (see Section 26). An example use is

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

void myMethod() {
  @SuppressWarnings("nullness") // with argument x, getObject always returns a non-null value
  @NonNull Object o2 = getObject(x);
```

The Nullness Checker supports an additional warning suppression key, `nullness:generic.argument`. 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 31.6.1, page 187.

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. In such cases, you can use the `@AssumeAssertion` string in an `assert` message (see Section 26.2).

If you need to suppress a warning within an expression, then sometimes writing an assertion is not convenient. In such a case, you can suppress warnings by writing a call to the `NullnessUtils.castNonNull` method. The rest of this section discusses the `castNonNull` method.

The Nullness Checker considers both the return value, and also the argument, to be non-null after the `castNonNull` method call. The Nullness Checker issues no warnings in any of the following code:

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

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

// It is possible, but not recommmended, to use castNonNull as a statement:
// (It would be better to write an assert statement with @AssumeAssertion
// in its message, instead.)
castNonNull(possiblyNull3);
possiblyNull3.toString();
```

The `castNonNull` method throws `AssertionError` if Java assertions are enabled and the argument is `null`. However, it is not intended for general defensive programming; see Section 26.2.1.

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 26.2.1 for an explanation of the distinction.

The Nullness Checker introduces a new method, rather than re-using an existing method such as `org.junit.Assert.assertNotNull(` or `com.google.common.base.Preconditions.checkNotNull(Object)`. Those methods are commonly used for defensive programming, so it is impossible to know the programmer's intent when writing them. Therefore, it is important to have a method call that is used only for warning suppression. See Section 26.2.1 for a discussion of the distinction between warning suppression and defensive programming.

3.5 Examples

3.5.1 Tiny examples

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

```
javac -processor org.checkerframework.checker.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 org.checkerframework.checker.nullness.NullnessChecker examples/NullnessExampleWithWarni
```

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

3.5.2 Example annotated source code

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`.

3.6 Tips for getting started

Here are some tips about getting started using the Nullness Checker on a legacy codebase. For more generic advice (not specific to the Nullness Checker), see Section 2.4.1.

Your goal is to add `@Nullable` annotations to the types of any variables that can be null. (The default is to assume that a variable is non-null unless it has a `@Nullable` annotation.) Then, you will run the Nullness Checker. Each of its errors indicates either a possible null pointer exception, or a wrong/missing annotation. When there are no more warnings from the checker, you are done!

We recommend that you start by searching the code for occurrences of `null` in the following locations; when you find one, write the corresponding annotation:

- in Javadoc: add `@Nullable` annotations to method signatures (parameters and return types).
- `return null`: add a `@Nullable` annotation to the return type of the given method.
- *`param`* `== null`: when a formal parameter is compared to `null`, then in most cases you can add a `@Nullable` annotation to the formal parameter's type

- *TypeName field* = null;: when a field is initialized to null in its declaration, then it needs either a @Nullable or a @MonotonicNonNull annotation. If the field is always set to a non-null value in the constructor, then you can just change the declaration to *Type field*;, without an initializer, and write no type annotation (because the default is @NonNull).
- declarations of contains, containsKey, containsValue, equals, get, indexOf, lastIndexOf, and remove (with Object as the argument type): change the argument type to @Nullable Object; for remove, also change the return type to @Nullable Object.

You should ignore all other occurrences of null within a method body. In particular, you (almost) never need to annotate local variables.

Only after this step should you run ant to invoke the Nullness Checker. The reason is that it is quicker to search for places to change than to repeatedly run the checker and fix the errors it tells you about, one at a time.

Here are some other tips:

- In any file where you write an annotation such as @Nullable, don't forget to add import org.checkerframework.checker.nullness.qual.*;.
- To indicate an array that can be null, write, for example: int @Nullable [].
  By contrast, @Nullable Object [] means a non-null array that contains possibly-null objects.
- If you know that a particular variable is definitely not null, but the Nullness Checker estimates that the variable might be null, then you can make the Nullness Checker trust your judgment by writing an assertion (see Section 26.2):

  ```
  assert var != null : "@AssumeAssertion(nullness)";
  ```

- To indicate that a routine returns the same value every time it is called, use @Pure (see Section 25.4.3).
- To indicate a method precondition (a contract stating the conditions under which a client is allowed to call it), you can use annotations such as @RequiresNonNull (see Section 3.2.2).

3.7 Other tools for nullness checking

The Checker Framework's nullness annotations are similar to annotations used in IntelliJ IDEA, FindBugs, JML, the JSR 305 proposal, NetBeans, and other tools. Also see Section 32.5 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 your code is already annotated with a different nullness annotation, you can reuse that effort. The Checker Framework comes with cleanroom re-implementations of annotations from other tools. It treats them exactly as if you had written the corresponding annotation from the Nullness Checker, as described in Figure 3.2.

Alternately, the Checker Framework can process those other annotations (as well as its own, if they also appear in your program). The Checker Framework has its own definition of the annotations on the left side of Figure 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. Also, note FindBugs's non-standard meaning for @Nullable (Section 3.7.2).

Because some of the names are the same (NonNull, Nullable), you can import at most one of the annotations with conflicting names; the other(s) must be written out fully rather than imported.

Note that some older tools interpret array and vararg declarations inconsistently with the Java specification. For example, they might interpret @NonNull Object [] as "non-null array of objects", rather than as "array of non-null objects" which is the correct Java interpretation. Such an interpretation is unfortunate and confusing. See Section 31.5.3 for some more details about this issue.

android.annotation.NonNull	
android.support.annotation.NonNull	
com.sun.istack.internal.NotNull	
edu.umd.cs.findbugs.annotations.NonNull	
javax.annotation.Nonnull	⇒ org.checkerframework.checker.nullness.qual.NonNull
javax.validation.constraints.NotNull	
org.eclipse.jdt.annotation.NonNull	
org.jetbrains.annotations.NotNull	
org.jmlspecs.annotation.NonNull	
org.netbeans.api.annotations.common.NonNull	

android.annotation.Nullable	
android.support.annotation.Nullable	
com.sun.istack.internal.Nullable	
edu.umd.cs.findbugs.annotations.Nullable	
edu.umd.cs.findbugs.annotations.CheckForNull	
edu.umd.cs.findbugs.annotations.UnknownNullness	
javax.annotation.Nullable	⇒ org.checkerframework.checker.nullness.qual.Nullable
javax.annotation.CheckForNull	
org.eclipse.jdt.annotation.Nullable	
org.jetbrains.annotations.Nullable	
org.jmlspecs.annotation.Nullable	
org.netbeans.api.annotations.common.NullAllowed	
org.netbeans.api.annotations.common.CheckForNull	
org.netbeans.api.annotations.common.NullUnknown	

Figure 3.2: Correspondence between other nullness annotations and the Checker Framework's annotations.

3.7.1 Which tool is right for you?

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

The 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 desire: 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 32.2) so that we can enhance the Checker Framework.

3.7.2 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.

With FindBugs, you annotate a declaration, which suppresses 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. Thus, the Checker Framework supports both use cases, whereas FindBugs supports only one and gives the programmer less flexibility.

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. As a separate issue, 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.)

3.7.3 Relationship to `Optional<T>`

Many null pointer exceptions occur because the programmer forgets to check whether a reference is null before dereferencing it. Java 8's `Optional<T>` class provides a partial solution: you cannot dereference the contained value without calling the `get` method.

However, the use of `Optional` for this purpose is unsatisfactory. First, it adds syntactic complexity, making your code longer and harder to read. (The `Optional` class provides some operations, such as `map` and `orElse`, that you would otherwise have to write; without these its code bloat would be even worse.) Second, there is no guarantee that the programmer remembers to call `isPresent` before calling `get`. Thus, use of `Optional` doesn't solve the underlying problem — it merely converts a `NullPointerException` into a `NoSuchElementException` exception, and in either case your code crashes.

The Nullness Checker does not suffer these limitations. It works with existing code and types, it ensures that you check for null wherever necessary, and it infers when the check for null is not necessary based on previous statements in the method.

3.8 Initialization Checker

Every object's fields start out as null. By the time the constructor finishes executing, the `@NonNull` fields have been set to a different value. Your code can suffer a NullPointerException when using a `@NonNull` field, if your code uses

the field during initialization. The Nullness Checker prevents this problem by warning you anytime that you may be accessing an uninitialized field. This check is useful because it prevents errors in your code. However, the analysis can be confusing to understand. If you wish to disable the initialization checks, pass the command-line argument `-AsuppressWarnings=uninitialized` when running the Nullness Checker. You will no longer get a guarantee of no null pointer exceptions, but you can still use the Nullness Checker to find most of the null pointer problems in your code.

An object is partially initialized from the time that its constructor starts until its constructor finishes. This is relevant to the Nullness Checker because while the constructor is executing — that is, before initialization completes — a `@NonNull` field may be observed to be null, until that field is set. In particular, the Nullness Checker issues a warning for code like this:

```
public class MyClass {
  private @NonNull Object f;
  public MyClass(int x, int y) {
    // Error because constructor contains no assignment to this.f.
    // By the time the constructor exits, f must be initialized to a non-null value.
  }
  public MyClass(int x) {
    // Error because this.f is accessed before f is initialized.
    // At the beginning of the constructor's execution, accessing this.f
    // yields null, even though field f has a non-null type.
    this.f.toString();
  }
  public MyClass(int x, int y, int z) {
    m();
  }
  public void m() {
    // Error because this.f is accessed before f is initialized,
    // even though the access is not in a constructor.
    // When m is called from the constructor, accessing f yields null,
    // even though field f has a non-null type.
    this.f.toString();
  }
```

When a field `f` is declared with a `@NonNull` type, then code can depend on the fact that the field is not `null`. However, this guarantee does not hold for a partially-initialized object.

The Nullness Checker uses three annotations to indicate whether an object is initialized (all its `@NonNull` fields have been assigned), under initialization (its constructor is currently executing), or its initialization state is unknown.

These distinctions are mostly relevant within the constructor, or for references to `this` that escape the constructor (say, by being stored in a field or passed to a method before initialization is complete). Use of initialization annotations is rare in most code.

The most common use for the `@UnderInitialization` annotation is for a helper routine that is called by constructor. For example:

```
class MyClass {
  Object field1;
  Object field2;
  Object field3;

  public MyClass(String arg1) {
    this.field1 = arg1;
    init_other_fields();
```

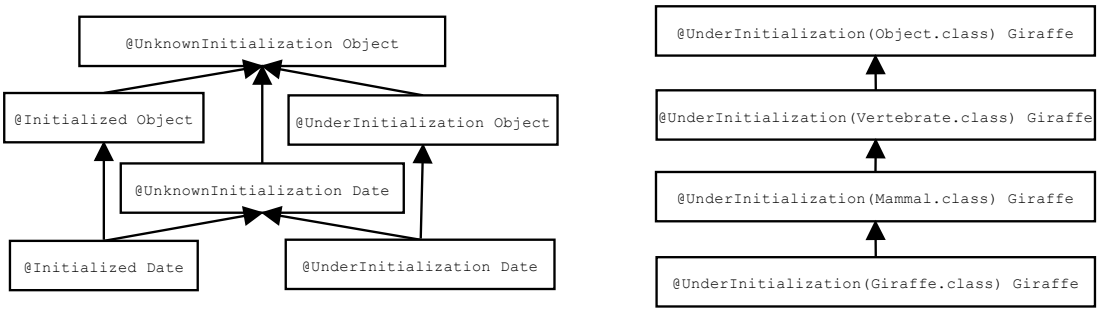


Figure 3.3: Partial type hierarchy for the Initialization type system. `@UnknownInitialization` and `@UnderInitialization` each take an optional parameter indicating how far initialization has proceeded, and the right side of the figure illustrates its type hierarchy in more detail.

```
  }

  // A helper routine that initializes all the fields other than field1.
  @EnsuresNonNull({"field2", "field3"})
  private void init_other_fields(@UnderInitialization(MyClass.class) MyClass this) {
    field2 = new Object();
    field3 = new Object();
  }
}
```

For compatibility with Java 6 and 7, you can write the receiver parameter in comments (see Section 27.2.1):

```
private void init_other_fields(/*>>>@UnderInitialization(MyClass.class) MyClass this*/) {
```

3.8.1 Initialization qualifiers

The initialization hierarchy is shown in Figure 3.3. The initialization hierarchy contains these qualifiers:

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

`@UnknownInitialization` indicates a type that may contain 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. `@UnknownInitialization` takes a parameter that is the class the object is definitely initialized up to. For instance, the type `@UnknownInitialization(Foo.class)` denotes an object in which every fields declared in `Foo` or its superclasses is initialized, but other fields might not be. Just `@UnknownInitialization` is equivalent to `@UnknownInitialization(Object.class)`.

`@UnderInitialization` indicates a type that contains a partially-initialized object that is under initialization — that is, its constructor is currently executing. It is otherwise the same as `@UnknownInitialization`. Within the constructor, `this` has `@UnderInitialization` type until all the `@NonNull` fields have been assigned.

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 `@UnknownInitialization` or as `@UnderInitialization`.

If a reference has `@UnknownInitialization` or `@UnderInitialization` type, then all of its `@NonNull` fields are treated as `@MonotonicNonNull`: when read, they are treated as being `@Nullable`, but when written, they are treated as being `@NonNull`.

The initialization hierarchy is orthogonal to the nullness hierarchy. It is legal for a reference to be `@NonNull @UnderInitialization`, `@Nullable @UnderInitialization`, `@NonNull @Initialized`, or `@Nullable @Initialized`. The nullness hierarchy tells you about the reference itself: might the reference be null? The initialization hierarchy tells

Declarations	Expression	Expression's nullness type, or checker error

`class C {`
`  @NonNull Object f;`
`  @Nullable Object g;`
`  ...`
`}`		

`@NonNull @Initialized C a;`	`a`	`@NonNull`
	`a.f`	`@NonNull`
	`a.g`	`@Nullable`
`@NonNull @UnderInitialization C b;`	`b`	`@NonNull`
	`b.f`	`@MonotonicNonNull`
	`b.g`	`@Nullable`
`@Nullable @Initialized C c;`	`c`	`@Nullable`
	`c.f`	error: deref of nullable
	`c.g`	error: deref of nullable
`@Nullable @UnderInitialization C d;`	`d`	`@Nullable`
	`d.f`	error: deref of nullable
	`d.g`	error: deref of nullable

Figure 3.4: Examples of the interaction between nullness and initialization. Declarations are shown at the left for reference, but the focus of the table is the expressions and their nullness type or error.

you about the `@NonNull` fields in the referred-to object: might those fields be temporarily null in contravention of their type annotation? Figure 3.4 contains some examples.

3.8.2 How an object becomes initialized

Within the constructor, `this` starts out with `@UnderInitialization` type. As soon as all of the `@NonNull` fields have been initialized, then `this` is treated as initialized. (See Section 3.8.3 for a slight clarification of this rule.)

The Initialization Checker issues an error if the constructor fails to initialize any `@NonNull` field. This ensures that the object is in a legal (initialized) 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.

All `@NonNull` fields must either have a default in the field declaration, or be assigned in the constructor or in a helper method that the constructor calls. If your code initializes (some) fields in a helper method, you will need to annotate the helper method with an annotation such as `@EnsuresNonNull({"field1", "field2"})` for all the fields that the helper method assigns. It's a bit odd, but you use that same annotation, `@EnsuresNonNull`, to indicate that a primitive field has its value set in a helper method, which is relevant when you supply the `-Alint=uninitialized` command-line option (see Section 3.1).

3.8.3 Partial initialization

So far, we have discussed initialization as if it is an all-or-nothing property: an object is non-initialized until initialization completes, and then it is initialized. The full truth is a bit more complex: during the initialization process an object can be partially initialized, and as the object's superclass constructors complete, its initialization status is updated. The Initialization Checker lets you express such properties when necessary.

Consider a simple example:

```
class A {
  Object a;
  A() {
    a = new Object();
```

```
  }
}
class B extends A {
  Object b;
  B() {
    super();
    b = new Object();
  }
}
```

Consider what happens during execution of `new B()`.

1. `B`'s constructor begins to execute. At this point, neither the fields of `A` nor those of `B` have been initialized yet.
2. `B`'s constructor calls `A`'s constructor, which begins to execute. No fields of `A` nor of `B` have been initialized yet.
3. `A`'s constructor completes. Now, all the fields of `A` have been initialized, and their invariants (such as that field `a` is non-null) can be depended on. However, because `B`'s constructor has not yet completed executing, the object being constructed is not yet fully initialized. When treated as an `A` (e.g., if only the `A` fields are accessed), the object is initialized, but when treated as a `B`, the object is still non-initialized.
4. `B`'s constructor completes. The object is initialized when treated as an `A` or a `B`. (And, the object is fully initialized if `B`'s constructor was invoked via a `new B()`. But the type system cannot assume that – there might be a `class C extends B { ... }`, and `B`'s constructor might have been invoked from that.)

At any moment during initialization, the superclasses of a given class can be divided into those that have completed initialization and those that have not yet completed initialization. More precisely, at any moment there is a point in the class hierarchy such that all the classes above that point are fully initialized, and all those below it are not yet initialized. As initialization proceeds, this dividing line between the initialized and uninitialized classes moves down the type hierarchy.

The Nullness Checker lets you indicate where the dividing line is between the initialized and non-initialized classes. The `@UnderInitialization(`*`classliteral`*`)` indicates the first class that is known to be fully initialized. When you write `@UnderInitialization(OtherClass.class) MyClass x;`, that means that variable `x` is initialized for `OtherClass` and its superclasses, and `x` is (possibly) uninitialized for `MyClass` and all subclasses.

We can now state a clarification of Section 3.8.2's rule for an object becoming initialized. As soon as all of the `@NonNull` fields in class *C* have been initialized, then `this` is treated as `@UnderInitialization(`*`C`*`)`, rather than treated as simply `@Initialized`.

The example above lists 4 moments during construction. At those moments, the type of the object being constructed is:

1. `@UnderInitialization B`
2. `@UnderInitialization A`
3. `@UnderInitialization(A.class) A`
4. `@UnderInitialization(B.class) B`

3.8.4 How to handle warnings

There are several ways to address a warning "error: the constructor does not initialize fields: ...".

- Declare the field as `@Nullable`. Recall that if you did not write an annotation, the field defaults to `@NonNull`.
- Declare the field as `@MonotonicNonNull`. This is appropriate if the field starts out as `null` but is later set to a non-null value. You may then wish to use the `@EnsuresNonNull` annotation to indicate which methods set the field, and the `@RequiresNonNull` annotation to indicate which methods require the field to be non-null.
- Initialize the field in the constructor or in the field's initializer, if the field should be initialized. (In this case, the Initialization Checker has found a bug!)

Do *not* initialize the field to an arbitrary non-null value just to eliminate the warning. Doing so degrades your code: it introduces a value that will confuse other programmers, and it converts a clear NullPointerException into a more obscure error.

If your code calls an instance method from a constructor, you may see a message such as the following:

```
Foo.java:123: error: call to initHelper() not allowed on the given receiver.
    initHelper();
               ^
  found   : @UnderInitialization(com.google.Bar.class) @NonNull MyClass
  required: @Initialized @NonNull MyClass
```

The problem is that the current object (`this`) is under initialization, but the receiver formal parameter (Section 31.5.1) of method `initHelper()` is implicitly annotated as `@Initialized`. If `initHelper()` doesn't depend on its receiver being initialized — that is, it's OK to call `x.initHelper` even if `x` is not initialized — then you can indicate that:

```
class MyClass {
  void initHelper(@UnknownInitialization MyClass this, String param1) { ... }
}
```

If you are using annotations in comments, you would write:

```
class MyClass {
  void initHelper(/*>>>@UnknownInitialization MyClass this,*/ String param1) { ... }
}
```

You are likely to want to annotate `initHelper()` with `@EnsuresNonNull` as well; see Section 3.2.2.

You may get the "call to ... is not allowed on the given receiver" error even if your constructor has already initialized all the fields. For this code:

```
public class MyClass {
  @NonNull Object field;
  public MyClass() {
    field = new Object();
    helperMethod();
  }
  private void helperMethod() {
  }
}
```

the Nullness Checker issues the following warning:

```
MyClass.java:7: error: call to helperMethod() not allowed on the given receiver.
    helperMethod();
                 ^
  found   : @UnderInitialization(MyClass.class) @NonNull MyClass
  required: @Initialized @NonNull MyClass
1 error
```

The reason is that even though the object under construction has had all the fields declared in `MyClass` initialized, there might be a subclass of `MyClass`. Thus, the receiver of `helperMethod` should be declared as `@UnderInitialization(MyClass.class)`, which says that initialization has completed for all the `MyClass` fields but may not have been completed overall. If `helperMethod` had been a public method that could also be called after initialization was actually complete, then the receiver should have type `@UnknownInitialization`, which is the supertype of `@UnknownInitialization` and `@UnderInitialization`.

3.8.5 More details about initialization checking

Suppressing warnings You can suppress warnings related to partially-initialized objects with `@SuppressWarnings("initialization")`.

Checking initialization of all fields, not just `@NonNull` ones When the `-Alint=uninitialized` command-line option is provided, then an object is considered uninitialized until *all* its fields are assigned, not just the `@NonNull` ones. See Section 3.1.

Use of method annotations A method with a non-initialized receiver may assume that a few fields (but not all of them) are non-null, and it sometimes sets some more fields to non-null values. To express these concepts, use the `@RequiresNonNull`, `@EnsuresNonNull`, and `@EnsuresNonNullIf` method annotations; see Section 3.2.2.

Source of the type system The type system enforced by the Initialization Checker is known as "Freedom Before Commitment" [SM11]. Our implementation changes its initialization modifiers ("committed", "free", and "unclassified") to "initialized", "unknown initialization", and "under initialization". Our implementation also has several enhancements. For example, it supports partial initialization (the argument to the `@UnknownInitialization` and `@UnderInitialization` annotations.

3.8.6 Rawness Initialization Checker

The Checker Framework supports two different initialization checkers that are integrated with the Nullness Checker. You can use whichever one you prefer.

One (described in most of Section 3.8) uses the three annotations `@Initialized`, `@UnknownInitialization`, and `@UnderInitialization`. We recommend that you use it.

The other (described here in Section 3.8.6) uses the two annotations `@Raw` and `@NonRaw`. The rawness type system is slightly easier to use but slightly less expressive.

To run the Nullness Checker with the rawness variant of the Initialization Checker, invoke the NullnessRawnessChecker rather than the NullnessChecker; that is, supply the `-processor org.checkerframework.checker.nullness.NullnessRawn` command-line option to javac. Although `@Raw` roughly corresponds to `@UnknownInitialization` and `@NonRaw` roughly corresponds to `@Initialized`, the annotations are not aliased and you must use the ones that correspond to the type-checker that you are running.

An object is *raw* from the time that its constructor starts until its constructor finishes. This is relevant to the Nullness Checker because while the constructor is executing — that is, before initialization completes — a `@NonNull` field may be observed to be null, until that field is set. In particular, the Nullness Checker issues a warning for code like this:

```
public class MyClass {
  private @NonNull Object f;
  public MyClass(int x, int y) {
    // Error because constructor contains no assignment to this.f.
    // By the time the constructor exits, f must be initialized to a non-null value.
  }
  public MyClass(int x) {
    // Error because this.f is accessed before f is initialized.
    // At the beginning of the constructor's execution, accessing this.f
    // yields null, even though field f has a non-null type.
    this.f.toString();
  }
  public MyClass(int x, int y, int z) {
    m();
  }
  public void m() {
```

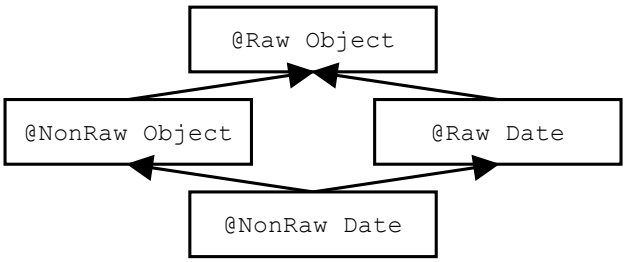


Figure 3.5: Partial type hierarchy for the Rawness Initialization type system.

```
    // Error because this.f is accessed before f is initialized,
    // even though the access is not in a constructor.
    // When m is called from the constructor, accessing f yields null,
    // even though field f has a non-null type.
    this.f.toString();
  }
```

In general, code can depend that field `f` is not `null`, because the field is declared with a `@NonNull` type. However, this guarantee does not hold for a partially-initialized object.

The Nullness Checker uses the `@Raw` annotation to indicate that an object is not yet fully initialized — that is, not all its `@NonNull` fields have been assigned. Rawness is mostly relevant within the constructor, or for references to `this` that escape the constructor (say, by being stored in a field or passed to a method before initialization is complete). Use of rawness annotations is rare in most code.

The most common use for the `@Raw` annotation is for a helper routine that is called by constructor. For example:

```
class MyClass {
  Object field1;
  Object field2;
  Object field3;

  public MyClass(String arg1) {
    this.field1 = arg1;
    init_other_fields();
  }

  // A helper routine that initializes all the fields other than field1
  @EnsuresNonNull({"field2", "field3"})
  private void init_other_fields(@Raw MyClass this) {
    field2 = new Object();
    field3 = new Object();
  }
}
```

For compatibility with Java 6 and 7, you can write the receiver parameter in comments (see Section 27.2.1):

```
  private void init_other_fields(/*>>> @Raw MyClass this*/) {
```

Rawness qualifiers

The rawness hierarchy is shown in Figure 3.5. The rawness hierarchy contains these qualifiers:

Declarations	Expression	Expression's nullness type, or checker error

`class C {`
`  @NonNull Object f;`
`  @Nullable Object g;`
`  ...`
`}`		

`@NonNull @NonRaw C a;`	`a`	`@NonNull`
	`a.f`	`@NonNull`
	`a.g`	`@Nullable`
`@NonNull @Raw C b;`	`b`	`@NonNull`
	`b.f`	`@MonotonicNonNull`
	`b.g`	`@Nullable`
`@Nullable @NonRaw C c;`	`c`	`@Nullable`
	`c.f`	error: deref of nullable
	`c.g`	error: deref of nullable
`@Nullable @Raw C d;`	`d`	`@Nullable`
	`d.f`	error: deref of nullable
	`d.g`	error: deref of nullable

Figure 3.6: Examples of the interaction between nullness and rawness. Declarations are shown at the left for reference, but the focus of the table is the expressions and their nullness type or error.

`@Raw` indicates a type that may contain 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 `@NonNull` fields have been assigned. 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`.

`@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 24.2).

If a reference has `@Raw` type, then all of its `@NonNull` fields are treated as `@MonotonicNonNull`: 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 type annotation? Figure 3.6 contains some examples.

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 `@NonNull` 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.

All `@NonNull` fields must either have a default in the field declaration, or be assigned in the constructor or in a helper method that the constructor calls. If your code initializes (some) fields in a helper method, you will need to annotate the helper method with an annotation such as `@EnsuresNonNull({"field1", "field2"})` for all the fields that the helper method assigns. It's a bit odd, but you use that same annotation, `@EnsuresNonNull`, to indicate that a primitive field has its value set in a helper method, which is relevant when you supply the `-Alint=uninitialized` command-line option (see Section 3.1).

Partial initialization

So far, we have discussed rawness as if it is an all-or-nothing property: an object is fully raw until initialization completes, and then it is no longer raw. The full truth is a bit more complex: during the initialization process, an object can be partially initialized, and as the object's superclass constructors complete, its rawness changes. The Nullness Checker lets you express such properties when necessary.

Consider a simple example:

```
class A {
  Object a;
  A() {
    a = new Object();
  }
}
class B extends A {
  Object b;
  B() {
    super();
    b = new Object();
  }
}
```

Consider what happens during execution of `new B()`.

1. `B`'s constructor begins to execute. At this point, neither the fields of `A` nor those of `B` have been initialized yet.
2. `B`'s constructor calls `A`'s constructor, which begins to execute. No fields of `A` nor of `B` have been initialized yet.
3. `A`'s constructor completes. Now, all the fields of `A` have been initialized, and their invariants (such as that field `a` is non-null) can be depended on. However, because `B`'s constructor has not yet completed executing, the object being constructed is not yet fully initialized. When treated as an `A` (e.g., if only the `A` fields are accessed), the object is initialized (non-raw), but when treated as a `B`, the object is still raw.
4. `B`'s constructor completes. The object is fully initialized (non-raw), if `B`'s constructor was invoked via a `new B()` expression. On the other hand, if there was a `class C extends B { ... }`, and `B`'s constructor had been invoked from that, then the object currently under construction would *not* be fully initialized — it would only be initialized when treated as an `A` or a `B`, but not when treated as a `C`.

At any moment during initialization, the superclasses of a given class can be divided into those that have completed initialization and those that have not yet completed initialization. More precisely, at any moment there is a point in the class hierarchy such that all the classes above that point are fully initialized, and all those below it are not yet initialized. As initialization proceeds, this dividing line between the initialized and raw classes moves down the type hierarchy.

The Nullness Checker lets you indicate where the dividing line is between the initialized and non-initialized classes. You have two equivalent ways to indicate the dividing line: `@Raw` indicates the first class *below* the dividing line, or `@NonRaw(`*`classliteral`*`)` indicates the first class *above* the dividing line.

When you write `@Raw MyClass x;`, that means that variable `x` is initialized for all superclasses of `MyClass`, and (possibly) uninitialized for `MyClass` and all subclasses.

When you write `@NonRaw(Foo.class) MyClass x;`, that means that variable `x` is initialized for `Foo` and all its superclasses, and (possibly) uninitialized for all subclasses of `Foo`.

If `A` is a direct superclass of `B` (as in the example above), then `@Raw A x;` and `@NonRaw(B.class) A x;` are equivalent declarations. Neither one is the same as `@NonRaw A x;`, which indicates that, whatever the actual class of the object that `x` refers to, that object is fully initialized. Since `@NonRaw` (with no argument) is the default, you will rarely see it written.

We can now state a clarification of Section 3.8.6's rule for an object becoming non-raw. As soon as all of the `@NonNull` fields have been initialized, then `this` is treated as `@NonRaw(`*`typeofthis`*`)`, rather than treated as simply `@NonRaw`.

The example above lists 4 moments during construction. At those moments, the type of the object being constructed is:

1. `@Raw Object`
2. `@Raw Object`
3. `@NonRaw(A.class) A`
4. `@NonRaw(B.class) B`

Example As another example, consider the following 12 declarations:

```
@Raw Object rO;
@NonRaw(Object.class) Object nroO;
Object o;

@Raw A rA;
@NonRaw(Object.class) A nroA;  // same as "@Raw A"
@NonRaw(A.class) A nraA;
A a;

@NonRaw(Object.class) B nroB;
@Raw B rB;
@NonRaw(A.class) B nraB;  // same as "@Raw B"
@NonRaw(B.class) B nrbB;
B b;
```

In the following table, the type in cell C1 is a supertype of the type in cell C2 if: C1 is at least as high and at least as far left in the table as C2 is. For example, `nraA`'s type is a supertype of those of `rB`, `nraB`, `nrbB`, `a`, and `b`. (The empty cells on the top row are real types, but are not expressible. The other empty cells are not interesting types.)

`@Raw Object rO;`		
`@NonRaw(Object.class) Object nroO;`	`@Raw A rA;` `@NonRaw(Object.class) A nroA;`	`@NonRaw(Object.class) B nroB;`
	`@NonRaw(A.class) A nraA;`	`@Raw B rB;` `@NonRaw(A.class) B nraB;`
		`@NonRaw(B.class) B nrbB;`
`Object o;`	`A a;`	`B b;`

More details about rawness checking

Suppressing warnings 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!

Checking initialization of all fields, not just `@NonNull` ones When the `-Alint=uninitialized` command-line option is provided, then an object is considered raw until *all* its fields are assigned, not just the `@NonNull` ones. See Section 3.1.

Use of method annotations A method with a raw receiver often assumes that a few fields (but not all of them) are non-null, and sometimes sets some more fields to non-null values. To express these concepts, use the `@RequiresNonNull`, `@EnsuresNonNull`, and `@EnsuresNonNullIf` method annotations; see Section 3.2.2.

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.

Chapter 4

Map Key Checker

The Map Key Checker tracks which values are keys for which maps. If variable `v` has type `@KeyFor("m")...`, then the value of `v` is a key in Map `m`. That is, the expression `m.containsKey(v)` evaluates to `true`.

Section 3.2.4 describes how `@KeyFor` annotations enable the Nullness Checker (Chapter 3, page 24) to treat calls to `Map.get` more precisely by refining its result to `@NonNull` in some cases.

You will not typically run the Map Key Checker. It is automatically run by other checkers, in particular the Nullness Checker.

You can suppress warnings related to map keys with `@SuppressWarnings("keyfor")`; see Chapter 26, page 140.

4.1 Map key annotations

These qualifiers are part of the Map Key type system:

`@KeyFor(String[] maps)` indicates that the value assigned to the annotated variable is a key for at least the given maps.

`@UnknownKeyFor` is used internally by the type system but should never be written by a programmer. It indicates that the value assigned to the annotated variable is not known to be a key for any map. It is the default type qualifier.

`@KeyForBottom` is used internally by the type system but should never be written by a programmer.

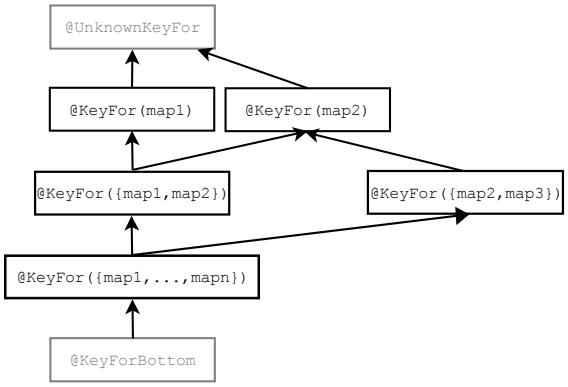


Figure 4.1: The subtyping relationship of the Map Key Checker's qualifiers. `@KeyFor(A)` is a supertype of `@KeyFor(B)` if and only if `A` is a subset of `B`. Qualifiers in gray are used internally by the type system but should never be written by a programmer.

4.2 Examples

The Map Key Checker keeps track of which variables reference keys to which maps. A variable annotated with `@KeyFor(mapSet)` can only contain a value that is a key for all the maps in *mapSet*. For example:

```
Map<String,Date> m, n;
@KeyFor("m") String km;
@KeyFor("n") String kn;
@KeyFor({"m", "n"}) String kmn;
km = kmn;   // OK - a key for maps m and n is also a key for map m
km = kn;    // error: a key for map n is not necessarily a key for map m
```

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

```
Map<String,Object> m;
Collection<@KeyFor("m") String> coll;
coll.add(x);   // error:  coll's element type is @KeyFor("m") String, but x does not have that type
m.put(x, ...);
```

The example type-checks if you reorder the two calls:

```
Map<String,Object> m;
Collection<@KeyFor("m") String> coll;
m.put(x, ...);    // after this statement, x has type @KeyFor("m") String
coll.add(x);      // OK
```

4.3 Inference of @KeyFor annotations

Within a method body, you usually do not have to write `@KeyFor` explicitly, because the checker infers it based on usage patterns. When the Map Key Checker encounters a run-time check for map keys, such as "`if (m.containsKey(k)) ...`", then the Map Key Checker refines the type of `k` to `@KeyFor("m")` within the scope of the test (or until `k` is side-effected within that scope). The Map Key Checker also infers `@KeyFor` annotations based on iteration over a map's key set or calls to `put` or `containsKey`. For more details about type refinement, see Section 25.4.

Suppose we have these declarations:

```
Map<String,Date> m = new Map<String,Date>();
String k = "key";
@KeyFor("m") String km;
```

Ordinarily, the following assignment does not type-check:

```
km = k;   // Error since k is not known to be a key for map m.
```

The following examples show cases where the Map Key Checker infers a `@KeyFor` annotation for variable `k` based on usage patterns, enabling the `km = k` assignment to type-check.

```
m.put(k, ...);
// At this point, the type of k is refined to @KeyFor("m") String.
km = k;   // OK


if (m.containsKey(k)) {
```

```
    // At this point, the type of k is refined to @KeyFor("m") String.
    km = k;   // OK
    ...
}
else {
    km = k;   // Error since k is not known to be a key for map m.
    ...
}
```

The following example shows a case where the Map Key Checker resets its assumption about the type of a field used as a key because that field may have been side-effected.

```
class MyClass {
    private Map<String,Object> m;
    private String k;   // The type of k defaults to @UnknownKeyFor String
    private @KeyFor("m") String km;

    public void myMethod() {
        if (m.containsKey(k)){
            km = k;   // OK: the type of k is refined to @KeyFor("m") String

            sideEffectFreeMethod();
            km = k;   // OK: the type of k is not affected by the method call
                      // and remains @KeyFor("m") String

            otherMethod();
            km = k;   // error: At this point, the type of k is once again
                      // @UnknownKeyFor String, because otherMethod might have
                      // side-effected k such that it is no longer a key for map m.
        }
    }

    @SideEffectFree
    private void sideEffectFreeMethod() { ... }

    private void otherMethod() { ... }
}
```

Chapter 5

Interning Checker

If the Interning Checker issues no errors for a given program, then all reference equality tests (i.e., all uses of "==") are proper; that is, == 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 has two benefits: it can save memory, and it can speed up testing for equality by permitting use of ==.

The Interning Checker prevents two types of errors in your code. First, == 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. Second, 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 these types: `String`, `Boolean`, `Byte`, `Character`, `Integer`, `Short`. 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. The `valueOf` methods in wrapper classes always (`Boolean`, `Byte`) or sometimes (`Character`, `Integer`, `Short`) return an interned result (JLS §5.1.7). 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 errors, you will need to annotate the declarations of any expression used as an argument to ==. Thus, the Interning Checker could also 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`, `@UsesObjectEquals` (which handles a surprising number of cases), and/or `@SuppressWarnings`.

To run the Interning Checker, supply the `-processor org.checkerframework.checker.interning.InterningChecker` command-line option to javac. For examples, see Section 5.4.

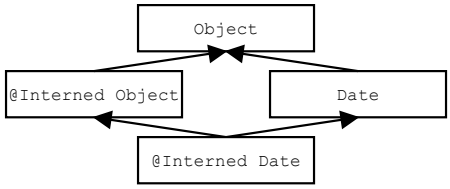


Figure 5.1: Type hierarchy for the Interning type system.

5.1 Interning annotations

These 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 24.2.
`@UsesObjectEquals` is a class (not type) annotation that indicates that this class's `equals` method is the same as that of `Object`. In other words, neither this class nor any of its superclasses overrides the `equals` method. Since `Object.equals` uses reference equality, this means that for such a class, == and `equals` are equivalent, and so the Interning Checker does not issue errors or warnings for either one.

5.2 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.

5.2.1 Implicit qualifiers

As described in Section 25.3, 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`.

5.3 What the Interning Checker checks

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

com.sun.istack.internal.Interned	⇒ org.checkerframework.checker.interning.qual.Interned

Figure 5.2: Correspondence between other interning annotations and the Checker Framework's annotations.

This example shows both sorts of problems:

```
          Date  date;
@Interned Date idate;
...
if (date == idate) { ... }  // error: reference equality test is unsafe
idate = date;               // error: idate'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. For example, to find only the interning errors related to uses of `String`, you can pass `-Acheckclass=java.lang.String`. The Interning Checker always checks all subclasses and superclasses of the given class.

5.3.1 Limitations of the Interning Checker

The Interning Checker conservatively assumes that the `Character`, `Integer`, and `Short` `valueOf` methods return a non-interned value. In fact, these methods sometimes return an interned value and sometimes a non-interned value, depending on the run-time argument (JLS §5.1.7). If you know that the run-time argument to `valueOf` implies that the result is interned, then you will need to suppress an error. (An alternative would be to enhance the Interning Checker to estimate the upper and lower bounds on char, int, and short values so that it can more precisely determine whether the result of a given `valueOf` call is interned.)

5.4 Examples

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

```
javac -processor org.checkerframework.checker.interning.InterningChecker examples/InterningExample.java
```

Compilation will complete without errors or warnings.

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

```
javac -processor org.checkerframework.checker.interning.InterningChecker examples/InterningExampleWithWa
```

The compiler will issue an error regarding violation of the semantics of `@Interned`.

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

5.5 Other interning annotations

The Checker Framework's interning annotations are similar to annotations used elsewhere.

If your code is already annotated with a different interning annotation, you can reuse that effort. The Checker Framework comes with cleanroom re-implementations of annotations from other tools. It treats them exactly as if you had written the corresponding annotation from the Interning Checker, as described in Figure 5.2.

Alternately, the Checker Framework can process those other annotations (as well as its own, if they also appear in your program). The Checker Framework has its own definition of the annotations on the left side of Figure 5.2, so that they can be used as type qualifiers. The Checker Framework interprets them according to the right side of Figure 5.2.

Chapter 6

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 annotated with `@GuardedBy`.

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 org.checkerframework.checker.lock.LockChecker` command-line option to javac.

6.1 Lock annotations

Summary of declaration annotations used by the Lock Checker.

Type annotation	**Indicates**
@GuardedBy(String lock)	Fields/variables only accessible after acquiring the given lock.
Declaration annotation	**Indicates**
@Holding(String[] locks)	Locks that must be held before the method is called.
@EnsuresLockHeld(String[] expressions)	Locks guaranteed to be held when the method returns.
@EnsuresLockHeldIf(String[] expr, boolean result)	Locks guaranteed to be held when the method returns the given result.
@LockingFree	The method does not make any use of locks or synchronization.

6.1.1 Type annotations for objects protected by locks

`@GuardedBy`(String lock) indicates a type whose value may be accessed only when the given lock is held. See the Javadoc for `GuardedBy` for an explanation of the argument and other details. 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.

6.1.2 Lock method annotations

The Lock Checker supports several annotations that specify method behavior. These are declaration annotations, not type annotations: they apply to the method itself rather than to some particular type.

`@EnsuresLockHeld`(String[] expressions)

`@EnsuresLockHeldIf`(String[] expressions, boolean result) indicate a method postcondition. With `@EnsuresLockHeld`, the given expressions are known to be objects used as locks and are known to be in a locked state after the method returns; this is useful for annotating a method that takes a lock. With `@EnsuresLockHeldIf`, if the annotated

method returns the given boolean value (true or false), the given expressions are known to be objects used as locks and are known to be in a locked state after the method returns; this is useful for annotating a method that conditionally takes a lock. See Section 6.2.2 for examples.

`@LockingFree` indicates that the method does not use synchronization/locking, directly or indirectly. This is used to facilitate dataflow analysis and is less restrictive than `@SideEffectFree`. It is especially useful for annotating library methods, including JDK methods. Since `@SideEffectFree` implies `@LockingFree`, if both are applicable then you should only write `@SideEffectFree`.

It is critical not to use this annotation for any method that uses synchronization/locking, directly or indirectly. This is because even methods that are guaranteed to release all locks they acquire could cause deadlocks. Although the Lock Checker currently does not aid with deadlock detection, this annotation must be used in anticipation that the Lock Checker eventually could.

6.1.3 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-aquire 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.

6.2 Examples

6.2.1 Examples of @GuardedBy and @Holding

The most common use of `@GuardedBy` is to annotate a field declaration type. However, other uses of `@GuardedBy` are possible.

Return types A return type 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;
x.toString(); // ILLEGAL because the lock is not held
synchronized(MyClass.myLock) {
  y.toString();  // OK: the lock is held
}
```

Formal parameters A parameter type may be annotated with `@GuardedBy`, which indicates that the method body must acquire the lock before accessing the parameter. A client may pass a non-`@GuardedBy` reference as an argument, since it is legal to access such a reference after the lock is acquired.

```
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) {
  helper1(c); // OK: passing a subtype in place of a the @GuardedBy supertype
  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
  }
}
```

6.2.2 Examples of @EnsuresLockHeld and @EnsuresLockHeldIf

`@EnsuresLockHeld` and `@EnsuresLockHeldIf` are primarily intended for annotating JDK locking methods, as in:

```
package java.util.concurrent.locks;

class ReentrantLock {

    @EnsuresLockHeld("this")
    public void lock();

    @EnsuresLockHeldIf (expression="this", result=true)
    public boolean tryLock();

[...]
}
```

They can also be used to annotate user methods, particularly for higher-level lock constructs such as a Monitor, as in this simplified example:

```
public class Monitor {

    private ReentrantLock lock; // Initialized in the constructor

[...]

    @EnsuresLockHeld("lock")
    public void enter() {
           lock.lock();
    }

[...]
}
```

6.2.3 Example of @LockingFree

`@LockingFree` is useful when a method does not make any use of synchronization or locks but causes other side effects (hence `@SideEffectFree` is not appropriate). `@SideEffectFree` implies `@LockingFree`, therefore if both are applicable, you should only write `@SideEffectFree`.

```
private Object myField;
private ReentrantLock lock; // Initialized in the constructor
private @GuardedBy("lock") Object x; // Initialized in the constructor

[...]

@LockingFree
// This method does not use locks or synchronization but cannot
// be annotated as @SideEffectFree since it alters myField.
void myMethod() {
    myField = new Object();
}

@SideEffectFree
int mySideEffectFreeMethod() {
    return 0;
```

net.jcip.annotations.GuardedBy
javax.annotation.concurrent.GuardedBy

$\Rightarrow$ org.checkerframework.checker.lock.qual.GuardedBy

Figure 6.1: Correspondence between other lock annotations and the Checker Framework's annotations.

```
}

void myUnlockingMethod() {
    lock.unlock();
}

void myUnannotatedEmptyMethod() {
}

void myOtherMethod() {
    if (lock.tryLock()) {
        x.toString(); // OK: the lock is held
        myMethod();
        x.toString(); // OK: the lock is still known to be held since myMethod is locking-free
        mySideEffectFreeMethod();
        x.toString(); // OK: the lock is still known to be held since mySideEffectFreeMethod
                      // is side-effect-free
        myUnlockingMethod();
        x.toString(); // ILLEGAL: myLockingMethod is not locking-free
    }
    if (lock.tryLock()) {
        x.toString(); // OK: the lock is held
        myUnannotatedEmptyMethod();
        x.toString(); // ILLEGAL: even though myUnannotatedEmptyMethod is empty, since it is
                      // not annotated with @LockingFree, the Lock Checker no longer knows
                      // the state of the lock.
        if (lock.isHeldByCurrentThread()) {
            x.toString(); // OK: the lock is known to be held
        }
    }
}
```

6.3 Other lock annotations

The Checker Framework's lock annotations are similar to annotations used elsewhere.

If your code is already annotated with a different lock annotation, you can reuse that effort. The Checker Framework comes with cleanroom re-implementations of annotations from other tools. It treats them exactly as if you had written the corresponding annotation from the Lock Checker, as described in Figure 6.1.

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 6.1, so that they can be used as type annotations. The Checker Framework interprets them according to the right side of Figure 6.1.

6.3.1 Relationship to annotations in *Java Concurrency in Practice*

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

The Lock Checker renames the method annotation to `@Holding`, and it generalizes the `@GuardedBy` annotation into a type annotation 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 meanings of the two annotations, and resolves ambiguity when `@GuardedBy` is written in a location that might apply to either the method or the return type.

(The JCIP book gives some rationales for reusing the annotation name for two purposes. One rationale 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.)

6.4 Possible extensions

The Lock Checker validates some uses of locks, but not all. It would be possible to enrich it with additional annotations. This would increase the programmer annotation burden, but would provide additional guarantees.

Lock ordering: Specify that one lock must be acquired before or after another, or specify a global ordering for all locks. This would prevent deadlock.

Not-holding: Specify that a method must not be called if any of the listed locks are held.

These features are supported by Clang's thread-safety analysis.

6.5 A note on Lock Checker internals

The following type qualifiers are inferred and used internally by the Lock Checker and should never need to be written by the programmer. They are presented here for reference on how the Lock Checker works and to help understand warnings produced by the Lock Checker. You may skip this section if you are not seeing a warning mentioning `@LockHeld` or `@LockPossiblyHeld`.

These type qualifiers are used on the types of the objects that will be used as locks to protect other objects. The Lock Checker uses them to track the current state of locks at a given point in the code.

`@LockPossiblyHeld` indicates a type that may be used as a lock to protect a field/variable (i.e. an object of this type may be used as the expression in a `@GuardedBy` annotation) and the lock may or may not be currently held. Since any object can potentially be used as a lock, it in fact applies to all non-primitive types. This is the default type qualifier in the hierarchy and it is the top type.

`@LockHeld` indicates a type that may be used as a lock to protect a field/variable, and is currently in a locked state on the current thread. It is a subtype of `@LockPossiblyHeld` and is the bottom type.

Chapter 7

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. Use of an int instead of an object can reduce code size, memory requirements, and run time.

In cases when code has to use the fake enum pattern, the Fake Enum Checker, or 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.

Figure 7.1 shows part of the type hierarchy for the Fenum type system.

7.1 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`:

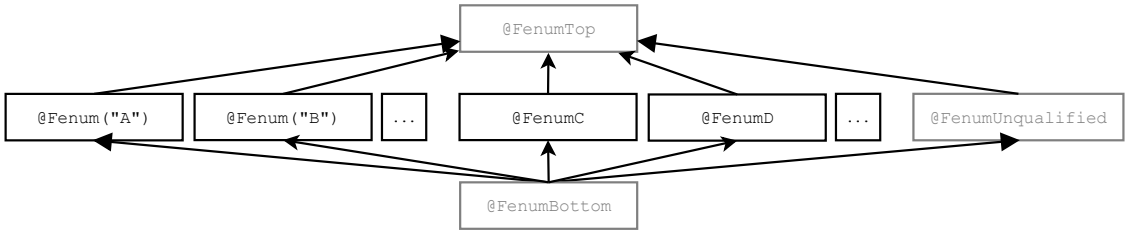


Figure 7.1: Partial type hierarchy for the Fenum type system. There are two forms of fake enumeration annotations — above, illustrated by `@Fenum("A")` and `@FenumC`. See Section 7.1 for descriptions of how to introduce both types of fenums. The type qualifiers in gray (`@FenumTop`, `@FenumUnqualified`, and `@FenumBottom`) should never be written in source code; they are used internally by the type system.

```
package myproject.qual;

import java.lang.annotation.*;
import org.checkerframework.framework.qual.SubtypeOf;
import org.checkerframework.framework.qual.TypeQualifier;

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

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

2. Use the provided `@Fenum` annotation, which 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.

7.2 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 24) can prevent failure to initialize a reference variable.)

7.3 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 org.checkerframework.checker.fenum.FenumChecker
      -Aquals=myproject.qual.MyFenum MyFile.java ...
```

- If your code uses the `@Fenum` annotation, you do not need the `-Aquals` option:

```
javac -processor org.checkerframework.checker.fenum.FenumChecker MyFile.java ...
```

7.4 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 a `@SuppressWarnings` annotation to either the field or class declaration, for example:

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

7.5 Example

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

```
@SuppressWarnings("fenum:assignment.type.incompatible")   // initialization of fake enums
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
  }
 }
```

7.6 References

- Java Language Specification on enums:
  https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.9
- Tutorial trail on enums:
  https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html
- Typesafe enum pattern:
  http://www.oracle.com/technetwork/java/page1-139488.html
- Java Tip 122: Beware of Java typesafe enumerations:
  http://www.javaworld.com/article/2077487/core-java/java-tip-122--beware-of-java-typesafe-enumerations
  html

Chapter 8

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 TaintingChecker` command-line option to javac.

8.1 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 24.2).

8.2 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 25.3.1 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 28 describes how to annotate external libraries):

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

8.3 `@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 Subtyping Checker (Chapter 22). 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 22.2), 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 29 for more details.

8.3.1 Qualifier Parameters

The Tainting Checker supports qualifier parameters. See section 24.3 for more details on qualifier parameters.

The qualifier parameter system currently (as of February 2015) incurs a 50% performance penalty. If this is unacceptable you can run the original Tainting Checker by passing `-processor org.checkerframework.checker.tainting.classic.Tainti` command-line option to javac.

Chapter 9

Regex Checker for regular expression syntax

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

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. Regular expression in Java also have capturing groups, which are delimited by parentheses and allow for extraction from text.

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.

For further details, including case studies, see a paper about the Regex Checker [SDE12].

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

9.1 Regex annotations

These qualifiers make up the Regex type system:

`@Regex` indicates valid regular expression `String`s. This qualifier takes an optional parameter of at the least the number of capturing groups in the regular expression. If not provided, the parameter defaults to 0.

`@PolyRegex` indicates qualifier polymorphism. For a description of `@PolyRegex`, see Section 24.2.

The subtyping hierarchy of the Regex Checker's qualifiers is shown in Figure 9.1.

9.2 Annotating your code with `@Regex`

9.2.1 Implicit qualifiers

As described in Section 25.3, the Regex Checker adds implicit qualifiers, reducing the number of annotations that must appear in your code. The checker implicitly adds the `Regex` qualifier with the parameter set to the correct number of capturing groups to any `String` literal that is a valid regex. The Regex Checker allows the `null` literal to be assigned to any type qualified with the `Regex` qualifier.

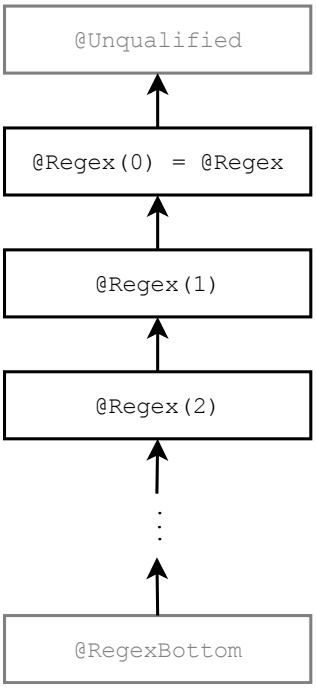


Figure 9.1: The subtyping relationship of the Regex Checker's qualifiers. Because the parameter to a `@Regex` qualifier is at least the number of capturing groups in a regular expression, a `@Regex` qualifier with more capturing groups is a subtype of a `@Regex` qualifier with fewer capturing groups. Qualifiers in gray are used internally by the type system but should never be written by a programmer.

```
public @Regex String parenthesize(@Regex String regex) {
    return "(" + regex + ")"; // Even though the parentheses are not @Regex Strings,
                              // the whole expression is a @Regex String
}
```

Figure 9.2: An example of the Regex Checker's support for concatenation of non-regular-expression Strings to produce valid regular expression Strings.

9.2.2 Capturing groups

The Regex Checker validates that a legal capturing group number is passed to `Matcher`'s `group`, `start` and `end` methods. To do this, the type of `Matcher` must be qualified with a `@Regex` annotation with the number of capturing groups in the regular expression. This is handled implicitly by the Regex Checker for local variables (see Section 25.4), but you may need to add `@Regex` annotations with a capturing group count to `Pattern` and `Matcher` fields and parameters.

9.2.3 Concatenation of partial regular expressions

In general, concatenating a non-regular-expression String with any other string yields a non-regular-expression String. The Regex Checker can sometimes determine that concatenation of non-regular-expression Strings will produce valid regular expression Strings. For an example see Figure 9.2.

```
String regex = getRegexFromUser();
if (! RegexUtil.isRegex(regex)) {
   throw new RuntimeException("Error parsing regex " + regex, RegexUtil.regexException(regex));
}
Pattern p = Pattern.compile(regex);
```

Figure 9.3: Example use of `RegexUtil` methods.

9.2.4 Testing whether a string is a regular expression

Sometimes, the Regex Checker cannot infer whether a particular expression is a regular expression — and sometimes your code cannot either! In these cases, you can use the `isRegex` method to perform such a test, and other helper methods to provide useful error messages. A common use is for user-provided regular expressions (such as ones passed on the command-line). Figure 9.3 gives an example of the intended use of the `RegexUtil` methods.

`RegexUtil.isRegex` returns `true` if its argument is a valid regular expression.

`RegexUtil.regexError` returns a `String` error message if its argument is not a valid regular expression, or `null` if its argument is a valid regular expression.

`RegexUtil.regexException` returns the `PatternSyntaxException` that `Pattern.compile(String)` throws when compiling an invalid regular expression. It returns `null` if its argument is a valid regular expression.

An additional version of each of these methods is also provided that takes an additional group count parameter. The `RegexUtil.isRegex` method verifies that the argument has at least the given number of groups. The `RegexUtil.regexError` and `RegexUtil.regexException` methods return a `String` error message and `PatternSyntaxException`, respectively, detailing why the given String is not a syntactically valid regular expression with at least the given number of capturing groups.

If you detect that a `String` is not a valid regular expression but would like to report the error higher up the call stack (potentially where you can provide a more detailed error message) you can throw a `RegexUtil.CheckedPatternSyntaxException`. This exception is functionally the same as a `PatternSyntaxException` except it is checked to guarantee that the error will be handled up the call stack. For more details, see the Javadoc for `RegexUtil.CheckedPatternSyntaxException`.

A potential disadvantage of using the `RegexUtil` class is that your code becomes dependent on the Checker Framework at run time as well as at compile time. You can avoid this by adding the Checker Framework to your project, or by copying the `RegexUtil` class into your own code.

9.2.5 Qualifier Parameters

The Regex Checker supports qualifier parameters. See section 24.3 for more details on qualifier parameters.

The qualifier parameter system currently (as of February 2015) incurs a 50% performance penalty. If this is unacceptable you can run the original Regex Checker by passing `-processor org.checkerframework.checker.regex.classic.RegexClass` as a command-line option to javac.

9.2.6 Suppressing warnings

If you are positive that a particular string that is being used as a regular expression is syntactically valid, but the Regex Checker cannot conclude this and issues a warning about possible use of an invalid regular expression, then you can use the `RegexUtil.asRegex` method to suppress the warning.

You can think of this method as a cast: it returns its argument unchanged, but with the type `@Regex String` if it is a valid regular expression. It throws an error if its argument is not a valid regular expression, but you should only use it when you are sure it will not throw an error.

There is an additional `RegexUtil.asRegex` method that takes a capturing group parameter. This method works the same as described above, but returns a `@Regex String` with the parameter on the annotation set to the value of the capturing group parameter passed to the method.

The use case shown in Figure 9.3 should support most cases so the `asRegex` method should be used rarely.

Chapter 10

Format String Checker

The Format String Checker prevents use of incorrect format strings in format methods such as `System.out.printf` and `String.format`.

The Format String Checker warns you if you write an invalid format string, and it warns you if the other arguments are not consistent with the format string (in number of arguments or in their types). Here are examples of errors that the Format String Checker detects at compile time. Section 10.3 provides more details.

```
String.format("%y", 7);           // error: invalid format string

String.format("%d", "a string");  // error: invalid argument type for %d

String.format("%d %s", 7);        // error: missing argument for %s
String.format("%d", 7, 3);        // warning: unused argument 3
String.format("{0}", 7);          // warning: unused argument 7, because {0} is wrong syntax
```

To run the Format String Checker, supply the `-processor org.checkerframework.checker.formatter.FormatterChecker` command-line option to javac.

10.1 Formatting terminology

Printf-style formatting takes as an argument a *format string* and a list of arguments. It produces a new string in which each *format specifier* has been replaced by the corresponding argument. The format specifier determines how the format argument is converted to a string. A format specifier is introduced by a `%` character. For example, `String.format("The %s is %d.","answer",42)` yields `"The answer is 42."`. `"The %s is %d."` is the format string, `"%s"` and `"%d"` are the format specifiers; `"answer"` and `42` are format arguments.

10.2 Format String Checker annotations

The `@Format` qualifier on a string type indicates a *valid* format string. The JDK documentation for the `Formatter` class explains the requirements for a valid format string. A programmer rarely writes the `@Format` annotation, as it is inferred for string literals. A programmer may need to write it on fields and on method signatures.

The `@Format` qualifier is parameterized with a list of conversion categories that impose restrictions on the format arguments. Conversion categories are explained in more detail in Section 10.2.1. The type qualifier for `"%d %f"` is for example `@Format({INT, FLOAT})`.

Consider the below `printFloatAndInt` method. Its parameter must be a format string that can be used in a format method, where the first format argument is "float-like" and the second format argument is "integer-like". The type of its parameter, `@Format({FLOAT, INT}) String`, expresses that contract.

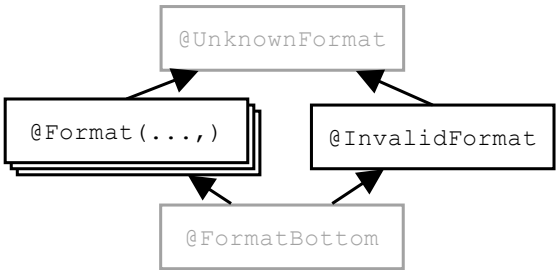


Figure 10.1: The Format String Checker type qualifier hierarchy. The figure does not show the subtyping rules among different `@Format(...)` qualifiers; see Section 10.2.1.

```
void printFloatAndInt(@Format({FLOAT, INT}) String fs) {
    System.out.printf(fs, 3.1415, 42);
}

printFloatAndInt("Float %f, Number %d");  // OK
printFloatAndInt("Float %f");             // error
```

Figure 10.1 shows all the type qualifiers. The annotations other than `@Format` are only used internally and cannot be written in your code. `@InvalidFormat` indicates an invalid format string — that is, a string that cannot be used as a format string. For example, the type of `"%y"` is `@InvalidFormat String`. `@FormatBottom` is the type of the `null` literal. `@Unqualified` is the default that is applied to strings that are not literals and on which the user has not written a `@Format` annotation.

10.2.1 Conversion Categories

Given a format specifier, only certain format arguments are compatible with it, depending on its "conversion" — its last, or last two, characters. For example, in the format specifier `"%d"`, the conversion `d` restricts the corresponding format argument to be "integer-like":

```
String.format("%d", 5);         // OK
String.format("%d", "hello");   // error
```

Many conversions enforce the same restrictions. A set of restrictions is represented as a *conversion category*. The "integer like" restriction is for example the conversion category `INT`. The following conversion categories are defined in the `ConversionCategory` enumeration:

`GENERAL` imposes no restrictions on a format argument's type. Applicable for conversions b, B, h, H, s, S.

`CHAR` requires that a format argument represents a Unicode character. Specifically, `char`, `Character`, `byte`, `Byte`, `short`, and `Short` are allowed. `int` or `Integer` are allowed if `Character.isValidCodePoint(argument)` would return `true` for the format argument. (The Format String Checker permits any `int` or `Integer` without issuing a warning or error — see Section 10.3.2.) Applicable for conversions c, C.

`INT` requires that a format argument represents an integral type. Specifically, `byte`, `Byte`, `short`, `Short`, `int` and `Integer`, `long`, `Long`, and `BigInteger` are allowed. Applicable for conversions d, o, x, X.

`FLOAT` requires that a format argument represents a floating-point type. Specifically, `float`, `Float`, `double`, `Double`, and `BigDecimal` are allowed. Surprisingly, integer values are not allowed. Applicable for conversions e, E, f, g, G, a, A.

`TIME` requires that a format argument represents a date or time. Specifically, `long`, `Long`, `Calendar`, and `Date` are allowed. Applicable for conversions t, T.

`UNUSED` imposes no restrictions on a format argument. This is the case if a format argument is not used as replacement for any format specifier. `"%2$s"` for example ignores the first format argument.

Further, all conversion categories accept `null`.

The same format argument may serve as a replacement for multiple format specifiers. Until now, we have assumed that the format specifiers simply consume format arguments left to right. But there are two other ways for a format specifier to select a format argument:

- *n*`$` specifies a one-based index *n*. In the format string `"%2$s"`, the format specifier selects the second format argument.
- The `<` *flag* references the format argument that was used by the previous format specifier. In the format string `"%d %<d"` for example, both format specifiers select the first format argument.

In the following example, the format argument must be compatible with both conversion categories, and can therefore be neither a `Character` nor a `long`.

```
format("Char %1$c, Int %1$d", (int)42);            // OK
format("Char %1$c, Int %1$d", new Character(42));  // error
format("Char %1$c, Int %1$d", (long)42);           // error
```

Only three additional conversion categories are needed represent all possible intersections of previously-mentioned conversion categories:

`NULL` is used if no object of any type can be passed as parameter. In this case, the only legal value is `null`. The format string `"%1$f %1$c"`, for example requires that the first format argument be `null`. Passing a value such as `4` or `4.2` would lead to an exception.

`CHAR_AND_INT` is used if a format argument is restricted by a `CHAR` and a `INT` conversion category (`CHAR` ∩ `INT`).

`INT_AND_TIME` is used if a format argument is restricted by an `INT` and a `TIME` conversion category (`INT` ∩ `TIME`).

All other intersections lead to already existing conversion categories. For example, `GENERAL` ∩ `CHAR` = `CHAR` and `UNUSED` ∩ `GENERAL` = `GENERAL`.

Figure 10.2 summarizes the subset relationship among all conversion categories.

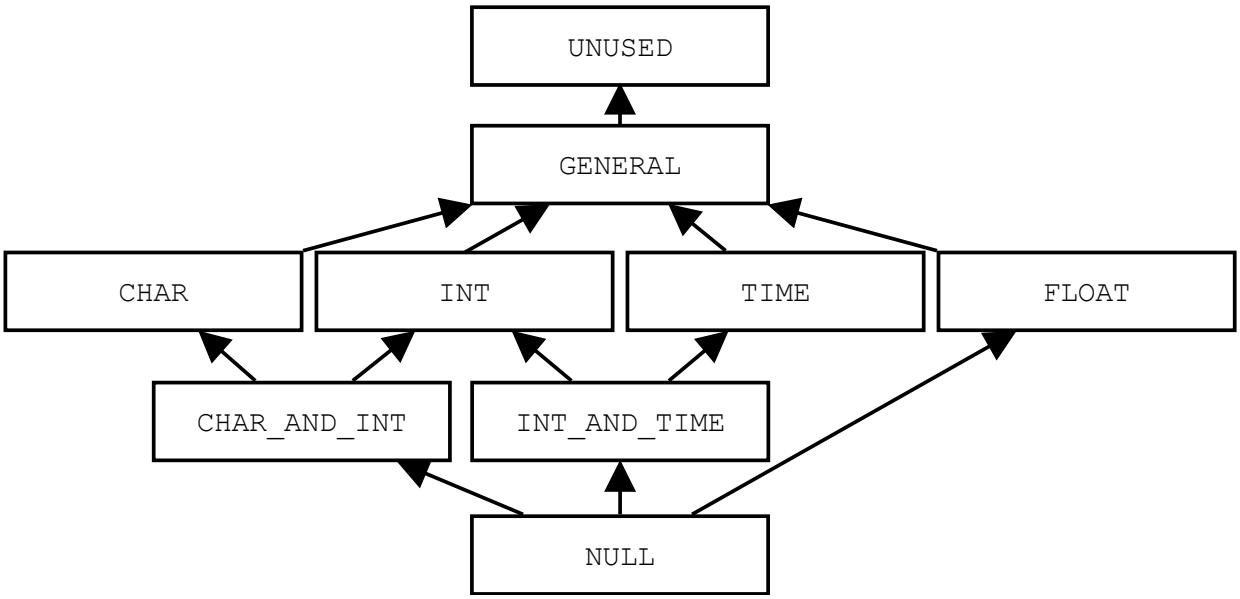


Figure 10.2: The subset relationship among conversion categories.

Here are the subtyping rules among different `@Format` qualifiers. It is legal to:

- use a format string with a weaker (less restrictive) conversion category than required.

- use a format string with fewer format specifiers than required, but a warning is issued.

The following example shows the subtyping rules in action:

```
@Format({FLOAT, INT})
String f;

f = "%f %d";       // Ok
f = "%s %d";       // OK, %s is weaker than %f
f = "%f";          // warning: last argument is ignored
f = "%f %d %s";    // error: too many arguments
f = "%d %d";       // error: %d is not weaker than %f

String.format(f, 0.8, 42);
```

10.3 What the Format String Checker checks

If the Format String Checker issues no errors, it provides the following guarantees:

1. The following guarantees hold for every format method invocation:
   (a) The format method's first parameter (or second if a `Locale` is provided) is a valid format string (or `null`).
   (b) A warning is issued if one of the format string's conversion categories is `UNUSED`.
   (c) None of the format string's conversion categories is `NULL`.
2. If the format arguments are passed to the format method as varargs, the Format String Checker guarantees the following additional properties:
   (a) No fewer format arguments are passed than required by the format string.
   (b) A warning is issued if more format arguments are passed than required by the format string.
   (c) Every format argument's type satisfies its conversion category's restrictions.
3. If the format arguments are passed to the format method as array, a warning is issued by the Format String Checker.

Following are examples for every guarantee:

```
String.format("%d", 42);                      // OK
String.format(Locale.GERMAN, "%d", 42);       // OK
String.format(new Object());                  // error (1a)
String.format("%y");                          // error (1a)
String.format("%2$s", "unused", "used");      // warning (1b)
String.format("%1$d %1$f", 5.5);              // error (1c)
String.format("%1$d %1$f %d", null, 6);       // error (1c)
String.format("%s");                          // error (2a)
String.format("%s", "used", "ignored");       // warning (2b)
String.format("%c",4.2);                      // error (2c)
String.format("%c", (String)null);            // error (2c)
String.format("%1$d %1$f", new Object[]{1});  // warning (3)
String.format("%s", new Object[]{"hello"});   // warning (3)
```

10.3.1 Possible false alarms

There are three cases in which the Format String Checker may issue a warning or error, even though the code cannot fail at run time. (These are in addition to the general conservatism of a type system: code may be correct because of

application invariants that are not captured by the type system.) In each of these cases, you can rewrite the code, or you can manually check it and write a `@SuppressWarnings` annotation if you can reason that the code is correct.

Case 1b: Unused format arguments. It is legal to provide more arguments than are required by the format string; Java ignores the extras. However, this is an uncommon case. In practice, a mismatch between the number of format specifiers and the number of format arguments is usually an error.

Case 1c: Format arguments that can only be `null`. It is legal to write a format string that permits only null arguments and throws an exception for any other argument. An example is `String.format("%1$d %1$f", null)`. The Format String Checker forbids such a format string. If you should ever need such a format string, simply replace the problematic format specifier with `"null"`. For example, you would replace the call above by `String.format("null null")`.

Case 3: Array format arguments. The Format String Checker performs no analysis of arrays, only of varargs invocations. It is better style to use varargs when possible.

10.3.2 Possible missed alarms

The Format String Checker helps prevent bugs by detecting, at compile time, which invocations of format methods will fail. While the Format String Checker finds most of these invocations, there are cases in which a format method call will fail even though the Format String Checker issued neither errors nor warnings. These cases are:

1. The format string is `null`. Use the Nullness Checker to prevent this.
2. A format argument's `toString` method throws an exception.
3. A format argument implements the `Formattable` interface and throws an exception in the `formatTo` method.
4. A format argument's conversion category is `CHAR` or `CHAR_AND_INT`, and the passed value is an `int` or `Integer`, and `Character.isValidCodePoint(argument)` returns `false`.

The following examples illustrate these limitations:

```
class A {
    public String toString() {
        throw new Error();
    }
}

class B implements Formattable {
    public void formatTo(Formatter fmt, int f,
            int width, int precision) {
        throw new Error();
    }
}

// The checker issues no errors or warnings for the
// following illegal invocations of format methods.
String.format(null);          // NullPointerException (1)
String.format("%s", new A()); // Error (2)
String.format("%s", new B()); // Error (3)
String.format("%c", (int)-1); // IllegalFormatCodePointException (4)
```

10.4 Implicit qualifiers

As described in Section 25.3, the Format String Checker adds implicit qualifiers, reducing the number of annotations that must appear in your code. The checker implicitly adds the `@Format` qualifier with the appropriate conversion categories to any String literal that is a valid format string.

10.5 FormatMethod

Your project may contain methods that forward their arguments to a format method. Consider for example the following `log` method:

```
@FormatMethod
void log(String format, Object... args) {
    if (enabled) {
        logfile.print(indent_str);
        logfile.printf(format , args);
    }
}
```

By attaching a `@FormatMethod` annotation to such a method, you can instruct the Format String Checker to check every invocation of the method. This check is analogous to the check done on every invocation of built in format methods like `String.format`.

10.6 Testing whether a format string is valid

The Format String Checker automatically determines whether each `String` literal is a valid format string or not. When a string is computed or is obtained from an external resource, then the string must be trusted or tested.

One way to test a string is to call the `FormatUtil.asFormat` method to check whether the format string is valid and its format specifiers match certain conversion categories. If this is not the case, `asFormat` raises an exception. Your code should catch this exception and handle it gracefully.

The following code examples may fail at run time, and therefore they do not type check. The type-checking errors are indicated by comments.

```
Scanner s = new Scanner(System.in);
String fs = s.next();
System.out.printf(fs, "hello", 1337);          // error: fs is not known to be a format string

Scanner s = new Scanner(System.in);
@Format({GENERAL, INT}) String fs = s.next();  // error: fs is not known to have the given type
System.out.printf(fs, "hello", 1337);          // OK
```

The following variant does not throw a run-time error, and therefore passes the type-checker:

```
Scanner s = new Scanner(System.in);
String format = s.next()
try {
    format = FormatUtil.asFormat(format, GENERAL, INT);
} catch (IllegalFormatException e) {
    // Replace this by your own error handling.
    System.err.println("The user entered the following invalid format string: " + format);
    System.exit(2);
}
// fs is now known to be of type: @Format({GENERAL, INT}) String
System.out.printf(format,"hello",1337);
```

A potential disadvantage of using the `FormatUtil` class is that your code becomes dependent on the Checker Framework at run time as well as at compile time. You can avoid this by adding the Checker Framework to your project, or by copying the `FormatUtil` class into your own code.

Chapter 11

Internationalization Format String Checker (I18n Format String Checker)

The Internationalization Format String Checker, or I18n Format String Checker, prevents use of incorrect i18n format strings.

If the I18n Format String Checker issues no warnings or errors, then `MessageFormat.format` will raise no error at run time. "I18n" is short for "internationalization" because there are 18 characters between the "i" and the "n".

Here are the examples of errors that the I18n Format Checker detects at compile time.

```
// Warning: the second argument is missing.
MessageFormat.format("{0} {1}", 3.1415);
// String argument cannot be formatted as Time type.
MessageFormat.format("{0, time}", "my string");
// Invalid format string: unknown format type: thyme.
MessageFormat.format("{0, thyme}", new Date());
// Invalid format string: missing the right brace.
MessageFormat.format("{0", new Date());
// Invalid format string: the argument index is not an integer.
MessageFormat.format("{0.2, time}", new Date());
// Invalid format string: "#.#.#" subformat is invalid.
MessageFormat.format("{0, number, #.#.#}", 3.1415);
```

For instructions on how to run the Internationalization Format String Checker, see Section 11.5.

The Internationalization Checker or I18n Checker (Chapter 12.2, page 78) has a different purpose. It verifies that your code is properly internationalized: any user-visible text should be obtained from a localization resource and all keys exist in that resource.

11.1 Internationalization Format String Checker annotations

The `MessageFormat` documentation specifies the syntax of the i18n format string.

These are the qualifiers that make up the I18n Format String type system. Figure 11.1 shows their subtyping relationships.

`@I18nFormat` represents a valid i18n format string. For example, `@I18nFormat({GENERAL, NUMBER, UNUSED, DATE})` is a legal type for `"{0}{1, number} {3, date}"`, indicating that when the format string is used, the first argument should be of `GENERAL` conversion category, the second argument should be of `NUMBER` conversion category, and so on. Conversion categories such as `GENERAL` are described in Section 11.2.

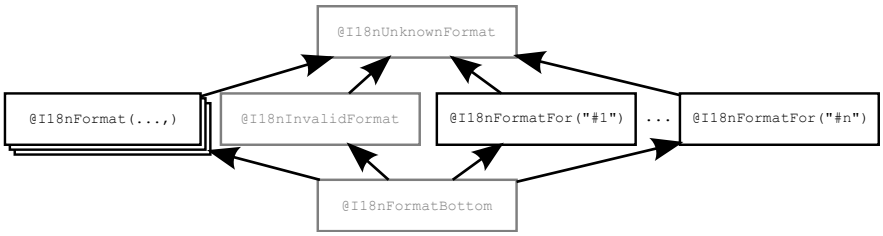


Figure 11.1: The Internationalization Format String Checker type qualifier hierarchy. The figure does not show the subtyping rules among different `@I18nFormat(...)` qualifiers; see Section 11.2. All `@I18nFormatFor` annotations are unrelated by subtyping. The qualifiers in gray are used internally by the checker and should never be written by a programmer.

`@I18nFormatFor` indicates that the qualified type is a valid i18n format string for use with some array of values. For example, `@I18nFormatFor("#2")` indicates that the string can be used to format the contents of the second parameter array. The argument is a Java expression whose syntax is explained in Section 25.5. An example of its use is:

```
static void method(@I18nFormatFor("#2") String format, Object... args) {
  // the body may use the parameters like this:
  MessageFormat.format(format, args);
}

method("{0, number} {1}", 3.1415, "A string");  // OK
// error: The string "hello" cannot be formatted as a Number.
method("{0, number} {1}", "hello", "goodbye");
```

`@I18nInvalidFormat` represents an invalid i18n format string. Programmers are not allowed to write this annotation. It is only used internally by the type checker.

`@I18nUnknownFormat` represents any string. The string might or might not be a valid i18n format string. Programmers are not allowed to write this annotation.

`@I18nFormatBottom` indicates that the value is definitely `null`. Programmers are not allowed to write this annotation.

11.2 Conversion categories

In a message string, the optional second element within the curly braces is called a *format type* and must be one of `number`, `date`, `time`, and `choice`. These four format types correspoond to different conversion categories. `date` and `time` correspond to *DATE* in the conversion categories figure. `choice` corresponds to *NUMBER*. The format type restricts what arguments are legal. For example, a date argument is not compatible with the `number` format type, i.e., `MessageFormat.format("{0, number}", new Date())` will throw an exception.

The I18n Checker represents the possible arguments via *conversion categories*. A conversion category defines a set of restrictions or a subtyping rule.

Figure 11.2 summarizes the subset relationship among all conversion categories.

Here are the subtyping rules among different `@I18nFormat` qualifiers. It is legal to:

- use a format string with a weaker (less restrictive) conversion category than required.
- use a format string with fewer format specifiers than required, but a warning is issued.

The following example shows the subtyping rules in action:

```
@I18nFormat({NUMBER, NUMBER}) String format;
// OK.
```

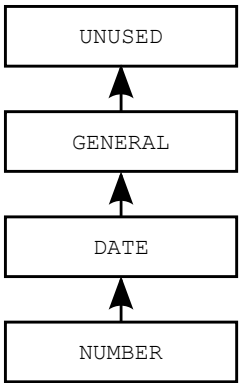


Figure 11.2: The subset relationship among i18n conversion categories.

```
format = "{0, number, #.#} {1, number}";
// OK, GENERAL is weaker (less restrictive) than NUMBER.
format = "{0, number} {1}";
// Error, the right-hand-side is stronger (more restrictive) than the left-hand-side's type.
format = "{0} {1} {2}";
```

The conversion categories are:

UNUSED indicates an unused argument. For example, in `MessageFormat.format("{0, number} {2, number}", 3.14, "Hello", 2.718)` , the second argument `Hello` is unused. Thus, the conversion categories for the format, `0, number 2, number`, is `(NUMBER, UNUSED, NUMBER)`.

GENERAL means that any value can be supplied as an argument.

DATE is applicable for date, time, and number types. An argument needs to be of `Date`, `Time`, or `Number` type or a subclass of them, including `Timestamp` and the classes listed immediately below.

NUMBER means that the argument needs to be of `Number` type or a subclass: `Number`, `AtomicInteger`, `AtomicLong`, `BigDecimal`, `BigInteger`, `Byte`, `Double`, `Float`, `Integer`, `Long`, `Short`.

11.3 What the Internationalization Format String Checker checks

The Internationalization Format String Checker checks calls to the i18n formatting method `MessageFormat.format` and guarantees the following:

1. The checker issues a warning for the following cases:
   (a) There are missing arguments from what is required by the format string.
   `MessageFormat.format("{0, number} {1, number}", 3.14); // Output:  3.14 {1}`
   (b) More arguments are passed than what is required by the format string.
   `MessageFormat.format("{0, number}", 1, new Date());`
   `MessageFormat.format("{0, number} {0, number}", 3.14, 3.14);`
   This does not cause an error at run time, but it often indicates a programmer mistake. If it is intentional, then you should suppress the warning (see Chapter 26).
   (c) Some argument is an array of objects.
   `MessageFormat.format("{0, number} {1}", array);`
   The checker cannot verify whether the format string is valid, so the checker conservatively issues a warning. This is a limitation of the Internationalization Format String Checker.
2. The checker issues an error for the following cases:
   (a) The format string is invalid.

- Unmatched braces.
  `MessageFormat.format("{0, time", new Date());`
- The argument index is not an integer or is negative.
  `MessageFormat.format("{0.2, time}", new Date());`
  `MessageFormat.format("{-1, time}", new Date());`
- Unknown format type.
  `MessageFormat.format("{0, foo}", 3.14);`
- Missing a format style required for `choice` format.
  `MessageFormat.format("{0, choice}", 3.14);`
- Wrong format style.
  `MessageFormat.format("{0, time, number}", 3.14);`
- Invalid subformats.
  `MessageFormat.format("{0, number, #.#.#}", 3.14)`

(b) Some argument's type doesn't satisfy its conversion category.
`MessageFormat.format("{0, number}", new Date());`

The Checker also detects illegal assignments: assigning a non-format-string or an incompatible format string to a variable declared as containing a specific type of format string. For example,

```
@I18nFormat({GENERAL, NUMBER}) String format;
// OK.
format = "{0} {1, number}";
// OK, GENERAL is weaker (less restrictive) than NUMBER.
format = "{0} {1}";
// OK, it is legal to have fewer arguments than required (less restrictive).
// But the warning will be issued instead.
format = "{0}";

// Error, the format string is stronger (more restrictive) than the specifiers.
format = "{0} {1} {2}";
// Error, the format string is more restrictive. NUMBER is a subtype of GENERAL.
format = "{0, number} {1, number}";
```

11.4 Resource files

A programmer rarely writes an i18n format string literally. (The examples in this chapter show that for simplicity.) Rather, the i18n format strings are read from a resource file. The program chooses a resource file at run time depending on the locale (for example, different resource files for English and Spanish users).
For example, suppose that the `resource1.properties` file contains

```
key1 = The number is {0, number}.
```

Then code such as the following:

```
String formatPattern = ResourceBundle.getBundle("resource1").getString("key1");
System.out.println(MessageFormat.format(formatPattern, 2.2361));
```

will output "The number is 2.2361." A different resource file would contain `key1 = El número es {0, number}.`

When you run the I18n Format String Checker, you need to indicate which resource file it should check. If you change the resource file or use a different resource file, you should re-run the checker to ensure that you did not make an error. The I18n Format String Checker supports two types of resource files: ResourceBundles and property files. The example above shows use of resource bundles. For more about checking property files, see Chapter 12, page 77.

11.5 Running the Internationalization Format Checker

The checker can be invoked by running one of the following commands (with the whole command on one line).

- Using ResourceBundles:
  `javac -processor org.checkerframework.checker.i18nformatter.I18nFormatterChecker -Abundlenames=MyResource MyFile.java`
- Using property files:
  `javac -processor org.checkerframework.checker.i18nformatter.I18nFormatterChecker -Apropfiles=MyResource.properties MyFile.java`
- Not using a property file. Use this if the programmer hard-coded the format patterns without loading them from a property file.
  `javac -processor org.checkerframework.checker.i18nformatter.I18nFormatterChecker MyFile.java`

11.6 Testing whether a string has an i18n format type

In the case that the checker cannot infer the i18n format type of a string, you can use the `I18nFormatUtil.hasFormat` method to define the type of the string in the scope of a conditional statement.

`I18nFormatUtil.hasFormat` returns `true` if the given string has the given i18n format type.

For an example, see Section 11.7.

11.7 Examples of using the Internationalization Format Checker

- Using `MessageFormat.format`.

```
// suppose the bundle "MyResource" contains:  key1={0, number} {1, date}
String value = ResourceBundle.getBundle("MyResource").getString("key1");
MessageFormat.format(value, 3.14, new Date());  // OK
// error: incompatible types in argument; found String, expected number
MessageFormat.format(value, "Text", new Date());
```

- Using the `I18nFormatUtil.hasFormat` method to check whether a format string has particular conversion categories.

```
void test1(String format) {
  if (I18nFormatUtil.hasFormat(format, I18nConversionCategory.GENERAL,
                                       I18nConversionCategory.NUMBER)) {
    MessageFormat.format(format, "Hello", 3.14);  // OK
    // error: incompatible types in argument; found String, expected number
    MessageFormat.format(format, "Hello", "Bye");
    // error: missing arguments; expected 2 but 1 given
    MessageFormat.format(format, "Bye");
    // error: too many arguments; expected 2 but 3 given
    MessageFormat.format(format, "A String", 3.14, 3.14);
  }
}
```

- Using `@I18nFormatFor` to ensure that an argument is a particular type of format string.

```
static void method(@I18nFormatFor("#2") String f, Object... args) {...}

method("{0, number} {1} {2, choice,0#zero|1#one|1<greater than one}", 3.14, "Hello", 100); // OK
```

```
// error: incompatible types in argument; found String, expected number
method("{0, number} {1}", "Bye", "Bye");
```

- Annotating a string with `@I18nFormat`.

```
@I18nFormat({I18nConversionCategory.DATE}) String;
s1 = "{0}";
// error: incompatible types in assignment
s1 = "{0, number}";
```

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 77) 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 78) verifies that code is properly internationalized. A Compiler Message Key Checker (Section 12.3, page 78) 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 General Property File Checker

The general 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 org.checkerframework.checker.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 `https://docs.oracle.com/javase/tutorial/i18n/resbundle/concept.html`.) 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, or I18n Checker, verifies that your code is properly internationalized. Internationalization is the process of designing software so that it can be adapted to different languages and locales without needing to change the code. Localization is the process of adapting internationalized software to specific languages and locales.

Internationalization 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.

If you use the Internationalization Checker, you may want to also use the Internationalization Format String Checker, or I18n Format String Checker (Chapter 11). It verifies that internationalization format strings are well-formed and used with arguments of the proper type, so that `MessageFormat.format` does not fail at run time.

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 general 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 org.checkerframework.checker.i18n.I18nChecker -Abundlenames=MyResource MyFile.java ...
```

You must specify the localization resource, which maps keys to user-visible strings. Like the general Property Key Checker, the Internationalization Checker supports two types of localization resource: `ResourceBundle`s 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 `String`s.

The single annotation used by this checker is `@CompilerMessageKey`. The Checker Framework is completely annotated; for example, class `org.checkerframework.framework.source.Result` uses `@CompilerMessageKey` in methods `failure` and `warning`. For most users of the Checker Framework there will be no need to annotate any `String`s, 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 org.checkerframework.checker.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 general property key checker. Within the Checker Framework we only use property files, so the `-Apropfiles=`*`prop_file`* option should be used.

Chapter 13

Signature Checker for string representations of types

The Signature String Checker, or Signature Checker for short, verifies that string representations of types and signatures are used correctly.

Java defines multiple different string representations for types (see Section 13.1), and it is easy to misuse them or to miss bugs during testing. Using the wrong string format leads to a run-time exception or an incorrect result. This is a particular problem for fully qualified and binary names, which are nearly the same — they differ only for nested classes and arrays.

13.1 Signature annotations

Java defines four main formats for the string representation of a type. There is an annotation for each of these representations. Figure 13.1 shows how they are related.

`@FullyQualifiedName` A *fully qualified name* (JLS §6.7), such as `package.Outer.Inner`, is used in Java code and in messages to the user.

`@BinaryName` A *binary name* (JLS §13.1), such as `package.Outer$Inner`, is the representation of a type in its own `.class` file.

`@FieldDescriptor` A *field descriptor* (JVMS §4.3.2), such as `Lpackage/Outer$Inner;`, is used in a `.class` file's constant pool, for example to refer to other types; it abbreviates primitives and arrays, and uses internal form (JVMS §4.2) for class names.

`@ClassGetName` The type representation used by the `Class.getName()`, `Class.forName(String)`, and `Class.forName(String, boolean, ClassLoader)` methods. This format is: for any non-array type, the binary name; and for any array type, a format like the FieldDescriptor field descriptor, but using "`.`" where the field descriptor uses "`/`".

`@SourceName` A source name is a string that is a valid fully qualified name *and* a valid binary name. A programmer should never or rarely use this — you should know how you intend to use a given variable. The checker infers it for literal strings such as `"package.MyClass"` that are valid in both formats, and you might occasionally see it in an error message. Likewise, you might see other types such as `SourceNameForNonArray`, `BinaryNameForNonArray`, and `FieldDescriptorForArray`, but you generally should not use them either.

Java also defines other string formats for a type: simple names (JLS §6.2), qualified names (JLS §6.2), and canonical names (JLS §6.7). The Signature Checker does not include annotations for these.

Here are examples of the supported formats:

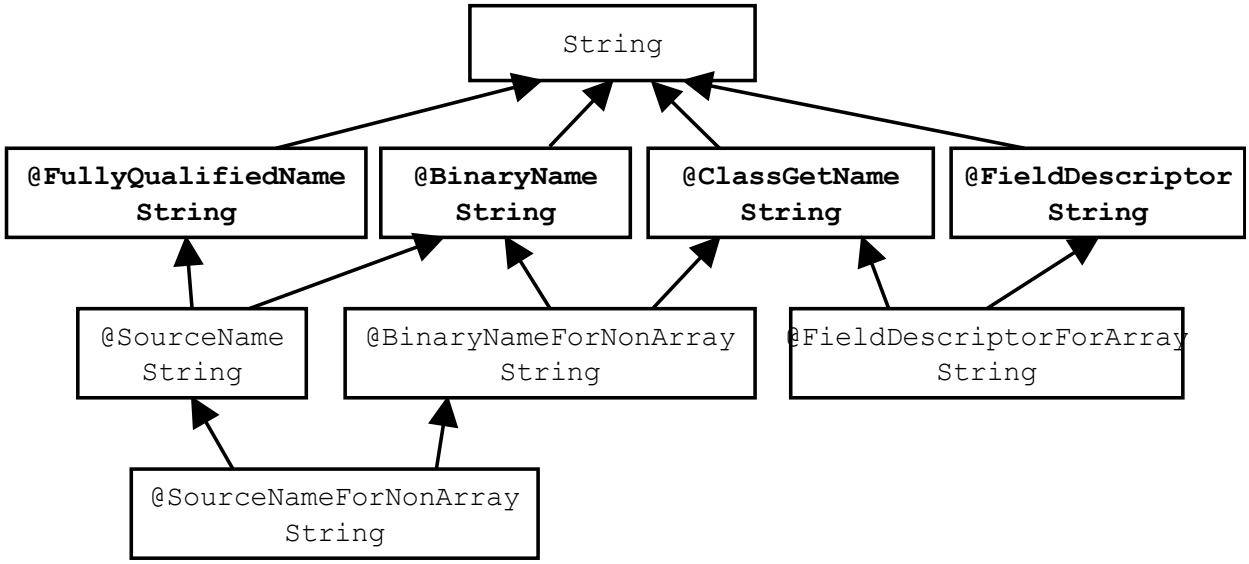


Figure 13.1: Partial type hierarchy for the Signature type system, showing string representations of a Java type. Programmers only need to write the boldfaced qualifiers, in the second row; qualifiers below those are included to improve the internal handling of String literals.

fully-qualified name	binary name	Class.getName	field descriptor
int	int	int	I
int[][]	int[][]	[[I	[[I
MyClass	MyClass	MyClass	LMyClass;
MyClass[]	MyClass[]	[LMyClass;	[LMyClass;
java.lang.Integer	java.lang.Integer	java.lang.Integer	Ljava/lang/Integer;
java.lang.Integer[]	java.lang.Integer[]	[Ljava.lang.Integer;	[Ljava/lang/Integer;
package.Outer.Inner	package.Outer$Inner	package.Outer$Inner	Lpackage/Outer$Inner;
package.Outer.Inner[]	package.Outer$Inner[]	[Lpackage.Outer$Inner;	[Lpackage/Outer$Inner;

Java defines one format for the string representation of a method signature:

`@MethodDescriptor` A *method descriptor* (JVMS §4.3.3) identifies a method's signature (its parameter and return types), just as a field descriptor identifies a type. The method descriptor for the method

```
Object mymethod(int i, double d, Thread t)
```

is

```
(IDLjava/lang/Thread;)Ljava/lang/Object;
```

13.2 What the Signature Checker checks

Certain methods in the JDK, such as `Class.forName`, are annotated indicating the type they require. The Signature Checker ensures that clients call them with the proper arguments. The Signature Checker does not reason about string operations such as concatenation, substring, parsing, etc.

To run the Signature Checker, supply the `-processor org.checkerframework.checker.signature.SignatureChecker` command-line option to javac.

Chapter 14

GUI Effect Checker

One of the most prevalent GUI-related bugs is *invalid UI update* or *invalid thread access*: accessing the UI directly from a background thread.

Most GUI frameworks (including Android, AWT, Swing, and SWT) create a single distinguished thread — the UI event thread — that handles all GUI events and updates. To keep the interface responsive, any expensive computation should be offloaded to *background threads* (also called *worker threads*). If a background thread accesses a UI element such as a JPanel (by calling a JPanel method or reading/writing a field of JPanel), the GUI framework raises an exception that terminates the program. To fix the bug, the background thread should send a request to the UI thread to perform the access on its behalf.

It is difficult for a programmer to remember which methods may be called on which thread(s). The GUI Effect Checker solves this problem. The programmer annotates each method to indicate whether:

- It accesses no UI elements (and may run on any thread); such a method is said to have the "safe effect".
- It may access UI elements (and must run on the UI thread); such a method is said to have the "UI effect".

The GUI Effect Checker verifies these effects and statically enforces that UI methods are only called from the correct thread. A method with the safe effect is prohibited from calling a method with the UI effect.

For example, the effect system can reason about when method calls must be dispatched to the UI thread via a message such as `Display.syncExec`.

```
@SafeEffect
public void calledFromBackgroundThreads(JLabel l) {
    l.setText("Foo");       // Error: calling a @UIEffect method from a @SafeEffect method
    Display.syncExec(new @UI Runnable {
        @UIEffect // inferred by default
        public void run() {
            l.setText("Bar");  // OK: accessing JLabel from code run on the UI thread
        }
    });

}
```

The GUI Effect Checker's annotations fall into three categories:

- effect annotations on methods (Section 14.1),
- class or package annotations controlling the default effect (Section 14.4), and
- *effect-polymorphism*: code that works for both the safe effect and the UI effect (Section 14.5).

14.1 GUI effect annotations

There are two primary GUI effect annotations:

- `@SafeEffect` is a method annotation marking code that must not access UI objects.
- `@UIEffect` is a method annotation marking code that may access UI objects. Most UI object methods (e.g., methods of `JPanel`) are annotated as `@UIEffect`.

`@SafeEffect` is a sub-effect of `@UIEffect`, in that it is always safe to call a `@SafeEffect` method anywhere it is permitted to call a `@UIEffect` method. We write this relationship as

$$\texttt{@SafeEffect} \prec \texttt{@UIEffect}$$

14.2 What the GUI Effect Checker checks

The GUI Effect Checker ensures that only the UI thread accesses UI objects. This prevents GUI errors such as invalid UI update and invalid thread access.

The GUI Effect Checker issues errors in the following cases:

- A `@UIEffect` method is invoked by a `@SafeEffect` method.
- Method declarations violate subtyping restrictions: a supertype declares a `@SafeEffect` method, and a subtype annotates an overriding version as `@UIEffect`.

Additionally, if a method implements or overrides a method in two supertypes (two interfaces, or an interface and parent class), and those supertypes give different effects for the methods, the GUI Effect Checker issues a warning (not an error).

14.3 Running the GUI Effect Checker

The GUI Effect Checker can be invoked by running the following command:

```
javac -processor org.checkerframework.checker.guieffect.GuiEffectChecker MyFile.java ...
```

14.4 Annotation defaults

The default method annotation is `@SafeEffect`, since most code in most programs is not related to the UI. This also means that typically, code that is unrelated to the UI need not be annotated at all.

The GUI Effect Checker provides three primary ways to change the default method effect for a class or package:

- `@UIType` is a class annotation that makes the effect for unannotated methods in that class default to `@UIEffect`. (See also `@UI` in Section 14.5.2.)
- `@UIPackage` is a *package* annotation, that makes the effect for unannotated methods in that package default to `@UIEffect`. It is not transitive; a package nested inside a package marked `@UIPackage` does not inherit the changed default.
- `@SafeType` is a class annotation that makes the effect for unannotated methods in that class default to `@SafeEffect`. Because `@SafeEffect` is already the default effect, `@SafeType` is only useful for class types inside a package marked `@UIPackage`.

There is one other place where the default annotation is not automatically `@SafeEffect`: anonymous inner classes. Since anonymous inner classes exist primarily for brevity, it would be unfortunate to spoil that brevity with extra annotations. By default, an anonymous inner class method that overrides or implements a method of the parent type inherits that method's effect. For example, an anonymous inner class implementing an interface with method `@UIEffect void m()` need not explicitly annotate its implementation of `m()`; the implementation will inherit the parent's effect. Methods of the anonymous inner class that are not inherited from a parent type follow the standard defaulting rules.

14.5 Polymorphic effects

Sometimes a type is reused for both UI-specific and background-thread work. A good example is the `Runnable` interface, which is used both for creating new background threads (in which case the `run()` method must have the `@SafeEffect`) and for sending code to the UI thread to execute (in which case the `run()` method may have the `@UIEffect`). But the declaration of `Runnable.run()` may have only one effect annotation in the source code. How do we reconcile these conflicting use cases?

Effect-polymorphism permits a type to be used for both UI and non-UI purposes. It is similar to Java's generics in that you define, then use, the effect-polymorphic type. Recall that to *define* a generic type, you write a type parameter such as `<T>` and use it in the body of the type definition; for example, `class List<T> { ... T get() {...} ... }`. To *instantiate* a generic type, you write its name along with a type argument; for example, `List<Date> myDates;`.

14.5.1 Defining an effect-polymorphic type

To declare that a class is effect-polymorphic, annotate its definition with `@PolyUIType`. To use the effect variable in the class body, annotate a method with `@PolyUIEffect`. It is an error to use `@PolyUIEffect` in a class that is not effect-polymorphic.

Consider the following example:

```
@PolyUIType
public interface Runnable {
    @PolyUIEffect
    void run();
}
```

This declares that class `Runnable` is parameterized over one generic effect, and that when `Runnable` is instantiated, the effect argument will be used as the effect for the `run` method.

14.5.2 Using an effect-polymorphic type

To instantiate an effect-polymorphic type, write one of these three type qualifiers before a use of the type:

- `@AlwaysSafe` instantiates the type's effect to `@SafeEffect`.
- `@UI` instantiates the type's effect to `@UIEffect`. *Additionally*, it changes the default method effect for the class to `@UIEffect`.
- `@PolyUI` instantiates the type's effect to `@PolyUIEffect` for the same instantiation as the current (containing) class. For example, this is the qualifier of the receiver `this` inside a method of a `@PolyUIType` class, which is how one method of an effect-polymorphic class may call an effect-polymorphic method of the same class.

As an example:

```
@AlwaysSafe Runnable s = ...;    s.run();    // s.run() is @SafeEffect
@PolyUI Runnable p = ...;        p.run();    // p.run() is @PolyUIEffect (context-dependent)
@UI Runnable u = ...;            u.run();    // u.run() is @UIEffect
```

It is an error to apply an effect instantiation qualifier to a type that is not effect-polymorphic.

14.5.3 Subclassing a specific instantiation of an effect-polymorphic type

Sometimes you may wish to subclass a specific instantiation of an effect-polymorphic type, just as you may extend `List<String>`.

To do this, simply place the effect instantiation qualifier by the name of the type you are defining, e.g.:

```
@UI
public class UIRunnable extends Runnable {...}
@AlwaysSafe
public class SafeRunnable extends Runnable {...}
```

The GUI Effect Checker will automatically apply the qualifier to all classes and interfaces the class being defined extends or implements. (This means you cannot write a class that is a subtype of a `@AlwaysSafe Foo` and a `@UI Bar`, but this has not been a problem in our experience.)

14.5.4 Subtyping with polymorphic effects

With three effect annotations, we must extend the static sub-effecting relationship:

$$\texttt{@SafeEffect} \prec \texttt{@PolyUIEffect} \prec \texttt{@UIEffect}$$

This is the correct sub-effecting relation because it is always safe to call a `@SafeEffect` method (whether from an effect-polymorphic method or a UI method), and a `@UIEffect` method may safely call any other method.

This induces a subtyping hierarchy on type qualifiers:

$$\texttt{@AlwaysSafe} \prec \texttt{@PolyUI} \prec \texttt{@UI}$$

This is sound because a method instantiated according to any qualifier will always be safe to call in place of a method instantiated according to one of its super-qualifiers. This allows clients to pass "safer" instances of some object type to a given method.

14.6 References

The ECOOP 2013 paper "JavaUI: Effects for Controlling UI Object Access" includes some case studies on the checker's efficacy, including descriptions of the relatively few false warnings we encountered. It also contains a more formal description of the effect system. You can obtain the paper at:
`http://homes.cs.washington.edu/~mernst/pubs/gui-thread-ecoop2013-abstract.html`

Chapter 15

Units Checker

For many applications, it is important to use the correct units of measurement for primitive types. For example, NASA's Mars Climate Orbiter (cost: $327 million) was lost because of a discrepancy between use of the metric unit Newtons and the imperial measure Pound-force.

The *Units Checker* ensures consistent usage of units. For example, consider the following code:

```
@m int meters = 5 * UnitsTools.m;
@s int secs = 2 * UnitsTools.s;
@mPERs int speed = meters / secs;
```

Due to the annotations `@m` and `@s`, the variables `meters` and `secs` are guaranteed to contain only values with meters and seconds as units of measurement. Utility class `UnitsTools` provides constants with which unqualified integer are multiplied to get values of the corresponding unit. The assignment of an unqualified value to `meters`, as in `meters = 99`, will be flagged as an error by the Units Checker.

The division `meters/secs` takes the types of the two operands into account and determines that the result is of type meters per second, signified by the `@mPERs` qualifier. We provide an extensible framework to define the result of operations on units.

15.1 Units annotations

The checker currently supports two varieties of units annotations: kind annotations (`@Length`, `@Mass`, ...) and the SI units (`@m`, `@kg`, ...).

Kind annotations can be used to declare what the expected unit of measurement is, without fixing the particular unit used. For example, one could write a method taking a `@Length` value, without specifying whether it will take meters or kilometers. The following kind annotations are defined:

`@Area`
`@Current`
`@Length`
`@Luminance`
`@Mass`
`@Speed`
`@Substance`
`@Temperature`
`@Time`

For each kind of unit, the corresponding SI unit of measurement is defined:

1. For `@Area`: the derived units square millimeters `@mm2`, square meters `@m2`, and square kilometers `@km2`

2. For `@Current`: Ampere `@A`
3. For `@Length`: Meters `@m` and the derived units millimeters `@mm` and kilometers `@km`
4. For `@Luminance`: Candela `@cd`
5. For `@Mass`: kilograms `@kg` and the derived unit grams `@g`
6. For `@Speed`: meters per second `@mPERs` and kilometers per hour `@kmPERh`
7. For `@Substance`: Mole `@mol`
8. For `@Temperature`: Kelvin `@K` and the derived unit Celsius `@C`
9. For `@Time`: seconds `@s` and the derived units minutes `@min` and hours `@h`

You may specify SI unit prefixes, using enumeration `Prefix`. The basic SI units (`@s`, `@m`, `@g`, `@A`, `@K`, `@mol`, `@cd`) take an optional `Prefix` enum as argument. For example, to use nanoseconds as unit, you could use `@s(Prefix.nano)` as a unit type. You can sometimes use a different annotation instead of a prefix; for example, `@mm` is equivalent to `@m(Prefix.milli)`.

Class `UnitsTools` contains a constant for each SI unit. To create a value of the particular unit, multiply an unqualified value with one of these constants. By using static imports, this allows very natural notation; for example, after statically importing `UnitsTools.m`, the expression `5 * m` represents five meters. As all these unit constants are public, static, and final with value one, the compiler will optimize away these multiplications.

15.2 Extending the Units Checker

You can create new kind annotations and unit annotations that are specific to the particular needs of your project. An easy way to do this is by copying and adapting an existing annotation. (In addition, search for all uses of the annotation's name throughout the Units Checker implementation, to find other code to adapt; read on for details.)

Here is an example of a new unit annotation.

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@TypeQualifier
@SubtypeOf( { Time.class } )
@UnitsMultiple(quantity=s.class, prefix=Prefix.nano)
@Target(ElementType.TYPE_USE, ElementType.TYPE_PARAMETER)
public @interface ns {}
```

The `@SubtypeOf` meta-annotation specifies that this annotation introduces an additional unit of time. The `@UnitsMultiple` meta-annotation specifies that this annotation should be a nano multiple of the basic unit `@s`: `@ns` and `@s(Prefix.nano)` behave equivalently and interchangeably. Most annotation definitions do not have a `@UnitsMultiple` meta-annotation.

To take full advantage of the additional unit qualifier, you need to do two additional steps. (1) Provide constants that convert from unqualified types to types that use the new unit. See class `UnitsTools` for examples (you will need to suppress a checker warning in just those few locations). (2) Put the new unit in relation to existing units. Provide an implementation of the `UnitsRelations` interface as a meta-annotation to one of the units.

See demonstration `examples/units-extension/` for an example extension that defines Hertz (hz) as scalar per second, and defines an implementation of `UnitsRelations` to enforce it.

15.3 What the Units Checker checks

The Units Checker ensures that unrelated types are not mixed.

All types with a particular unit annotation are disjoint from all unannotated types, from all types with a different unit annotation, and from all types with the same unit annotation but a different prefix.

Subtyping between the units and the unit kinds is taken into account, as is the `@UnitsMultiple` meta-annotation.

Multiplying a scalar with a unit type results in the same unit type.

The division of a unit type by the same unit type results in the unqualified type.

Multiplying or dividing different unit types, for which no unit relation is known to the system, will result in a `MixedUnits` type, which is separate from all other units. If you encounter a `MixedUnits` annotation in an error message, ensure that your operations are performed on correct units or refine your `UnitsRelations` implementation.

The Units Checker does *not* change units based on multiplication; for example, if variable `mass` has the type `@kg double`, then `mass * 1000` has that same type rather than the type `@g double`. (The Units Checker has no way of knowing whether you intended a conversion, or you were computing the mass of 1000 items. You need to make all conversions explicit in your code, and it's good style to minimize the number of conversions.)

15.4 Running the Units Checker

The Units Checker can be invoked by running the following commands.

- If your code uses only the SI units that are provided by the framework, simply invoke the checker:

  ```
  javac -processor org.checkerframework.checker.units.UnitsChecker MyFile.java ...
  ```

- If you define your own units, provide the name of the annotations using the `-Aunits` option:

  ```
  javac -processor org.checkerframework.checker.units.UnitsChecker \
        -Aunits=myproject.qual.MyUnit,myproject.qual.MyOtherUnit MyFile.java ...
  ```

15.5 Suppressing warnings

One example of when you need to suppress warnings is when you initialize a variable with a unit type by a literal value. To remove this warning message, it is best to introduce a constant that represents the unit and to add a `@SuppressWarnings` annotation to that constant. For examples, see class `UnitsTools`.

15.6 References

- The GNU Units tool provides a comprehensive list of units:
  `http://www.gnu.org/software/units/`
- The F# units of measurement system inspired some of our syntax:
  `https://en.wikibooks.org/wiki/F_Sharp_Programming/Units_of_Measure`

Chapter 16

Constant Value Checker

The Constant Value Checker is a constant propagation analysis: for each variable, it determines whether that variable's value can be known at compile time.

There are two ways to run the Constant Value Checker.

- Typically, it is automatically run by another type checker. When using the Constant Value Checker as part of another checker, the `statically-executable.astub` file in the Constant Value Checker directory must be passed as a stub file for the checker.
- Alternately, you can run just the Constant Value Checker, by supplying the following command-line option to javac: `-processor org.checkerframework.common.value.ValueChecker -Astubs=statically-executable.astub`

16.1 Annotations

The Constant Value Checker uses type annotations to indicate the value of an expression (Section 16.1.1), and it uses method annotations to indicate methods that the Constant Value Checker can execute at compile time (Section 16.1.2).

16.1.1 Type Annotations

Typically, the programmer does not write any type annotations. Rather, the type annotations are inferred by the Constant Value Checker. The programmer is also permitted to write type annotations. This is only necessary in locations where the Constant Value Checker does not infer annotations: on fields and method signatures.

The type annotations are `@BoolVal`, `@IntVal`, `@DoubleVal`, and `@StringVal`.

Each type annotation takes as an argument a set of values, and its meaning is that at run time, the expression evaluates to one of the values. For example, an expression of type `@StringVal("a", "b")` evaluates to one of the values `"a"`, `"b"`, or `null`. The set is limited to 10 entries; if a variable could be more than 10 different values, the Constant Value Checker gives up and its type becomes `@UnknownVal` instead.

Figure 16.1 shows the subtyping relationship among the type annotations. For two annotations of the same type, subtypes have a smaller set of possible values, as also shown in the figure. Because `int` can be casted to `double`, an `@IntVal` annotation is a subtype of a `@DoubleVal` annotation with the same values.

Figure 16.2 shows how the Constant Value Checker infers type annotations (using flow-sensitive type qualifier refinement, Section 25.4).

16.1.2 Compile-time execution of expressions

Whenever all the operands of an expression are compile-time constants (that is, their types have constant-value type annotations), the Constant Value Checker attempts to execute the expression. This is independent of any optimizations performed by the compiler and does not affect the code that is generated.

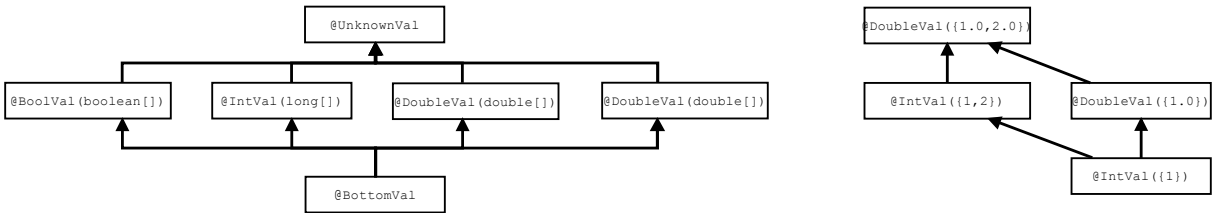


Figure 16.1: The type qualifier hierarchy of the Constant Value Checker annotations. Qualifiers in gray are used internally by the type system but should never be written by a programmer. On the right are examples of additional subtyping relationships that depend on the annotations' arguments.

```
public void foo(boolean b) {
    int i = 1;     // i has type:  @IntVal({1}) int
    if (b) {
        i = 2;     // i now has type:  @IntVal({2}) int
    }
                   // i now has type:  @IntVal({1,2}) int
    i = i + 1;     // i now has type:  @IntVal({2,3}) int
}
```

Figure 16.2: The Constant Value Checker infers different types for a variable on different lines of the program.

The Constant Value Checker statically executes operators that do not throw exceptions (e.g., `+`, `-`, `<<`, `!=`), and also calls to methods annotated with `@StaticallyExecutable`.

A `@StaticallyExecutable` method must be `@Pure` (side-effect-free and deterministic). Additionally, a `@StaticallyExecutable` method and any method it calls must be on the classpath for the compiler, because they are reflectively called at compile-time to perform the constant value analysis. Any standard library methods (such as those annotated as `@StaticallyExecutable` in file `statically-executable.astub`) will already be on the classpath.

To use `@StaticallyExecutable` on methods in your own code, you should first compile the code without the Constant Value Checker and then add the location of the resulting `.class` files to the classpath. This can be done by either adding the destination path to your environment variable `CLASSPATH` or by passing the argument `-classpath path/to/class/files` to the call. The latter would look similar to: `-processor org.checkerframework.common.value.ValueChec -Astubs=statically-executable.astub -classpath $CLASSPATH:$MY_PROJECT/build/`

```
@StaticallyExecutable @Pure
public int foo(int a, int b) {
    return a + b;
}

public void bar() {
    int a = 5;          // a has type:  @IntVal({5}) int
    int b = 4;          // b has type:  @IntVal({4}) int
    int c = foo(a, b);  // c has type:  @IntVal({9}) int
}
```

Figure 16.3: The `@StaticallyExecutable` annotation enables constant propagation through method calls.

16.2 Warnings

The Constant Value Checker issues a warning if it cannot load and run, at compile time, a method marked as `@StaticallyExecutable`. If it issues such a warning, then the return value of the method will be `@UnknownVal` instead of being able to be resolved to a specific value annotation. Some examples of these:

- `[class.find.failed] Failed to find class named Test.`
  The checker could not find the class specified for resolving a `@StaticallyExecutable` method. Typically this is caused by not providing the path of a class-file needed to the classpath.
- `[method.find.failed] Failed to find a method named foo with argument types [@IntVal(3) int]. Treating result as @UnknownVal`
  The checker could not find the method `foo(int)` specified for resolving a `@StaticallyExecutable` method, but could find the class. This is usually due to providing an outdated version of the class-file that does not contain the `@StaticallyExecutable` method.
- `[method.evaluation.exception] Failed to evaluate method public static int Test.foo(int) because it threw an exception: java.lang.ArithmeticException: / by zero. Treating result as @UnknownVal`
  An exception was thrown when trying to statically execute the method. In this case it was a divide-by-zero exception. If the arguments to the method each only had one value in their annotations then this exception will always occur when the program is actually run as well. If there are multiple possible values then the exception might not be thrown on every execution, depending on the run-time values.

There is one other situation in which the Constant Value Checker produces a warning message:

- `[too.many.values.given] Annotation ignored because the maximum number of values tracked is 10.`
  The Constant Value Checker only tracks up to 10 possible values for an expression. If you write an annotation with more values than will be tracked, the annotation is ignored.

Chapter 17

Aliasing Checker

The Aliasing Checker identifies expressions that definitely have no aliases.

Two expressions are aliased when they have the same non-primitive value; that is, they are references to the identical Java object in the heap. Another way of saying this is that two expressions, *exprA* and *exprB*, are aliases of each other when *exprA*==*exprB* at the same program point.

Assigning to a variable or field typically creates an alias. For example, after the statement `a = b;`, the variables `a` and `b` are aliased.

Knowing that an expression is not aliased permits more accurate reasoning about how side effects modify the expression's value.

To run the Aliasing Checker, supply the `-processor org.checkerframework.common.aliasing.AliasingChecker` command-line option to javac. However, a user rarely runs the Aliasing Checker directly. This type system is mainly intended to be used together with other type systems. For example, the SPARTA information flow type-checker (Section 23.8) uses the Aliasing Checker to improve its type refinement — if an expression has no aliases, a more refined type can often be inferred, otherwise the type-checker makes conservative assumptions.

17.1 Aliasing annotations

There are two possible types for an expression:

`@MaybeAliased` is the type of an expression that might have an alias. This is the default, so every unannotated type is `@MaybeAliased`. (This includes the type of `null`.)

`@Unique` is the type of an expression that has no aliases.
The `@Unique` annotation is only allowed at local variables, method parameters, constructor results, and method returns. A constructor's result should be annotated with `@Unique` only if the constructor's body does not creates an alias to the constructed object.

There are also two annotations, which are currently trusted instead of verified, that can be used on formal parameters (including the receiver parameter, `this`):

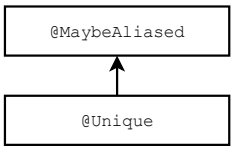


Figure 17.1: Type hierarchy for the Aliasing type system.

@NonLeaked identifies a formal parameter that is not leaked nor returned by the method body. For example, the formal parameter of the String copy constructor, `String(String s)`, is `@NonLeaked` because the body of the method only makes a copy of the parameter.

@LeakedToResult is used when the parameter may be returned, but it is not otherwise leaked. For example, the receiver parameter of `StringBuffer.append(StringBuffer this, String s)` is `@LeakedToResult`, because the method returns the updated receiver.

17.2 Leaking contexts

This section lists the expressions that create aliases. These are also called "leaking contexts".

Assignments After an assignment, the left-hand side and the right-hand side are typically aliased. (The only counterexample is when the right-hand side is a fresh expression; see Section 17.4.)

```
@Unique Object u = ...;
Object o = u;                    // (not.unique) type-checking error!
```

If this example type-checked, then `u` and `o` would be aliased. For this example to type-check, either the `@Unique` annotation on the type of `u`, or the `o = u;` assignment, must be removed.

Method calls and returns (pseudo-assignments) Passing an argument to a method is a "pseudo-assignment" because it effectively assigns the argument to the formal parameter. Return statements are also pseudo-assignments. As with assignments, the left-hand side and right-hand side of pseudo-assignments are typically aliased.
Here is an example for argument-passing:

```
void foo(Object o) { ... }

@Unique Object u = ...;
foo(u);   // type-checking error, because foo may create an alias of the passed argument
```

Passing a non-aliased reference to a method does not necessarily create an alias. However, the body of the method might create an alias or leak the reference. Thus, the Aliasing Checker always treats a method call as creating aliases for each argument unless the corresponding formal parameter is marked as @`@NonLeaked` or @`@LeakedToResult`.
Here is an example for a return statement:

```
Object id(@Unique Object p) {
    return p;     // (not.unique) type-checking error!
}
```

If this code type-checked, then it would be possible for clients to write code like this:

```
@Unique Object u = ...;
Object o = id(u);
```

after which there is an alias to `u` even though it is declared as `@Unique`.
However, it is permitted to write

```
Object id(@LeakedToResult Object p) {
    return p;
}
```

after which the following code type-checks:

```
@Unique Object u = ...;
id(u);                   // method call result is not used
Object o1 = ...;
Object o2 = id(o1);      // argument is not @Unique
```

Throws A thrown exception can be captured by a catch block, which creates an alias of the thrown exception.

```
void foo() {
    @Unique Exception uex = new Exception();
    try {
        throw uex;    // (not.unique) type-checking error!
    } catch (Exception ex) {
        // uex and ex refer to the same object here.
    }
}
```

Array initializers Array initializers assign the elements in the initializers to corresponding indexes in the array, therefore expressions in an array initializer are leaked.

```
void foo() {
    @Unique Object o = new Object();
    Object[] ar = new Object[] { o };  // (not.unique) type-checking error!
    // The expressions o and ar[0] are now aliased.
}
```

17.3 Restrictions on where `@Unique` may be written

The `@Unique` qualifier may not be written on locations such as fields, array elements, and type parameters.

As an example of why `@Unique` may not be written on a field's type, consider the following code:

```
class MyClass {
    @Unique Object field;
    void foo() {
        MyClass myClass2 = this;
        // this.field is now an alias of myClass2.field
    }
}
```

That code must not type-check, because `field` is declared as `@Unique` but has an alias. The Aliasing Checker solves the problem by forbidding the `@Unique` qualifier on subcomponents of a structure, such as fields. Other solutions might be possible; they would be more complicated but would permit more code to type-check.

`@Unique` may not be written on a type parameter for similar reasons. The assignment

```
List<@Unique Object> l1 = ...;
List<@Unique Object> l2 = l1;
```

must be forbidden because it would alias `l1.get(0)` with `l2.get(0)` even though both have type `@Unique`. The Aliasing Checker forbids this code by rejecting the type `List<@Unique Object>`.

17.4 Aliasing type refinement

Type refinement enables a type checker to treat an expression as a subtype of its declared type. For example, even if you declare a local variable as `@MaybeAliased` (or don't write anything, since `@MaybeAliased` is the default), sometimes the Aliasing Checker can determine that it is actually `@Unique`. For more details, see Section 25.4.

The Aliasing Checker treats type refinement in the usual way, except that at (pseudo-)assignments the right-hand-side (RHS) may lose its type refinement, before the left-hand-side (LHS) is type-refined. The RHS always loses its type refinement (it is widened to `@MaybeAliased`, and its declared type must have been `@MaybeAliased`) except in the following cases:

```
// Annotations on the StringBuffer class, used in the examples below.
// class StringBuffer {
//  @Unique StringBuffer();
//  StringBuffer append(@LeakedToResult StringBuffer this, @NonLeaked String s);
// }

void foo() {
    StringBuffer sb = new StringBuffer();    // sb is refined to @Unique.

    StringBuffer sb2 = sb;                   // sb loses its refinement.
    // Both sb and sb2 have aliases and because of that have type @MaybeAliased.
}

void bar() {
    StringBuffer sb = new StringBuffer();     // sb is refined to @Unique.

    sb.append("someString");
    // sb stays @Unique, as no aliases are created.

    StringBuffer sb2 = sb.append("someString");
    // sb is leaked and becomes @MaybeAliased.

    // Both sb and sb2 have aliases and because of that have type @MaybeAliased.
}
```

Figure 17.2: Example of Aliasing Checker's type refinement rules.

- The RHS is a fresh expression — an expression that returns a different value each time it is evaluated. In practice, this is only method/constructor calls with `@Unique` return type. A variable/field is not fresh because it can return the same value when evaluated twice.
- The LHS is a `@NonLeaked` formal parameter and the RHS is an argument in a method call or constructor invocation.
- The LHS is a `@LeakedToResult` formal parameter, the RHS is an argument in a method call or constructor invocation, and the method's return value is discarded — that is, the method call or constructor invocation is written syntactically as a statement rather than as a part of a larger expression or statement.

A consequence of the above rules is that most method calls are treated conservatively. If a variable with declared type `@MaybeAliased` has been refined to `@Unique` and is used as an argument of a method call, it usually loses its `@Unique` refined type.

Figure 17.2 gives an example of the Aliasing Checker's type refinement rules.

Chapter 18

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 org.checkerframework.checker.linear.LinearChecker` command-line option to javac.

Figure 18.1 gives an example of the Linear Checker's rules.

18.1 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 currently 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 18.1: Example of Linear Checker rules.

18.2 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 hash table, because hash table 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 19

IGJ immutability checker

Note: The IGJ type-checker has some known bugs and limitations. Nonetheless, it may still be useful to you.

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 org.checkerframework.checker.igj.IGJChecker` command-line option to javac. For examples, see Section 19.7.

19.1 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 19.5), which is a slightly different dialect of IGJ than that described in the ESEC/FSE paper.

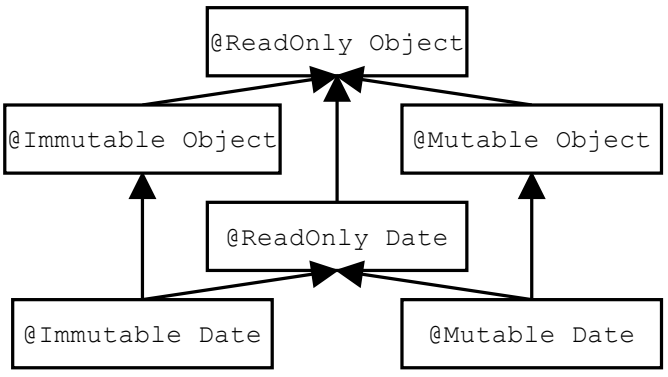


Figure 19.1: Type hierarchy for three of IGJ's type qualifiers.

19.2 IGJ Annotations

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

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

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

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

`@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. `@AssignsFields` is assumed to be true of the result of a constructor, so it does not need to be written there.

`@I` simulates mutability overloading or the template behavior of generics. It can be applied to classes, methods, and parameters. See Section 19.5.3.

For additional details, see [ZPA+07].

19.3 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.

19.4 Implicit and default qualifiers

As described in Section 25.3, 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 25.4).
- `@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>`).

19.5 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.

19.5.1 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.

19.5.2 Syntax Changes

- Immutability is specified through type annotations [Ern08] (Section 19.2), 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 19.5.3.

19.5.3 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 FileDescriptor this) { }
}

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

}
```

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

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

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

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

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

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

19.6 Iterators and their abstract state

This section explains why the receiver of `Iterator.next()` is annotated as `@ReadOnly`.

An iterator conceptually has two pieces of state:

1. the underlying collection
2. an index into that collection (indicating the next object to be returned)

We choose to exclude the index from the abstract state of the iterator. That is, a change to the index does not count as a mutation of the iterator itself.

Changes to the underlying collection are more important and interesting, and unintentional changes are much more likely to lead to important errors. Therefore, this choice about the iterator's abstract state appears to be more useful than other choices. For example, if the iterator's abstract state included both the underlying collection and the index, then there would be no way to express, or check, that `Iterator.next` does not change the underlying collection.

19.7 Examples

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

```
javac -processor org.checkerframework.checker.igj.IGJChecker examples/IGJExample.java
```

The IGJ Checker itself is also annotated with IGJ annotations.

Chapter 20

Javari immutability checker

Note: The Javari type-checker has some known bugs and limitations. Nonetheless, it may still be useful to you.

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://types.cs.washington.edu/javari/`) contains papers that explain the Javari language and type system. By contrast to those papers, the Javari Checker uses an annotation-based dialect of the Javari language.

The Javarifier tool infers Javari types for an existing program; see Section 20.2.2.

Also consider the IGJ Checker (Chapter 19). The IGJ type system is more expressive than that of Javari, and the IGJ Checker is a bit more robust. However, IGJ lacks a type inference tool such as Javarifier.

To run the Javari Checker, supply the `-processor org.checkerframework.checker.javari.JavariChecker` command-line option to javac. For examples, see Section 20.5.

20.1 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.

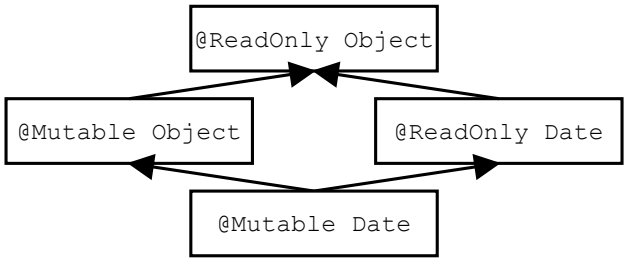


Figure 20.1: Type hierarchy for Javari's ReadOnly type qualifier.

@PolyRead (previously named `@RoMaybe`) specifies polymorphism over mutability; it simulates mutability overloading. It can be applied to methods and parameters. See Section 24.2 and the `@PolyRead` Javadoc for more details.

@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.

20.2 Writing Javari annotations

20.2.1 Implicit qualifiers

As described in Section 25.3, 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`.

20.2.2 Inference of Javari annotations

It can be tedious to write annotations in your code. The Javarifier tool (`http://types.cs.washington.edu/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.

20.3 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.

20.4 Iterators and their abstract state

For an explanation of why the receiver of `Iterator.next()` is annotated as `@ReadOnly`, see Section 19.6.

20.5 Examples

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

```
javac -processor org.checkerframework.checker.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 21

Reflection resolution

A call to `Method.invoke` might reflectively invoke any method, so the annotated JDK contains conservative annotations for `Method.invoke`. These conservative library annotations often cause a checker to issue false positive warnings when type-checking code that uses reflection.

If you supply the `-AresolveReflection` command-line option, the Checker Framework attempts to resolve reflection. At each call to `Method.invoke` or `Constructor.newInstance`, the Checker Framework first soundly estimates which methods might be invoked at runtime. When type-checking the call, the Checker Framework uses a library annotation that indicates the parameter and return types of the possibly-invoked methods.

If the estimate of invoked methods is small, these types are precise and the checker issues fewer false positive warnings. If the estimate of invoked methods is large, these types are no better than the conservative library annotations.

Reflection resolution is disabled by default, because it increases the time to type-check a program. You should enable reflection resolution with the `-AresolveReflection` command-line option if, for some call site of `Method.invoke` or `Constructor.newInstance` in your program:

1. the conservative library annotations on `Method.invoke` or `Constructor.newInstance` cause false positive warnings,
2. the set of possibly-invoked methods or constructors can be known at compile time, and
3. the reflectively invoked methods/constructors are on the class path at compile time.

Reflection resolution does not change your source code or generated code. In particular, it does not replace the `Method.invoke` or `Constructor.newInstance` calls.

The command-line option `-AresolveReflection=debug` outputs verbose information about the reflection resolution process.

Section 21.1 first describes the MethodVal and ClassVal Checkers, which reflection resolution uses internally. Then, Section 21.2 gives examples of reflection resolution.

21.1 MethodVal and ClassVal Checkers

The implementation of reflection resolution internally uses the ClassVal Checker (Section 21.1.1) and the MethodVal Checker (Section 21.1.2). They are very similar to the Constant Value Checker (Section 16) in that their annotations estimate the run-time value of an expression.

In some cases, you may need to write annotations such as `@ClassVal`, `@MethodVal`, `@StringVal` and `@ArrayLen` (from the Constant Value Checker, Section 16) to aid in reflection resolution. Often, though, these annotations can be inferred (Section 21.1.3).

21.1.1 ClassVal Checker

The ClassVal Checker defines the following annotations:

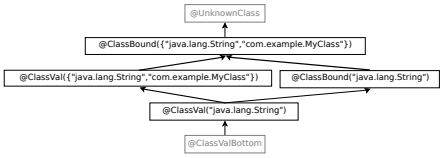


Figure 21.1: Partial type hierarchy for the ClassVal type system. The type qualifiers in gray (`@UnknownClass` and `@ClassValBottom`) should never be written in source code; they are used internally by the type system.

`@ClassVal(String[] value)` If an expression has `@ClassVal` type with a single argument, then its exact run-time value is known at compile time. For example, `@ClassVal("java.util.HashMap")` indicates that the `Class` object represents the `java.util.HashMap` class.
If multiple arguments are given, then the expression's run-time value is known to be in that set.
The arguments are binary names (JLS §13.1).

`@ClassBound(String[] value)` If an expression has `@ClassBound` type, then its run-time value is known to be upper-bounded by that type. For example, `@ClassBound("java.util.HashMap")` indicates that the `Class` object represents `java.util.HashMap` or a subclass of it.
If multiple arguments are given, then the run-time value is equal to or a subclass of some class in that set.
The arguments are binary names (JLS §13.1).

`@UnknownClass` Indicates that there is no compile-time information about the run-time value of the class — or that the Java type is not `Class`. This is the default qualifier, and it may not be written in source code.

`@ClassValBottom` Type given to the `null` literal. It may not be written in source code.

Subtyping rules

Figure 21.1 shows part of the type hierarchy of the ClassVal type system. `@ClassVal(A)` is a subtype of `@ClassVal(B)` if A is a subset of B. `@ClassBound(A)` is a subtype of `@ClassBound(B)` if A is a subset of B. `@ClassVal(A)` is a subtype of `@ClassBound(B)` if A is a subset of B.

21.1.2 MethodVal Checker

The MethodVal Checker defines the following annotations:

`@MethodVal(String[] className, String[] methodName, int[] params)` Indicates that an expression of type `Method` or `Constructor` has a run-time value in a given set. If the set has size *n*, then each of `@MethodVal`'s arguments is an array of size *n*, and the *i*th method in the set is represented by { className[i], methodName[i], params[i] }. For a constructor, the method name is "`<init>`".
Consider the following example:

```
@MethodVal(className={"java.util.HashMap", "java.util.HashMap"},
           methodName={"containsKey", "containsValue"},
           params={1, 1})
```

This `@MethodVal` annotation indicates that the `Method` is either `HashMap.containsKey` with 1 formal parameter or `HashMap.containsValue` with 1 formal parameter.
The `@MethodVal` type qualifier indicates the number of parameters that the method takes, but not their type. This means that the Checker Framework's reflection resolution cannot distinguish among overloaded methods.

`@UnknownMethod` Indicates that there is no compile-time information about the run-time value of the method — or that the Java type is not `Method` or `Constructor`. This is the default qualifier, and it may not be written in source code.

`@MethodValBottom` Type given to the `null` literal. It may not be written in source code.

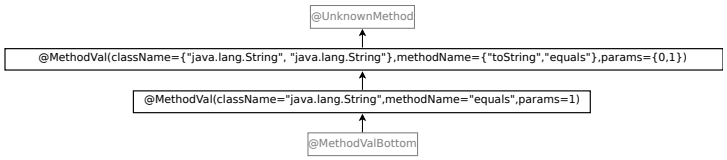


Figure 21.2: Partial type hierarchy for the MethodVal type system. The type qualifiers in gray (`@UnknownMethod` and `@MethodValBottom`) should never be written in source code; they are used internally by the type system.

$$\frac{bn \text{ is the binary name of } \texttt{C}}{\texttt{C.class} : \textbf{@ClassVal(}bn\textbf{)}}$$

$$\frac{s : \texttt{@StringVal(}\nu\texttt{)}}{\texttt{Class.forName(}s\texttt{)} : \textbf{@ClassVal(}\nu\textbf{)}}$$

$$\frac{e : \tau \quad bn \text{ is the binary name of } \tau}{e\texttt{.getClass()} : \textbf{@ClassBound(}bn\textbf{)}}$$

$$\frac{\begin{array}{c}(e : \texttt{@ClassBound(}\nu\texttt{)} \ \vee \ e : \texttt{@ClassVal(}\nu\texttt{)}) \\ s \ : \ \texttt{@StringVal(}\mu\texttt{)} \quad p \ : \ \texttt{@ArrayLen(}\pi\texttt{)}\end{array}}{e\texttt{.getMethod(}s\texttt{,}p\texttt{)} : \textbf{@MethodVal(}\texttt{cn=}\nu\texttt{,mn=}\mu\texttt{,np=}\pi\textbf{)}}$$

$$\frac{e : \texttt{@ClassVal(}\nu\texttt{)} \quad p : \texttt{@ArrayLen(}\pi\texttt{)}}{e\texttt{.getConstructor(}p\texttt{)} : \textbf{@MethodVal(}\texttt{cn=}\nu\texttt{,}mn = \texttt{"<init>"}\texttt{,}np = \pi\textbf{)}}$$

Figure 21.3: Example inference rules for @ClassVal, @ClassBound, and @MethodVal. Additional rules exist for expressions with similar semantics but that call methods with different names or signatures.

Subtyping rules

Figure 21.2 show part of the type hierarchy of the MethodVal type system. `@MethodVal(classname=CA, methodname=MA, params=PA)` is a subtype of `@MethodVal(classname=CB, methodname=MB, params=PB)` if

$$\forall \text{indexes} i \exists \text{an index } j : CA[i] = CB[j], MA[i] = MA[j], and PA[i] = PB[j]$$

where CA, MA, and PA are lists of equal size and CB, MB, and PB are lists of equal size.

21.1.3 MethodVal and ClassVal inference

The developer rarely has to write `@ClassVal` or `@MethodVal` annotations, because the Checker Framework infers them according to Figure 21.3. Most readers can skip this section, which explains the inference rules.

The ClassVal Checker infers the exact class name (`@ClassVal`) for a `Class` literal (`C.class`), and for a static method call (e.g., `Class.forName(arg)`, `ClassLoader.loadClass(arg)`, ...) if the argument is a statically computable expression. In contrast, it infers an upper bound (`@ClassBound`) for instance method calls (e.g., `obj.getClass()`).

The MethodVal Checker infers `@MethodVal` annotations for `Method` and `Constructor` types that have been created using a method call to Java's Reflection API:

- `Class.getMethod(String name, Class<?>... paramTypes)`
- `Class.getConstructor(Class<?>... paramTypes)`

Note that an exact class name is necessary to precisely resolve reflectively-invoked constructors since a constructor in a subclass does not override a constructor in its superclass. This means that the MethodVal Checker does not infer a `@MethodVal` annotation for `Class.getConstructor` if the type of that class is `@ClassBound`. In contrast, either an exact class name or a bound is adequate to resolve reflectively-invoked methods because of the subtyping rules for overridden methods.

21.2 Reflection resolution example

Consider the following example, in which the Nullness Checker employs reflection resolution to avoid issuing a false positive warning.

```
public class LocationInfo {
    @NonNull Location getCurrentLocation() {  ...  }
}

public class Example {
    LocationInfo privateLocation = ... ;
    String getCurrentCity() throws Exception {
        Method getCurrentLocationObj = LocationInfo.class.getMethod("getCurrentLocation");
        Location currentLocation = (Location) getCurrentLocationObj.invoke(privateLocation);
        return currentLocation.nameOfCity();
    }
}
```

When reflection resolution is not enabled, the Nullness Checker uses conservative annotations on the `Method.invoke` method signature:

```
@Nullable Object invoke(@NonNull Object recv, @NonNull Object ...  args)
```

This causes the Nullness Checker to issue the following warning even though `currentLocation` cannot be null.

```
error: [dereference.of.nullable] dereference of possibly-null reference currentLocation
        return currentLocation.nameOfCity();
               ^
1 error
```

When reflection resolution is enabled, the MethodVal Checker infers that the `@MethodVal` annotation for `getCurrentLocationObj` is:

```
@MethodVal(className="LocationInfo", methodName="getCurrentLocation", params=0)
```

Based on this `@MethodVal` annotation, the reflection resolver determines that the reflective method call represents a call to `getCurrentLocation` in class `LocationInfo`. The reflection resolver uses this information to provide the following precise procedure summary to the Nullness Checker, for this call site only:

```
@NonNull Object invoke(@NonNull Object recv, @Nullable Object ...  args)
```

Using this more precise signature, the Nullness Checker does not issue the false positive warning shown above.

Chapter 22

Subtyping Checker

The Subtyping 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 Subtyping Checker can accommodate all of the type system enhancements that can be declaratively specified (see Chapter 29). 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 25.4) and qualifier polymorphism (Section 24.2).

The Subtyping Checker is also useful to type system designers who wish to experiment with a checker before writing code; the Subtyping 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 Subtyping Checker if you do not need typestate analysis. For more details (including a definition of "typestate"), see Chapter 23.1. See Section 31.6.2 for a simpler alternative.

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 29.

22.1 Using the Subtyping Checker

The Subtyping Checker is used in the same way as other checkers (using the `-processor org.checkerframework.common.subtyping.SubtypingChecker` 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. For example,
  ```
  javac -processor org.checkerframework.common.subtyping.SubtypingChecker
        -Aquals=myproject.qual.MyQual,myproject.qual.OtherQual MyFile.java ...
  ```
  It serves the same purpose as the `@TypeQualifiers` annotation used by other checkers (see section 29.7).
  The annotations listed in `-Aquals` must be accessible to the compiler during compilation in the classpath. In other words, they must already be compiled (and, typically, be on the javac bootclasspath) before you run the Subtyping Checker with `javac`. It is not sufficient to supply their source files on the command line.

To suppress a warning issued by the Subtyping Checker, use a `@SuppressWarnings` annotation, with the argument being the unqualified, uncapitalized name of any of the annotations passed to `-Aquals`. This will suppress all warnings, regardless of which of the annotations is involved in the warning. (As a matter of style, you should choose one of the annotations as your `@SuppressWarnings` key and stick with it for that entire type hierarchy.)

22.2 Subtyping 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 Subtyping Checker for the `Encrypted` type system, follow three steps.

1. Define two annotations for the `Encrypted` and `PossiblyUnencrypted` qualifiers:

```
package myqual;

import java.lang.annotation.Target;
import java.lang.annotation.ElementType;
import org.checkerframework.framework.qual.SubtypeOf;
import org.checkerframework.framework.qual.TypeQualifier;

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

```
package myqual;

import java.lang.annotation.Target;
import java.lang.annotation.ElementType;
import org.checkerframework.framework.qual.DefaultQualifierInHierarchy;
import org.checkerframework.framework.qual.SubtypeOf;
import org.checkerframework.framework.qual.TypeQualifier;

/**
 * Denotes that the representation of an object might not be encrypted.
 */
@TypeQualifier
@DefaultQualifierInHierarchy
@SubtypeOf({})
@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})
public @interface PossiblyUnencrypted {}
```

   Don't forget to compile these classes:

```
$ javac myqual/Encrypted.java myqual/PossiblyUnencrypted.java
```

   The resulting `.class` files 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 (say, in file `YourProgram.java`):

```
import myqual.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 Subtyping Checker, specifying the `@Encrypted` annotation using the `-Aquals` option. You should add the `Encrypted` classfile to the processor classpath:

```
$ javac -processorpath myqualpath -processor org.checkerframework.common.subtyping.SubtypingChecker \
        -Aquals=myqual.Encrypted,myqual.PossiblyUnencrypted YourProgram.java

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

Chapter 23

Third-party checkers

The Checker Framework has been used to build other checkers that are not distributed together with the framework. This chapter mentions just a few of them. They are listed in chonological order; older ones appear first and newer ones appear last.

They are externally-maintained, so if you have problems or questions, you should contact their maintainers rather than the Checker Framework maintainers.

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

23.1 Typestate checkers

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 Subtyping Checker to create a type-checker, you can extend a typestate checker to create a type-checker that supports typestate analysis. An extensible typestate analysis by Adam Warski that builds on the Checker Framework is available at `http://www.warski.org/typestate.html`.

23.1.1 Comparison to flow-sensitive type refinement

The Checker Framework's flow-sensitive type refinement (Section 25.4) 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. (Unused variables (Section 25.6) 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).

23.2 Units and dimensions checker

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

Unlike the Units Checker that is distributed with the Checker Framework (see Section 15), this checker includes dynamic checks and permits annotation arguments that are Java expressions. This added flexibility, however, requires that you use a special version both of the Checker Framework and of the javac compiler.

23.3 Thread locality checker

Loci, a checker for thread locality, is available at `http://www.it.uu.se/research/upmarc/loci/`. Developer resources are available at the project page `http://java.net/projects/loci/`.

23.4 Safety-Critical Java checker

A checker for Safety-Critical Java (SCJ, JSR 302) is available at `http://sss.cs.purdue.edu/projects/oscj/checker/checker.html`. Developer resources are available at the project page `http://code.google.com/p/scj-jsr302/`.

23.5 Generic Universe Types checker

A checker for Generic Universe Types, a lightweight ownership type system, is available from `https://ece.uwaterloo.ca/~wdietl/ownership/`.

23.6 EnerJ checker

A checker for EnerJ, an extension to Java that exposes hardware faults in a safe, principled manner to save energy with only slight sacrifices to the quality of service, is available from `http://sampa.cs.washington.edu/research/approximation/enerj.html`.

23.7 CheckLT taint checker

CheckLT uses taint tracking to detect illegal information flows, such as unsanitized data that could result in a SQL injection attack. CheckLT is available from `http://checklt.github.io/`.

23.8 SPARTA information flow type-checker for Android

SPARTA is a security toolset aimed at preventing malware from appearing in an app store. SPARTA provides an information-flow type-checker that is customized to Android but can also be applied to other domains. The SPARTA toolset is available from `http://types.cs.washington.edu/sparta/`. The paper "Collaborative verification of information flow for a high-assurance app store" appeared in CCS 2014.

Chapter 24

Generics and polymorphism

This chapter describes support for Java generics (also known as "parametric polymorphism") and polymorphism over type qualifiers.

The Checker Framework currently supports two schemes for polymorphism over type qualifiers.

Section 24.2 describes the original scheme, which uses method-based annotations that are meta-annotated with `@PolymorphicQualifier`.

Section 24.3 describes the qualifier parameters scheme, in which qualifier parameters are specified for classes and methods similarly to Java generics. The qualifier parameter scheme is more powerful than the original approach, but is currently (as of February 2015) experimental and incurs a 50% performance penalty. Currently, only the Tainting Checker (Chapter 8) and the Regex Checker (Chapter 9) support qualifier parameters.

24.1 Generics (parametric polymorphism or type polymorphism)

The Checker Framework fully supports type-qualified Java generic types and methods (also known in the research literature as "parametric polymorphism"). When instantiating a generic type, clients supply the qualifier along with the type argument, as in `List<@NonNull String>`.

24.1.1 Raw types

Before running any pluggable type-checker, we recommend that you eliminate raw types from your code (e.g., your code should use `List<...>` as opposed to `List`). Your code should compile without warnings when using the standard Java compiler and the `-Xlint:unchecked -Xlint:rawtypes` command-line options. Using generics helps prevent type errors just as using a pluggable type-checker does, and makes the Checker Framework's warnings easier to understand.

If your code uses raw types, then the Checker Framework will do its best to infer the Java type parameters and the type qualifiers. If it infers imprecise types that lead to type-checking warnings elsewhere, then you have two options. You can convert the raw types such as `List` to parameterized types such as `List<String>`, or you can supply the `-AignoreRawTypeArguments` command-line option. That option causes the Checker Framework to ignore all subtype tests for type arguments that were inferred for a raw type.

24.1.2 Restricting instantiation of a generic class

When you define a generic class in Java, the `extends` clause of the generic type parameter (known as the "upper bound") requires that the corresponding type argument must be a subtype of the bound. For example, given the definition `class G<T extends Number> {...}`, the upper bound is `Number` and a client can instantiate it as `G<Number>` or `G<Integer>` but not `G<Date>`.

You can write a type qualifier on the `extends` clause to make the upper bound a qualified type. For example, you can declare that a generic list class can hold only non-null values:

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

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

That is, in the above example, all arguments that replace `T` in `MyList<T>` must be subtypes of `@NonNull Object`.

Conceptually, each generic type parameter has two bounds — a lower bound and an upper bound — and at instantiation, the type argument must be within the bounds. Java only allows you to specify the upper bound; the lower bound is implicitly the bottom type `void`. The Checker Framework gives you more power: you can specify both an upper and lower bound for type parameters and wildcards. For the upper bound, write a type qualifier on the `extends` clause, and for the lower bound, write a type qualifier on the type variable.

```
class MyList<@LowerBound T extends @UpperBound Object> { ... }
```

For a concrete example, recall the type system of the Regex Checker (see Figure 9, page 61) in which `@Regex(0)` :> `@Regex(1)` :> `@Regex(2)` :> `@Regex(3)` :> ....

```
class MyRegexes<@Regex(5) T extends @Regex(1) String> { ... }

MyRegexes<@Regex(0) String> mu;   // error - @Regex(0) is not a subtype of @Regex(1)
MyRegexes<@Regex(1) String> m1;   // OK
MyRegexes<@Regex(3) String> m3;   // OK
MyRegexes<@Regex(5) String> m5;   // OK
MyRegexes<@Regex(6) String> m6;   // error - @Regex(6) is not a supertype of @Regex(5)
```

The above declaration states that the upper bound of the type variable is `@Regex(1) String` and the lower bound is `@Regex(5) void`. That is, arguments that replace `T` in `MyList<T>` must be subtypes of `@Regex(1) String` and supertypes of `@Regex(5) void`. Since `void` cannot be used to instantiate a generic class, `MyList` may be instantiated with `@Regex(1) String` through `@Regex(5) String`.

To specify an exact bound, place the same annotation on both bounds. For example:

```
class MyListOfNonNulls<@NonNull T extends @NonNull Object> { ... }
class MyListOfNullables<@Nullable T extends @Nullable Object> { ... }

MyListOfNonNulls<@NonNull Number> v1;      // OK
MyListOfNonNulls<@Nullable Number> v2;     // error
MyListOfNullables<@NonNull Number> v4;     // error
MyListOfNullables<@Nullable Number> v3;    // OK
```

It is an error if the lower bound is not a subtype of the upper bound.

```
class MyClass<@Nullable T extends @NonNull Object>  // error @Nullable is not a supertype of @NonNull
```

Defaults

If the `extends` clause is omitted, then the upper bound defaults to `@TopType Object`. If no type annotation is written on the type parameter name, then the lower bound defaults to `@BottomType void`. If the `extends` clause is written but contains no type qualifier, then the normal defaulting rules apply to the type in the `extends` clause (see Section 25.3.2).

These rules mean that even though in Java the following two declarations are equivalent:

```
class MyClass<T>
class MyClass<T extends Object>
```

they may specify different type qualifiers on the upper bound, depending on the type system's defaulting rules.

24.1.3 Type annotations on a use of a generic type variable

A type annotation on a use of 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`. This feature is only rarely used.

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

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

The type annotation does not restrict how `MyClass2` may be instantiated. 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`.

24.1.4 Annotations on wildcards

At an instantiation of a generic type, a Java wildcard indicates that some constraints are known on the type argument, but the type argument is not known exactly. For example, you can indicate that the type parameter for variable `ls` is some unknown subtype of `CharSequence`:

```
List<? extends CharSequence> ls;
ls = new ArrayList<String>();      // OK
ls = new ArrayList<Integer>();     // error - Integer is not a subtype of CharSequence
```

For more details about wildcards, see the Java tutorial on wildcards or JLS §4.5.1.

You can write a type annotation on the bound of a wildcard:

```
List<? extends @NonNull CharSequence> ls;
ls = new ArrayList<@NonNull String>();      // OK
ls = new ArrayList<@Nullable String>();     // error - @Nullable is not a subtype of @NonNull
```

Conceptually, every wildcard has two bounds — an upper bound and a lower bound. Java only permits you to write the upper bound (with `<? extends SomeType>`) or the lower bound (with `<? super OtherType>`), but not both; the unspecified bound is implicitly the top type `Object` or the bottom type `void`. The Checker Framework is more flexible: it lets you simultaneously write annotations on both the top and the bottom type. To annotate the implicit bound, write the type annotation before the `?`. For example:

```
List<@LowerBound ? extends @UpperBound CharSequence> lo;
List<@UpperBound ? super @NonNull Number> ls;
```

For an unbounded wildcard (`<?>`, with neither bound specified), the annotation in front of a wildcard applies to both bounds. The following three declarations are equivalent (except that you cannot write the bottom type `void`; note that `Void` does not denote the bottom type):

```
List<@NonNull ?> lnn;
List<@NonNull ? extends @NonNull Object> lnn;
List<@NonNull ? super @NonNull void> lnn;
```

Note that the annotation in front of a type parameter always applies to its lower bound, because type parameters can only be written with `extends` and never `super`.

The defaulting rules for wildcards also differ from those of type parameters (see Section 25.3.4).

24.1.5 Examples of qualifiers on a type parameter

Recall that `@Nullable X` is a supertype of `@NonNull X`, for any *X*. Most of of the following types mean different things:

```
class MyList1<@Nullable T> { ... }
class MyList1a<@Nullable T extends @Nullable Object> { ... } // same as MyList1
class MyList2<@NonNull T extends @NonNull Object> { ... }
class MyList2a<T extends @NonNull Object> { ... } // same as MyList2
class MyList3<T extends @Nullable Object> { ... }
```

`MyList1` and `MyList1a` must be instantiated with a nullable type. The implementation of `MyList1` must be able to consume (store) a null value and produce (retrieve) a null value.

`MyList2` and `MyList2a` must be instantiated with non-null type. The implementation of `MyList2` has to account for only non-null values — it does not have to account for consuming or producing null.

`MyList3` may be instantiated either way: with a nullable type or a non-null type. The implementation of `MyList3` must consider that it may be instantiated either way — flexible enough to support either instantiation, yet rigorous enough to impose the correct constraints of the specific instantiation. It must also itself comply with the constraints of the potential instantiations.

One way to express the difference among `MyList1`, `MyList2`, and `MyList3` is by comparing what expressions are legal in the implementation of the list — that is, what expressions may appear in the ellipsis in the declarations above, such as inside a method's body. Suppose each class has, in the ellipsis, these declarations:

```
T t;
@Nullable T nble;      // Section "Type annotations on a use of a generic type variable", below,
@NonNull T nn;         // further explains the meaning of "@Nullable T" and "@NonNull T".
void add(T arg) { }
T get(int i) { }
```

Then the following expressions would be legal, inside a given implementation — that is, also within the ellipses. (Compilable source code appears as file `checker-framework/checker/tests/nullness/generics/GenericsExample.java`.)

	MyList1	MyList2	MyList3
t = null;	OK	error	error
t = nble;	OK	error	error
nble = null;	OK	OK	OK
nn = null;	error	error	error
t = this.get(0);	OK	OK	OK
nble = this.get(0);	OK	OK	OK
nn = this.get(0);	error	OK	error
this.add(t);	OK	OK	OK
this.add(nble);	OK	error	error
this.add(nn);	OK	OK	OK

The differences are more significant when the qualifier hierarchy is more complicated than just `@Nullable` and `@NonNull`.

24.1.6 Covariant type parameters

Java types are *invariant* in their type parameter. This means that `A<X>` is a subtype of `B<Y>` only if `X` is identical to `Y`. For example, `ArrayList<Number>` is a subtype of `List<Number>`, but neither `ArrayList<Integer>` nor `List<Integer>` is a subtype of `List<Number>`. (If they were, there would be a type hole in the Java type system.) For the same reason, type parameter annotations are treated invariantly. For example, `List<@Nullable String>` is not a subtype of `List<String>`.

When a type parameter is used in a read-only way — that is, when values of that type are read but are never assigned — then it is safe for the type to be *covariant* in the type parameter. Use the `@Covariant` annotation to indicate this. When a type parameter is covariant, two instantiations of the class with different type arguments have the same subtyping relationship as the type arguments do.

For example, consider `Iterator`. Its elements can be read but not written, so `Iterator<@Nullable String>` can be a subtype of `Iterator<String>` without introducing a hole in the type system. Therefore, its type parameter is annotated with `@Covariant`. The first type parameter of `Map.Entry` is also covariant. Another example would be the type parameter of a hypothetical class `ImmutableList`.

The `@Covariant` annotation is trusted but not checked. If you incorrectly specify as covariant a type parameter that that can be written (say, the class performs a `set` operation or some other mutation on an object of that type), then you have created an unsoundness in the type system. For example, it would be incorrect to annotate the type parameter of `ListIterator` as covariant, because `ListIterator` supports a `set` operation.

24.1.7 Method type argument inference and type qualifiers

Sometimes method type argument inference does not interact well with type qualifiers. In such situations, you might need to provide explicit method type arguments, for which the syntax is as follows:

```
Collections.</*@MyTypeAnnotation*/ Object>sort(l, c);
```

This uses Java's existing syntax for specifying a method call's type arguments.

24.2 Qualifier polymorphism

This section describes the original Checker Framework scheme for qualifier polymorphism. Section 24.3 describes an alternative scheme that uses qualifier parameters.

The Checker Framework supports type *qualifier* polymorphism for methods, which permits a single method to have multiple different qualified type signatures. This is similar to Java's generics, but is used in situations where you cannot use Java generics. If you can use generics, you typically do not need to use a polymorphic qualifier such as `@PolyNull`.

To *use* a polymorphic qualifier, just write it on a type. For example, you can write `@PolyNull` anywhere in a method that you would write `@NonNull` or `@Nullable`. A polymorphic qualifier can be used on a method signature or body. It may not be used on a class or field.

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 24.2.1.

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 the same as Java's rule for resolving overloaded methods.

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, ElementType.TYPE_PARAMETER})
public @interface PolyNull { }
```

See Section 24.2.5 for a way you can sometimes avoid defining a new polymorphic qualifier.

24.2.1 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

```
// Returns null if either argument is null.
@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);
```

At a call site, the most specific applicable signature is selected.

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.

24.2.2 Relationship to subtyping and generics

Qualifier polymorphism has the same purpose and plays the same role as Java's generics. If a method is written using generics, it usually does not need qualifier polymorphism. If you have legacy code that is not written generically, and you cannot change it to use generics, then you can use qualifier polymorphism to achieve a similar effect, with respect to type qualifiers only. The base Java types are still treated non-generically.

Why not use ordinary subtyping to handle qualifier polymorphism? Ordinarily, when you want a method to work on multiple types, you can just use Java's subtyping. For example, the `equals` method is declared to take an `Object` as its first formal parameter, but it can be called on a `String` or a `Date` because those are subtypes of `Object`.

In most cases, the same subtyping mechanism works with type qualifiers. `String` is a supertype of `@Interned String`, so a method `toUpperCase` that is declared to take a `String` parameter can also be called on a `@Interned String` argument.

You use qualifier polymorphism in the same cases when you would use Java's generics. (If you can use Java's generics, then that is often better and you don't also need to use qualifier polymorphism.) One example is when you want a method to operate on collections with different types of elements. Another example is when you want two different formal parameters to be of the same type, without constraining them to be one specific type.

24.2.3 Using 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());
}
```

24.2.4 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 index, in an array, of a given element:

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

If `@PolyNull` were replaced with either `@Nullable` or `@NonNull`, then one of these safe client calls would be rejected:

```
  @Nullable Object[] a1;
  @NonNull Object[] a2;

  indexOf(a1, someObject);
  indexOf(a2, someObject);
```

Of course, it would be better style to use a generic method, as in either of these signatures:

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

The examples in this section use arrays, but analogous collection examples exist.

These examples show that use of a single polymorphic qualifier may be necessary in legacy code, but can often be avoided by use of better code style.

24.2.5 The `@PolyAll` qualifier applies to every type system

Each type system has its own polymorphic type qualifier. If some method is qualifier-polymorphic over every type qualifier hierarchy, then it is tedious, and leads to an explosion in the number of type annotations, to place every `@Poly*` qualifier on that method.

For example, a method that only performs == on array elements will work no matter what the array's element types are:

```
/** Searches for the first occurrence of the given element in the array,
 *  testing for equality using == (not the equals method). */
public static int indexOfEq(@PolyAll Object[] a, @Nullable Object elt) {
  for (int i=0; i<a.length; i++)
    if (elt == a[i])
      return i;
  return -1;
}
```

The `@PolyAll` qualifier takes an optional argument so that you can specify multiple, independent polymorphic type qualifiers. For example, the method also works no matter what the type argument on the second argument is. This signature is overly restrictive:

```
/** Returns true if the arrays are elementwise equal,
 *  testing for equality using == (not the equals method). */
public static int eltwiseEqualUsingEq(@PolyAll Object[] a, @PolyAll Object elt) {
  for (int i=0; i<a.length; i++)
    if (elt != a[i])
      return false;
  return true;
}
```

That signature requires the element type annotation to be identical for the two arguments. For example, it forbids this invocation:

```
  @Mutable Object[] x;
@Immutable Object   y;
... indexOf(x, y) ...
```

A better signature lets the two arrays' element types vary independently:

```
public static int eltwiseEqualUsingEq(@PolyAll(1) Object[] a, @PolyAll(2) Object elt)
```

Note that in this case, the `@Nullable` annotation on `elt`'s type is no longer necessary, since it is subsumed by `@PolyAll`.

The `@PolyAll` annotation at a location *l* applies to every type qualifier hierarchy for which no explicit qualifier is written at location *l*. For example, a declaration like `@PolyAll @NonNull Object elt` is polymorphic over every type system *except* the nullness type system, for which the type is fixed at `@NonNull`. That would be the proper declaration for `elt` if the body had used `elt.equals(a[i])` instead of `elt == a[i]`.

24.3 Qualifier parameters

This section describes qualifier parameters which is the new, more-powerful qualifier polymorphism scheme. As of February 2015, only the Tainting Checker (Chapter 8) and the Regex Checker (Chapter 9) support qualifier parameters. Other checkers with qualifier polymorphism support use the original qualifier polymorphism scheme (Section 24.2).

Qualifier parameters provide a way for you to re-use the same code with different type qualifiers in a type-safe manner.

Qualifier parameters are very similar to Java generics, so if you understand the benefits of generics and how to use them, you will find qualifier parameters natural. Both mechanisms are used on classes and methods where different instances of the class have different types. Without generics or qualifier parameters, the types of the members would have to be overly general, which would cause information loss, compiler warnings, the need for casts, and potentially run-time errors. Generics parameterize a class or method with a *type*, so that a client can specialize the definition with a type as in `List<Integer>` or `List<String>`. By contrast, qualifier parameters enable a client to specialize the definition with just a *qualifier* as in `MyClass⟪@Regex⟫` or `MyClass⟪@NonNull⟫`.

24.3.1 Motivation for qualifier parameters

As an example of a problem that qualifier parameters solve, consider the `Holder` class below. In some uses of `Holder`, the `item` field holds a `@Tainted String` value, and in other uses of `Holder`, the `item` field holds an `@Untainted String` value. The only declaration of `item` that is consistent with all uses is `@Tainted String`, which is a supertype of `@Untainted String`. When an `@Untainted String` value is put in a `Holder`, a cast is required when the value is later retrieved.

```
class Holder {
    @Tainted String item;   // overly-general declaration, leads to casts
}

// taintedHolder can hold both @Tainted and @Untainted values
Holder taintedHolder = new Holder();
taintedHolder.item = getTaintedValue();
@Tainted String taintedString = taintedHolder.item;  // OK; type-checks with the Tainting Checker.

// The programmer intends untaintedHolder to hold only @Untainted values
Holder untaintedHolder = new Holder();
untaintedHolder.item = getUntaintedValue();
@Untainted String untaintedString = untaintedHolder.item;  // safe code, but Tainting Checker compile-time error.
// A cast makes the assignment type-check, but casts are unsound and error-prone.
String untaintedString = (@Untainted untaintedString) untaintedHolder.item;
taintedHolder.item = getTaintedValue(); // An error that we would like the type sysetm to catch
```

Qualifier parameters allow sound type-checking of this code without the use of casts.

24.3.2 Overview of qualifier parameters

These following examples add qualifier parameters to `Holder` from Section 24.3.1 to allow sound type-checking.

For clarity, this section displays qualifier parameters using an idealized syntax using double angle brackets, ⟪...⟫. Note that this is not the actual syntax you will use in source code, which is described in Section 24.3.4.

In the qualifier parameter system, a class can declared to have one or more qualifier parameters. For example, a qualifier parameter can be added to the `Holder` class:

```
class Holder ⟪Q⟫ {

}
```

This declares that `Holder` takes one qualifier parameter, named `Q`.

`Q` can be referenced inside the `Holder` class. In the following, `item` will have the same qualifier that `Holder` is instantiated with:

```
class Holder ⟪Q⟫ {
    @Q String item;
}
```

References and instantiations of `Holder` specify a qualifier argument for its parameter `Q`.

```
Holder⟪Q=@Tainted⟫ taintedHolder;
Holder⟪Q=@Untainted⟫ untaintedHolder;
```

Qualifier parameters permit instantiating a class with the appropriate type qualifier rather than relying on an overly-general declaration. Therefore, the following code type-checks without casts:

```
Holder⟪Q=@Tainted⟫ taintedHolder = new Holder⟪Q=@Tainted⟫();
@Tainted String s = holder.item;

Holder⟪Q=@Untainted⟫ untaintedHolder = new Holder⟪Q=@Untainted⟫();
@Untainted String s = holder.item;
```

Like generics, two classes with different qualifier parameters have no subtyping relationship:

```
taintedHolder = untaintedHolder;     // Error: not a subtype
untaintedHolder = taintedHolder;     // Error: not a subtype
Holder⟪Q=@Tainted⟫ taintedHolder2;
taintedHolder = taintedHolder2;      // OK:  the qualifier argument is the same for both
```

24.3.3 Qualifier parameter wildcards

As with Java generics, wildcard extends and super bounds may be used. Wildcards create a subtyping relationship between classes with qualifier parameters. See the Java tutorial at `http://docs.oracle.com/javase/tutorial/java/generics/subtyping.html` for more information on subtyping relationships with wildcards.

```
Holder⟪Q=@Tainted⟫ holder;
Holder⟪Q=? extends @Tainted⟫ holderExtends;
Holder⟪Q=? super @Tainted⟫ holderSuper;

holder = holderExtends;    // Error: not a subtype
holderExtends = holder;    // OK

holder = holderSuper;    // Error: not a subtype
holderSuper = holder;    // OK
```

For soundness, when a class is parameterized with a wildcard, members of a qualified class that use the parameter as their type have restrictions on their use, just as in Java. In particular, a member of a qualified class with an extends-bounded wildcard may only be set to `null`. A member of a qualified class with a super-bounded wildcard will always have the top type when accessed.

```
Holder⟪Q=? extends @Untainted⟫ holderExtends;
@Untainted String s1 = holderExtends.item;  // OK
holderExtends.item = getTaintedString();    // Error: only null can be assigned to item

Holder⟪Q=? super @Untainted⟫ holderSuper;
@Untainted String s2 = holderSuper.item;    // Error: item has the top type
holderSuper.item = getUntaintedString();    // OK
```

24.3.4 Syntax of qualifier parameters

The examples in Sections 24.3.2–24.3.3 used double angle brackets, ⟪. . .⟫, for qualifier parameter declarations and qualifier arguments. In real source code, qualifier parameter declarations and uses, and qualifier arguments, are specified via Java annotations.

- To declare a qualifier parameter, use `@Class`*`Typesystem`*`Param` or `@Method`*`Typesystem`*`Param` and give a name for the parameter, as in `@ClassTaintingParam("main")`.
- To use a qualifier parameter, write `@Var` and indicate the parameter being used, as in `@Var(arg="main")`.
- To supply a qualifier argument, write the argument annotation (e.g., `@Tainted`), but supply a `param` argument, as in `@Tainted("main")` which means that `@Tainted` is the argument to the parameter named `main`.

These annotations are summarized in Figure 24.1 and are more fully explained below.

Each type system that supports qualifier parameters has its own copy of these annotations. The functionality of the annotations is the same, but since a java file might be annotated with annotations for multiple type systems, i.e. have annotations for both the Regex and the Tainting checker, there must be a different copy of each annotation so that the Checker Framework can determine the checker that an annotation belongs to.

	Generic Equivalent	Idealized Syntax	Actual Syntax
Declare a class parameter	`class Holder<T> {}`	`class Holder⟪Q⟫ {}`	`@ClassTaintingParam("Q") class Holder {}`
Declare a method parameter	`<T> void do() {}`	`⟪V⟫ void do() {}`	`@MethodTaintingParam("V") void do() {}`
Instantiate (supply an argument)	`Holder<String>`	`Holder⟪Q=@Tainted⟫`	`@Tainted(param="Q") Holder`
Use a parameter	`<T> void do(T t) {}`	`⟪V⟫ void do(@V Object o) {}`	`@MethodTaintingParam("V") void do(@Var(arg="V") Object o) {}`
Use a parameter as an argument	`<T> void do(List<T> t) {}`	`⟪V⟫ void do(Holder⟪Q=@V⟫ h) {}`	`@MethodTaintingParam("V") void do(@Var(arg="V" param="Q") Holder o) {}`
Instantiate without constraints	`Holder<?>`	`Holder⟪Q=?⟫`	`@Wild(param="Q") Holder`
Instantiate with upper bound	`Holder<? extends Object>`	`Holder⟪Q=? extends @Tainted⟫`	`@Tainted(param="Q", wildcard=Wildcard.EXTENDS) Holder`
Instantiate with lower bound	`Holder<? super Object>`	`Holder⟪Q=? super @Tainted⟫`	`@Tainted(param="Q", wildcard=Wildcard.SUPER) Holder`

Figure 24.1: Comparison of the syntax of Java generics, the idealized syntax used in Sections 24.3.2–24.3.3, and the actual syntax used in Java source code.

`@ClassTaintingParam` Declares a qualifier parameter for a class.

```
// Equivalent to
class Holder ⟪Q⟫ {

}

// Declare a parameter "main"
@ClassTaintingParam("main")
class Holder {

}

// The parameter "main" can now be set
@Tainted(param="main") Holder h;
```

`@MethodTaintingParam` Declares a qualifier parameter for a method.

Qualifier arguments to a method are never specified explicitly; they are inferred by the Checker Framework based on the parameters passed to the method invocation. Unlike Java generics, there is no way to explicitly specify method qualifier parameters on an invocation.

```
class Util {

  // Declare a method parameter.
  @MethodTaintingParam("meth")
  public static @Var("meth") String id(@Var("meth") String in) {
      return in;
  }
}

// Qualifier arguments are inferred.
@Untainted String untainted = Util.id(getUntaintedString());
```

`@Var` Declares a use of a qualifier parameter. The `arg` field specifies which qualifier parameter in the surrounding scope the type should get its value from. For example:

```
// Equivalent to
class Holder ⟪Q⟫ {
  @Q String item;
}
```

```
// Declare a parameter
@ClassTaintingParam ("main")
class Holder {
  // item will have the qualifier that Holder is instantiated with
  @Var(arg="main") String item;
}

@Tainted(param="main") Holder h1 = new @Tainted(param="main") Holder();
@Tainted String value1 = h1.item;

@Untainted(param="main") Holder h2 = new @Untainted(param="main") Holder();
@Untainted String value1 = h2.item;
```

The `"param"` field specifies that the value of the qualifier parameter specified by `"arg"` should be used as the parameter to another qualifier type. For example:

```
// Equivalent to
class Holder ⟪Q⟫ {
  @Q String item;
  Holder⟪Q=@Q⟫ nestedHolder;
}

@ClassTaintingParam ("main")
class Holder {
  // item will have the qualifier that Holder is instantiated with
  @Var(arg="main") String item;

  // nestedHolder will be instantiated with the same qualifier as the
  // enclosing "main" parameter
  @Var(arg="main", param="main") Holder nestedHolder;
}

@Tainted(param="main") Holder h1 = new @Tainted(param="main") Holder();
@Tainted(param="main") Holder nestedHolder = h1;
@Tainted String value1 = h1.nestedHolder.item;

@Untainted(param="main") Holder h2 = new @Untainted(param="main") Holder();
@Untainted(param="main") Holder nestedHolder2 = h2;
@Untainted String value1 = h2.nestedHolder.item;
```

`@Tainted` When the `param` field is not set, this annotation behaves as described in Chapter 8 and indicates that the value is tainted. For example:

```
// The value should be considered tainted
@Tainted String tainted  = getTaintedString();
```

When the `param` param field is set, the annotation indicates that the value of the `@Tainted` qualifier should be used as the qualifier argument to the class that it annotates. For example:

```
// Equivalent to Holder⟪@Tainted⟫ holder

// This declares a Holder object, whose Tainting qualifier parameter is set to @Tainted.
// Holder must have been declared to have a Tainting qualifier parameter
```

```
// by using the @ClassTaintingParam annotation.
@Tainted(param="main") Holder holder;
```

The `wildcard` field can be set to a `Wildcard` value. This allows qualifier parameters to act like wildcards.

```
// Equivalent to Holder⟪? extends @Untainted⟫

// Instantiate Holder with a wildcard parameter.
@Untainted(param="main", wildcard=Wildcard.EXTENDS) Holder extends;

// OK because of the extends bound
extendsHolder = new @Untainted(param="main") Holder();
// Error: the new Holder is not a subtype of extendsHolder
extendsHolder = new @Untainted(param="main") Holder();
```

`@Untainted` `@Untainted` behaves the same as `@Tainted` but for untainted values.

`@Wild` Declares that a class has an unknown qualifier parameter. This is useful in cases where the qualifier parameter in the class is not used or is used in very limited ways.

```
// Equivalent to
Holder⟪?⟫ h1 = new Holder⟪@Untainted⟫();

@Wild(param="main") Holder h1 = new @Untainted(param="main") Holder;

// Error: item is not guaranteed to be an @Untainted value.
@Untainted String s1 = h1.item;
```

`@PolyTainted` Enables method qualifier polymorphism. When the field `param` is not set, `@PolyTainted` behaves as described Section 24.2. For example:

```
class Util {
  static @PolyTainted String id(@PolyTainted String in) {
      return in;
  }
}

@Untainted String s = Util.id(getUntaintedString());  // OK
```

The field `param` can be used to specify that the inferred qualifier parameter should be used as an argument to another parameterized type. In this mode `@PolyTainted` is a shorthand for a combination of `@MethodTaintingParam` and `@Var`. For example:

```
class Util {
  static @PolyTainted(param="main") Holder id(@PolyTainted(param="main") Holder in) {
      return in;
  }
}

// Equivalent to this code
@MethodTaintingParam("meth")
public static @Var(arg="meth", param="main") Holder id(@Var(arg="meth", param="main) Holder in) {
    return in;
}
```

24.3.5 Primary qualifiers

Type system specific annotations, like `@Tainted` or `@Regex`, have dual uses in the qualifier parameter system. When their `"param"` field is set, they are used as a argument to a qualifier parameter.

When their `"param"` field is not set, they apply directly to a type and not to any qualifier parameters of the type. We call the qualifier that applies directly to a type the primary qualifier. For example an `@Tainted String` is a String with a tainted value and its primary qualifier is `@Tainted`.

`@Var` can also be used to set primary qualifiers by omitting the `"param"` field on the annotation.

Chapter 25

Advanced type system features

This chapter describes features that are automatically supported by every checker written with the Checker Framework. You may wish to skim or skip this chapter 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.

25.1 Invariant array types

Java's type system is unsound with respect to arrays. That is, the Java type-checker approves code that is unsafe and will cause a run-time crash. Technically, the problem is that Java has "covariant array types", such as treating `String[]` as a subtype of `Object[]`. Consider the following example:

```
String[] strings = new String[] {"hello"};
Object[] objects = strings;
objects[0] = new Object();
String myString = strs[0];
```

The above code puts an `Object` in the array `strings` and thence in `myString`, even though `myString = new Object()` should be, and is, rejected by the Java type system. Java prevents corruption of the JVM by doing a costly run-time check at every array assignment; nonetheless, it is undesirable to learn about a type error only via a run-time crash rather than at compile time.

When you pass the `-AinvariantArrays` command-line option, the Checker Framework is stricter than Java, in the sense that it treats arrays invariantly rather than covariantly. This means that a type system built upon the Checker Framework is sound: you get a compile-time guarantee without the need for any run-time checks. But it also means that the Checker Framework rejects code that is similar to what Java unsoundly accepts. The guarantee and the compile-time checks are about your extended type system. The Checker Framework does not reject the example code above, which contains no type annotations.

Java's covariant array typing is sound if the array is used in a read-only fashion: that is, if the array's elements are accessed but the array is not modified. However, fact about read-only usage is not built into any of the type-checkers except those that are specifically about immutability: IGJ (see Chapter 19, page 98) and Javari (see Chapter 20, page 102). Therefore, when using other type systems along with `-AinvariantArrays`, you will need to suppress any warnings that are false positives because the array is treated in a read-only way.

25.2 Context-sensitive type inference for array constructors

When you write an expression, the Checker Framework gives it the most precise possible type, depending on the particular expression or value. For example, when using the Regex Checker (Chapter 9, page 61), the string `"hello"` is

given type `@Regex String` because it is a legal regular expression (whether it is meant to be used as one or not) and the string `"(foo"` is given the type `@Unqualified String` because it is not a legal regular expression.

Array constructors work differently. When you create an array with the array constructor syntax, such as the right-hand side of this assignment:

```
String[] myStrings = {"hello"};
```

then the expression does not get the most precise possible type, because doing so could cause inconvenience. Rather, its type is determined by the context in which it is used: the left-hand side if it is in an assignment, the declared formal parameter type if it is in a method call, etc.

In particular, if the expression `{"hello"}` were given the type `@Regex String[]`, then the assignment would be illegal! But the Checker Framework gives the type `String[]` based on the assignment context, so the code type-checks.

If you prefer a specific type for a constructed array, you can indicate that either in the context (change the declaration of `myStrings`) or in a `new` construct (change the expression to `new @Regex String[] {"hello"}`).

25.3 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.

Most of this section is applicable only to source code that is being checked by a checker. When the compiler reads a `.class` file that was checked by a checker, the `.class` file contains the explicit or defaulted annotations from the source code and no defaulting is necessary. When the compiler reads a `.class` file that was not checked by a checker, the `.class` file contains only explicit annotations and defaulting might be necessary; see Section 25.3.5 for these rules.

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

1. If a type qualifier is present in the source code, that qualifier is used.
2. The type system adds implicit qualifiers. This happens whether or not the programmer has written an explicit type qualifier.
   Here are some examples of implicit qualifiers:
   - In the Nullness type system (see Chapter 3, page 24), `enum` values, string literals, and method receivers are always non-null.
   - In the Interning type system (see Chapter 5, page 47), string literals and `enum` values are always interned.

   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.
   Implicit qualifiers arise from two sources:

   built-in Implicit qualifiers can be built into a type system (Section 29.4), in which case the type system's documentation explains all of the type system's implicit qualifiers. Both of the above examples are built into the Nullness type system.

   programmer-declared A programmer may introduce an implicit annotation on each use of class *C* by writing a qualifier on the declaration of class *C*. If `MyClass` is declared as `class @MyAnno MyClass {...}`, then each occurrence of `MyClass` in the source code is treated as if it were `@MyAnno MyClass`.
3. If there is no explicit or implicit qualifier on a type, then a default qualifier is applied; see Section 25.3.1.

   At this point (after step 3), 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 25.4.

25.3.1 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 29.3.4); this is usually CLIMB-to-top (Section 25.3.2).
- 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 `"org.checkerframework.checker.nullness.qual.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 org.checkerframework.framework.qual.*;        // for DefaultQualifier[s]
import org.checkerframework.checker.nullness.qual.NonNull;

@DefaultQualifier(NonNull.class)
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.class)
  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.class),
  @DefaultQualifier(Mutable.class)
})
```

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 25.3) 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.)

25.3.2 Defaulting rules and CLIMB-to-top

Each type system defines a default qualifier. For example, the default qualifier for the Nullness Checker is `@NonNull`. That means that when a user writes a type such as `Date`, the Nullness Checker interprets it as `@NonNull Date`.

We recommend that the type system apply that default qualifier to most but not all types. In particular, we recommend the CLIMB-to-top rule. This rule states that the *top* qualifier in the hierarchy is applied to the CLIMB locations: **C**asts, **L**ocals, **I**nstanceof, and i**M**plicit **B**ounds. For example, when the user writes a type such as `Date` in such a location, the Nullness Checker interprets it as `@Nullable Date` (because `@Nullable` is the top qualifier in the hierarchy, see Figure 3.1).

The CLIMB-to-top rule is used only for unannotated source code that is being processed by a checker. For unannotated libraries (code read by the compiler in `.class` or `.jar` form), the checker uses conservative defaults (Section 25.3.5).

The rest of this section explains the rationale and implementation of CLIMB-to-top.

Here is the rationale for CLIMB-to-top:

- Casts and local variables (including resource variables in the try-with-resources construct, variables in for statements, exception parameters etc.) should be defaulted to top because they are the locations to which type refinement (Section 25.4) is applied. If they start as the top type, then the Checker Framework chooses the best (most general) possible type for them. As a result, a programmer rarely writes an explicit annotation on any of those locations.

  Catch arguments, known as exception parameters in the Java Language Specification, should be defaulted to top for most checkers; otherwise, an error will be issued. This is because exceptions of arbitrary qualified types can be thrown and the Checker Framework does not provide runtime checks.
- Instanceof types are defaulted to top for a similar reason: so that programmers do not need to write annotations on them. If the `instanceof`'s qualifier is top, then the Checker Framework will never issue an error that the `instanceof` qualifier is not compatible with the argument.
- Implicit upper bounds are defaulted to top to allow them to be instantiated in any way. If a user declared `class C<T> { ... }`, then we assume that the user intended to allow any instantiation of the class, and the declaration is interpreted as `class C<T extends @Nullable Object> { ... }` rather than as `class C<T extends @NonNull Object> { ... }`. The latter would forbid instantiations such as `C<@Nullable String>`, or would require rewriting of code. On the other hand, if a user writes an explicit bound such as `class C<T extends D> { ... }`, then the user intends some restriction on instantiation and can write a qualifier on the upper bound as desired.

  This rule means that the upper bound of `class C<T>` is defaulted differently than the upper bound of `class C<T extends Object>`. It would be more confusing for "`Object`" to be defaulted differently in `class C<T extends Object>` and in an instantiation `C<Object>`, and for the upper bounds to be defaulted differently in `class C<T extends Object>` and `class C<T extends Date>`.

- Implicit *lower* bounds are defaulted to the bottom type, again to allow maximal instantiation. Note that Java does not allow a programmer to express both the upper and lower bounds of a type, but the Checker Framework allows the programmer to specify either or both; see Section 24.1.2.

Here is how the CLIMB-to-top rule is expressed for the Nullness Checker:

```
@DefaultQualifierInHierarchy
public @interface NonNull { }

@DefaultFor({ DefaultLocation.LOCAL_VARIABLE, DefaultLocation.RESOURCE_VARIABLE,
    DefaultLocation.IMPLICIT_UPPER_BOUNDS })
public @interface Nullable { }
```

Note that `DefaultLocation.LOCAL_VARIABLE` includes casts and instanceof. As mentioned above, the exception parameters are always non-null, so `DefaultLocation.EXCEPTION_PARAMETER` is excluded from the above list.

A type system designer does not have to use the CLIMB-to-top rule. In addition, a user may choose a different rule for defaults using the `@DefaultQualifier` annotation; see Section 25.3.1.

25.3.3 Inherited defaults

In certain situations, it would be convenient for an annotation on a superclass member to be automatically inherited by subclasses that override it. This feature would reduce both annotation effort and program comprehensibility. In general, a program is read more often than it is edited/annotated, so the Checker Framework does not currently support this feature. Here are more detailed justifications:

- Currently, a user can determine the annotation on a parameter or return value by looking at a single file. If annotations could be inherited from supertypes, then a user would have to examine all supertypes to understand the meaning of an unannotated type in a given file.
- Different annotations might be inherited from a supertype and an interface, or from two interfaces. Presumably, the subtype's annotations would be stronger than either (the greatest lower bound in the type system), or an error would be thrown if no such annotations existed.

If these issues can be resolved, then the feature may be added in the future. Or, it may be added optionally, and each type-checker implementation can enable it if desired.

25.3.4 Inherited wildcard annotations

If a wildcard is unbounded and has no annotation (e.g. `List<?>`), the annotations on the wildcard's bounds are copied from the type parameter to which the wildcard is an argument. For example, the two wildcards in the declarations below are equivalent.

```
class MyList<@Nullable T extends @Nullable Object> {}

MyList<?> listOfNullables;
MyList<@Nullable ? extends @Nullable Object> listOfNullables;
```

We copy these annotations because wildcards must be within the bounds of their corresponding type parameter. Therefore, there would be many false positive `type.argument.type.incompatible` warnings if the bounds of a wildcard were defaulted differently from the bounds of its corresponding type parameter. Here is another example:

```
class MyList<@Regex(5) T extends @Regex(1) Object> {}

MyList<?> listOfRegexes;
MyList<@Regex(5) ? extends @Regex(1) Object> listOfRegexes;
```

Note, this applies only to unbounded wildcards. The two wildcards in the following example are equivalent.

```
class MyList<@Nullable T extends @Nullable Object> {}

List<? extends Object> listOfNonNulls;
List<@NonNull ? extends @NonNull Object> listOfNonNulls2;
```

Note, the upper bound of the wildcard `? extends Object` is defaulted to `@NonNull` using the CLIMB-to-top rule (see Section 25.3.2).

25.3.5 Default qualifiers for `.class` files (conservative library defaults)

Note: For release 1.9.3, the conservative library defaults presented in this section are off by default and can be turned on by supplying the `-AsafeDefaultsForUnannotatedBytecode` command-line option. Starting with release 1.9.4, they will be turned on by default and it will be possible to turn them off by supplying a `-AunsafeDefaultsForUnannotatedBytecode` command-line option. That option will replace the existing `-AsafeDefaultsForUnannotatedBytecode` command-line option.

The defaulting rules presented so far apply to source code that is read by the compiler. When the compiler reads a `.class` file, different defaulting rules apply.

If the checker was run during the compiler execution that created the `.class` file, then there is no need for defaults: the `.class` file has an explicit qualifier at each type use. (Furthermore, unless warnings were suppressed, those qualifiers are guaranteed to be correct.) When you are performing pluggable type-checking, it is best to ensure that the compiler only reads such `.class` files. Section 28.1 discusses how to create annotated libraries.

If the checker was not run during the compiler execution that created the `.class` file, then the `.class` file contains only the type qualifiers that the programmer wrote explicitly. (Furthermore, there is no guarantee that these qualifiers are correct, since they have not been checked.) In this case, each checker decides what qualifier to use for the locations where the programmer did not write an annotation. The typical choice is:

- For method parameters, use the bottom qualifier (see Section 29.3.5).
- For method return values, use the top qualifier (see Section 29.3.5).

For example, an unannotated method

```
String concatenate(String p1, String p2)
```

in a classfile would be interpreted as

```
@Top String concatenate(@Bottom String p1, @Bottom String p2)
```

There is no single possible default that is sound for fields. In the rare circumstance that there is a mutable public field in an unannotated library, the Checker Framework may fail to warn about code that can misbehave at run time. The Checker Framework developers are working to improve handling of mutable public fields in unannotated libraries.

These choices are conservative. They are likely to cause many false-positive type-checking errors, which will help you to know which library methods need annotations. You can then write those library annotations (see Chapter 28) or alternately suppress the warnings (see Section 26).

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

In order to reduce your 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 25.3.1) type. This functionality eliminates some false positive warnings, but it never introduces unsoundness nor causes an error to be missed.

As an example, suppose you write

```
@Nullable String myVar;
...
myVar = "hello";
myVar.hashCode();
```

The Nullness Checker issues a warning whenever a method such as `hashCode()` is called on a possibly-null value, which may result in a null pointer exception. The Nullness Checker need not issue a warning in this case. In particular, after the assignment, type-checker treats `myVar` as having type `@NonNull String`, which is a subtype of its declared type.

Here is another example:

```
@Nullable String myVar;
...
if (myVar != null) {
  myVar.hashCode();
}
```

Once again, the Nullness Checker need not issue a warning. Within the body of the `if` test, the type of `myVar` is `@NonNull String`, even though `myVar` is declared as `@Nullable String`.

Array element types and generic arguments are never changed by type refinement. Changing these components of a type never yields a subtype of the declared type. For example, `List<Number>` is *not* a subtype of `List<Object>`. Similarly, the Checker Framework does not treat `Number[]` as a subtype of `Object[]`; see Section 25.1 for why.

By default, all checkers, including new checkers that you write, automatically incorporate type refinement. Most of the time, users don't have to think about, and may not even notice, type refinement. The checkers simply do the right thing even when a programmer omits an annotation on a local variable, or when a programmer writes an unnecessarily general type in a declaration.

The functionality has a variety of names: automatic type refinement, flow-sensitive type qualifier inference, local type inference, and sometimes just "flow".

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 32.2) that includes a small piece of Java code that reproduces the problem.

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).

Type inference is never performed for method parameters of non-private methods, nor for non-private fields. More generally, the inferred information is never written to the `.class` file as user-written annotations are. If the checker did inference in externally-visible locations and wrote it to the `.class` file, then the resulting `.class` file would be different depending on whether an annotation processor had been run or not. It is a design goal that the same annotations appear in the `.class` file regardless of whether the class is compiled with or without the checker, and this requires that any public signature be fully annotated by the user rather than inferred.

The `@TerminatesExecution` annotation indicates that a given method never returns. This can enable the flow-sensitive type refinement to be more precise.

25.4.1 Run-time tests and type refinement

Some type systems support a run-time test that the Checker Framework can use to refine types within the scope of a conditional such as `if`, after an `assert` statement, etc.

Whether a type system supports such a run-time test depends on whether the type system is computing properties of data itself, or properties of provenance (the source of the data). An example of a property about data is whether a string is a regular expression. An example of a property about provenance is units of measure: there is no way to look at the representation of a number and determine whether it is intended to represent kilometers or miles.

Type systems that support a run-time test are:

- Nullness Checker for null pointer errors (see Chapter 3, page 24)
- Map Key Checker to track which values are keys in a map (see Chapter 4, page 44)
- Regex Checker to prevent use of syntactically invalid regular expressions (see Chapter 9, page 61)
- Format String Checker to ensure that format strings have the right number and type of `%` directives (see Chapter 10, page 65)
- Internationalization Format String Checker to ensure that i18n format strings have the right number and type of `{}` directives (see Chapter 11, page 71)

Type systems that do not currently support a run-time test, but could do so with some additional implementation work, are

- Interning Checker for errors in equality testing and interning (see Chapter 5, page 47)
- Lock Checker for concurrency and lock errors (see Chapter 6, page 50)
- Property File Checker to ensure that valid keys are used for property files and resource bundles (see Chapter 12, page 77)
- Internationalization Checker to ensure that code is properly internationalized (see Chapter 12.2, page 78)
- Signature String Checker to ensure that the string representation of a type is properly used, for example in `Class.forName` (see Chapter 13, page 80).
- Constant Value Checker to determine whether an expression's value can be known at compile time (see Chapter 16, page 89)

Type systems that cannot support a run-time test are:

- Initialization Checker to ensure all fields are set in the constructor (see Chapter 3.8, page 33)
- Fake Enum Checker to allow type-safe fake enum patterns (see Chapter 7, page 56)
- Tainting Checker for trust and security errors (see Chapter 8, page 59)
- GUI Effect Checker to ensure that non-GUI threads do not access the UI, which would crash the application (see Chapter 14, page 82)

- Units Checker to ensure operations are performed on correct units of measurement (see Chapter 15, page 86)
- Aliasing Checker to identify whether expressions have aliases (see Chapter 17, page 92)
- Linear Checker to control aliasing and prevent re-use (see Chapter 18, page 96)
- IGJ Checker for mutation errors (incorrect side effects), based on the IGJ type system (see Chapter 19, page 98)
- Javari Checker for mutation errors (incorrect side effects), based on the Javari type system (see Chapter 20, page 102)
- Subtyping Checker for customized checking without writing any code (see Chapter 22, page 108)

25.4.2 Fields and flow-sensitive 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 `updatedAt` is a nullable field:

  ```
  class DBObject {
    @Nullable Date updatedAt;

    void persistData() {
      ... // write to disk or other non-volatile memory
      updatedAt = null;
    }

    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 `SideEffectFree` or `Pure` (see Section 25.4.3). When a method is annotated as `SideEffectFree`, the flow analyzer carries the inferred types across the method invocation boundary.

25.4.3 Side effects, determinism, purity, and flow-sensitive analysis

As described above, a checker can use a refined type for an expression from the time when the checker infers that the value has that refined type, until the checker can no longer support that inference.

The refined type begins at a test (such as `if (myvar != null) ...`) or an assignment. If the assignment occurs within a method body, write a postcondition annotation such as `@EnsuresNonNull`.

The refined type ends at an assignment or possible assignment. Any method call has the potential to side-effect any field, so calling a method typically causes the checker to discard its knowledge of the refined type. This is undesirable if the method doesn't actually re-assign the field.

There are three annotations, collectively called purity annotations, that you can use to help express what effects a method call does not have. Usually, you only need to use `@SideEffectFree`.

`@SideEffectFree` indicates that the method has no (visible) side effects.

`@Deterministic` indicates that if the method is called multiple times with the same arguments, then it returns the same result.

`@Pure` indicates that the method is both @SideEffectFree and @Deterministic.

The Javadoc of the annotations describes their semantics and how they are checked. This manual section gives examples and supplementary information.

For example, consider the following declarations and uses:

```
@Nullable Object myField;

int computeValue() { ... }

...
if (myField != null) {
  int result = computeValue();
  myField.toString();
}
```

Ordinarily, the Nullness Checker would issue a warning regarding the `toString()` call, because the receiver `myField` might be `null`, according to the `@Nullable` annotation on the declaration of `myField`. Even though the code checked the value of `myField`, the call to `computeValue` might have re-set it to null. If you change the declaration of `computeValue` to

```
@SideEffectFree int computeValue() { ... }
```

then the Nullness Checker issues no warnings, because it can reason that the second occurrence of `myField` has the same (non-null) value as the one in the test.

As a more complex example, consider the following declaration and uses:

```
@Nullable Object getField(Object arg) { ... }

...
if (x.getField(y) != null) {
  x.getField(y).toString();
}
```

Ordinarily, the Nullness Checker would issue a warning regarding the `toString()` call, because the receiver `x.getField(y)` might be `null`, according to the `@Nullable` annotation in the declaration of `getField`. If you change the declaration of `getField` to

```
@Pure @Nullable Object getField(Object arg) { ... }
```

then the Nullness Checker issues no warnings, because it can reason that the two invocations `x.getField(y)` have the same value, and therefore that `x.getField(y)` is non-null within the then branch of the if statement.

If a method is side-effect-free or pure, then it would be legal to annotate its receiver and every parameter as `@ReadOnly`, in the IGJ (Chapter 19) or Javari (Chapter 20) type systems. The reverse is not true, because the method might side-effect a global variable. (Also, for the case of `@Pure`, the method might not be deterministic.)

If you supply the command-line option `-AsuggestPureMethods`, then the Checker Framework will suggest methods that can be marked as `@SideEffectFree`, `@Deterministic`, or `@Pure`.

Currently, purity annotations are trusted. Purity annotations on called methods affect type-checking of client code. However, you can make a mistake by writing `@SideEffectFree` on the declaration of a method that is not actually side-effect-free or by writing `@Deterministic` on the declaration of a method that is not actually deterministic. To enable checking of the annotations, supply the command-line option `-AcheckPurityAnnotations`. It is not enabled by default because of a high false positive rate. In the future, after a new purity-checking analysis is implemented, the Checker Framework will default to checking purity annotations.

It can be tedious to annotate library methods with purity annotations such as `@SideEffectFree`. If you supply the command-line option `-AassumeSideEffectFree`, then the Checker Framework will unsoundly assume that every called method is side-effect-free. This can make flow-sensitive type refinement much more effective, since method calls will not cause the analysis to discard information that it has learned. However, this option can mask real errors. It is most appropriate when you are starting out annotating a project, or if you are using the Checker Framework to find some bugs but not to give a guarantee that no more errors exist of the given type.

A common error is:

```
MyClass.java:1465: error: int hashCode() in MyClass cannot override int hashCode(Object this) in java.lang
    public int hashCode() {
               ^
  found   : []
  required: [SIDE_EFFECT_FREE, DETERMINISTIC]
```

The reason for the error is that the `Object` class is annotated as:

```
class Object {
  ...
  @Pure int hashCode() { ... }
}
```

(where `@Pure` means both `@SideEffectFree` and `@Deterministic`). Every overriding definition, including those in your program, must use be at least as strong a specification; in particular, every overriding definition must be annotated as `@Pure`.

You can fix the definition by adding `@Pure` to your method definition. Alternately, you can suppress the warning. You can suppress each such warning individually using `@SuppressWarnings("purity.invalid.overriding")`, or you can use the `-AsuppressWarnings=purity.invalid.overriding` command-line argument to suppress all such warnings. In the future, the Checker Framework will support inheriting annotations from superclass definitions.

25.4.4 Assertions

If your code contains an `assert` statement, then your code could behave in two different ways at run time, depending on whether assertions are enabled or disabled via the `-ea` or `-da` command-line options to java.

By default, the Checker Framework outputs warnings about any error that could happen at run time, whether assertions are enabled or disabled.

If you supply the `-AassumeAssertionsAreEnabled` command-line option, then the Checker Framework assumes assertions are enabled. If you supply the `-AassumeAssertionsAreDisabled` command-line option, then the Checker Framework assumes assertions are disabled. You may not supply both command-line options. It is uncommon to supply either one.

These command-line arguments have no effect on processing of `assert` statements whose message contains the text `@AssumeAssertion`; see Section 26.2.

25.5 Writing Java expressions as annotation arguments

Sometimes, it is necessary to write a Java expression as the argument to an annotation. The annotations that take a Java expression as an argument include:

- `@RequiresQualifier`
- `@EnsuresQualifier`
- `@EnsuresQualifierIf`
- `@RequiresNonNull`
- `@EnsuresNonNull`
- `@EnsuresNonNullIf`
- `@KeyFor`
- `@I18nFormatFor`

The expression is a subset of legal Java expressions:

- the receiver object, `this`.
- the receiver object as seen from the superclass, `super`. This can be used to refer to fields shadowed in the subclass (although shadowing fields is discouraged in Java).
- a formal parameter. Write `#` followed by the **one-based** parameter index. For example: `#1`, `#3`. It is not permitted to write `#0` to refer to the receiver object; use `this` instead.
- a static variable. Write the class name and the variable, as in `System.out`.
- a field of any expression. For example: `next`, `this.next`, `#1.next`.
- an array access. For example: `this.myArray[i]`, `vals[#1]`.
- literals: string, integer, long, null.
- a method invocation on any expression. This even works for overloaded methods and methods with type parameters. For example: `m1(x, y.z, #2)`, `a.m2("hello")`.

You may optionally omit a leading "`this.`", just as in Java. Thus, `this.next` and `next` are equivalent.

One unusual feature is that the method call is allowed to have side effects. If a specification is going to be checked at run time via assertions, then the specification must not use methods with side effects. But, the Checker Framework works at compile time, so it allows side effects. The current implementation will never able to prove such a contract, but it is able to use the information (when checking the method body with preconditions, or when checking the callers code with postconditions). This can be useful to annotate trusted methods precisely (e.g., `java.io.BufferedReader.ready()`).

(A side note: The formal parameter syntax `#1` is less natural in source code than writing the formal parameter name. This syntax is necessary for separate compilation, when an annotated method has already been compiled into a `.class` file and a client of that method is later compiled. In the `.class` file, no formal parameter name information is available, so it is necessary to use a number to indicate a formal parameter.)

Limitations: The following Java expressions may not currently be written:

- Some literals: floats, doubles, chars, and class literals.
- String concatenation expressions.
- Mathematical operators (plus, minus, division, ...).
- Comparisons (equality, less than, etc.).
- Quantification over all array components (e.g. to express that all array elements are non-null).

25.6 Unused fields

In an inheritance hierarchy, subclasses often introduce new methods and fields. For example, a `Marsupial` (and its subclasses such as `Kangaroo`) might have a variable `pouchSize` indicating the size of the animal's pouch. The field does not exist in superclasses such as `Mammal` and `Animal`, so Java issues a compile-time error if a program tries to access `myMammal.pouchSize`.

If you cannot use subtypes in your program, you can enforce similar requirements using type qualifiers. For fields, use the `@Unused` annotation (Section 25.6.1), which enforces that a field or method may only be accessed from a receiver expression with a given annotation (or one of its subtypes). For methods, annotate the receiver parameter `this`; then a method call type-checks only if the actual receiver is of the specified type.

Also see the discussion of typestate checkers, in Chapter 23.1.

25.6.1 `@Unused` annotation

A Java subtype can have more fields than its supertype. For example:

```
class Animal { }
class Mammal extends Animal { ... }
class Marsupial extends Mammal {
  int pouchSize;  // pouch capacity, in cubic centimeters
  ...
}
```

You can simulate the same effect for type qualifiers: the `@Unused` annotation on a field declares that the field may *not* be accessed via a receiver of the given qualified type (or any *super*type). For example:

```
class Animal {
  @Unused(when=Mammal.class)
  int pouchSize;  // pouch capacity, in cubic centimeters
  ...
}
@interface Mammal { }
@interface Marsupial { }

@Marsupial Animal joey = ...;
... joey.pouchSize ...    // OK
@Mammal Animal mae = ...;
... mae.pouchSize ...    // compile-time error
```

The above class declaration is like writing

```
class @Mammal-Animal { ... }
class @Marsupial-Animal {
  int pouchSize;  // pouch capacity, in cubic centimeters
  ...
}
```

Chapter 26

Suppressing warnings

When the Checker Framework reports a warning, it's best to change the code or its annotations, to eliminate the warning. Alternately, you can suppress the warning, which does not change the code but prevents the Checker Framework from reporting this particular warning to you.

You may wish to suppress checker warnings because of unannotated libraries or un-annotated portions of your own code, because of application invariants that are beyond the capabilities of the type system, because of checker limitations, because you are interested in only some of the guarantees provided by a checker, or for other reasons. Suppressing a warning is similar to writing a cast in a Java program: the programmer knows more about the type than the type system does and uses the warning suppression or cast to convey that information to the type system.

You can suppress a single warning message (or those in a single method or class) by using the following mechanisms:

- the `@SuppressWarnings` annotation (Section 26.1), or
- the `@AssumeAssertion` string in an `assert` message (Section 26.2).

You can suppress warnings throughout the codebase by using the following mechanisms:

- the `-AsuppressWarnings` command-line option (Section 26.3),
- the `-AskipUses` and `-AonlyUses` command-line options (Section 26.4),
- the `-AskipDefs` and `-AonlyDefs` command-line options (Section 26.5),
- the `-AuseSafeDefaultsForUnannotatedSourceCode` command-line option (Section 28.1),
- the `-Alint` command-line option (Section 26.6), or
- not using the `-processor` command-line option (Section 26.7).

Some type checkers can suppress warnings via

- checker-specific mechanisms (Section 26.8).

We now explain these mechanisms in turn.

26.1 `@SuppressWarnings` annotation

You can suppress specific errors and warnings by use of the `@SuppressWarnings` annotation, for example `@SuppressWarnings("interni` or `@SuppressWarnings("nullness")`. Section 26.1.1 explains the syntax of the argument string.

A `@SuppressWarnings` annotation may be placed on program declarations such as a local variable declaration, a method, or a class. It suppresses all warnings related to the given checker, for that program element. Section 26.1.2 discusses where the annotation may be written in source code.

Section 26.1.3 gives best practices for writing `@SuppressWarnings` annotations.

26.1.1 `@SuppressWarnings` syntax

The `@SuppressWarnings` annotation takes a string argument.

The most common usage is `@SuppressWarnings("`*`checkername`*`")`, as in `@SuppressWarnings("interning")` or `@SuppressWarnings("nullness")`. The argument *checkername* is in lower case and is derived from the way you invoke the checker. For example, if you invoke a checker as `javac -processor MyNiftyChecker ...`, then you would suppress its error messages with `@SuppressWarnings("mynifty")`. (An exception is the Subtyping Checker, for which you use the annotation name; see Section 22.1). While not recommended, using `@SuppressWarnings("all")` will suppress all warnings for all checkers.

The `@SuppressWarnings` argument string can also be of the form *checkername:messagekey*, in which case only errors/warnings relating to the given message key are suppressed. For example, `cast.unsafe` is the messagekey for warnings about an unsafe cast, and `cast.redundant` is the messagekey for warnings about a redundant cast.

Each warning from the compiler gives the most specific suppression key that can be used to suppress that warning. An example is `dereference.of.nullable` in

```
MyFile.java:107: error: [dereference.of.nullable] dereference of possibly-null reference myList
          myList.add(elt);
          ^
```

With the `-AshowSuppressWarningKeys` command-line option, the compiler lists every key that would suppress the warning, not just the most specific one.

26.1.2 Where `@SuppressWarnings` can be written

`@SuppressWarnings` is a declaration annotation, so it may be placed on program declarations such as a local variable declaration, a method, or a class. It cannot be used on statements, expressions, or types. To reduce the scope of a `@SuppressWarnings` annotation, it is sometimes desirable to extract part of an expression into a local variable, so that warnings can be suppressed just for that local variable's initializer expression.

As an example, consider suppressing a warnings at a cast that you know is safe. Here is an example that uses the Tainting Checker (Section 8); assume that `expr` has compile-time (declared) type `@Tainted String`, but you know that the run-time value of `expr` is untainted.

```
@SuppressWarnings("tainting:cast.unsafe") // expr is untainted because ... [explanation goes here]
@Untainted String myvar = expr;
```

It would have been *illegal* to write

```
@Untainted String myvar;
...
@SuppressWarnings("tainting:cast.unsafe") // expr is untainted because ...
myvar = expr;
```

This does not work because Java does not permit annotations (such as `@SuppressWarnings`) on assignments or other statements or expressions.

26.1.3 Good practices when suppressing warnings

Suppress warnings in the smallest possible scope

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, rather than suppressing warnings for a larger expression or an entire method body. See Section 26.1.2.

Use a specific argument to `@SuppressWarnings`

It is best to use the most specific possible message key to suppress just a specific error that you know to be a false positive. The checker outputs this message key when it issues an error. If you use a broader `@SuppressWarnings` annotation, then it may mask other errors that you needed to know about.

The example of Section 26.1.2 could have been written as any one of the following, with the last one being the best style:

```
@SuppressWarnings("tainting")              // suppresses all tainting-related warnings
@SuppressWarnings("tainting:cast")         // suppresses tainting warnings about casts
@SuppressWarnings("tainting:cast.unsafe")  // suppresses tainting warnings about unsafe casts
```

Justify why the warning is a false positive

A `@SuppressWarnings` annotation asserts that the code is actually correct or safe (that is, no undesired behavior will occur), even though the type system is unable to prove that the code is correct or safe.

Whenever you write a `@SuppressWarnings` annotation, you should also write, typically on the same line, a code comment explaining why the code is actually correct. In some cases you might also justify why the code cannot be rewritten in a simpler way that would be amenable to type-checking.

This documentation will help you and others to understand the reason for the `@SuppressWarnings` annotation. It will also help if you decide to audit your code to verify all the warning suppressions.

26.2 `@AssumeAssertion` string in an `assert` message

You can suppress a warning by `assert`ing that some property is true, and placing the string `@AssumeAssertion(`*`warningkey`*`)` in the assertion message.

For example, in this code:

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

the Nullness Checker assumes that `x` is non-null from the `assert` statement forward, and so the expression `x.f` cannot throw a null pointer exception.

The `assert` expression must be an expression that would affect flow-sensitive type qualifier refinement (Section 25.4), if the expression appeared in a conditional test. Each type system has its own rules about what type refinement it performs.

The warning key is exactly as in the `@SuppressWarnings` annotation (Section 26.1). The same good practices apply as for `@SuppressWarnings` annotations, such as writing a comment justifying why the assumption is safe (Section 26.1.3).

The `-AassumeAssertionsAreEnabled` and `-AassumeAssertionsAreDisabled` command-line options (Section 25.4.4) do not affect processing of `assert` statements that have `@AssumeAssertion` in their message. Writing `@AssumeAssertion` means that the assertion would succeed if it were executed, and the Checker Framework makes use of that information regardless of the `-AassumeAssertionsAreEnabled` and `-AassumeAssertionsAreDisabled` command-line options.

26.2.1 Suppressing warnings and defensive programming

This section explains the distinction between two different uses for assertions (and for related methods like JUnit's `Assert.assertNotNull`).

Assertions are commonly used for two distinct purposes: documenting how the program works and debugging the program when it does not work correctly. By default, the Checker Framework assumes that each assertion is used for debugging: the assertion might fail at run time, and the programmer wishes to be informed at compile time about such run-time errors. On the other hand, if you write the `@AssumeAssertion` string in the `assert` message, then the Checker Framework assumes that you have used some other technique to verify that the assertion can never fail at run time, so the checker assumes the assertion passes and does not issue a warning.

Distinguishing the purpose of each assertion is important for precise type-checking. Suppose that a programmer encounters a failing test, adds an assertion to aid debugging, and fixes the test. The programmer leaves the assertion

in the program if the programmer is worried that the program might fail in a similar way in the future. The Checker Framework should not assume that the assertion succeeds — doing so would defeat the very purpose of the Checker Framework, which is to detect errors at compile time and prevent them from occurring at run time.

On the other hand, assertions sometimes document facts that a programmer has independently verified to be true, and the Checker Framework can leverage these assertions in order to avoid issuing false positive warnings. The programmer marks such assertions with the `@AssumeAssertion` string in the `assert` message. Only do so if you are sure that the assertion always succeeds at run time.

Sometimes methods such as `NullnessUtils.castNonNull` are used instead of assertions. Just as for assertions, you can treat them as debugging aids or as documentation. 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 suppress warnings related to it. 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.

26.3 `-AsuppressWarnings` command-line option

Supplying the `-AsuppressWarnings` command-line option is equivalent to writing a `@SuppressWarnings` annotation on every class that the compiler type-checks. The argument to `-AsuppressWarnings` is a comma-separated list of warning suppression keys, as in `-AsuppressWarnings=purity,uninitialized`.

When possible, it is better to write a `@SuppressWarnings` annotation with a smaller scope, rather than using the `-AsuppressWarnings` command-line option.

26.4 `-AskipUses` and `-AonlyUses` command-line options

You can suppress all errors and warnings at all *uses* of a given class, or suppress all errors and warnings except those at uses of a given class. (The class itself is still type-checked, unless you also use the `-AskipDefs` or `-AonlyDefs` command-line option, see 26.5).

Set the `-AskipUses` command-line option to a regular expression that matches class names (not file names) for which warnings and errors should be suppressed. Or, set the `-AonlyUses` command-line option to a regular expression that matches class names (not file names) for which warnings and errors should be emitted; warnings about uses of all other classes will be suppressed.

For example, suppose that you use "`-AskipUses=^java\.`" on the command line (with appropriate quoting) when invoking `javac`. Then the checkers will suppress all warnings related to classes whose fully-qualified name starts with `java.`, such as 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 "`-AskipUses="java\.lang\.|java\.util\."`" to suppress all warnings related to uses of classes belong to the `java.lang` or `java.util` packages.

You can supply both `-AskipUses` and `-AonlyUses`, in which case the `-AskipUses` argument takes precedence, and `-AonlyUses` does further filtering but does not add anything that `-AskipUses` removed.

Warning: Use the `-AonlyUses` command-line option with care, because it can have unexpected results. For example, if the given regular expression does not match classes in the JDK, then the Checker Framework will suppress every

warning that involves a JDK class such as `Object` or `String`. The meaning of `-AonlyUses` may be refined in the future. Oftentimes `-AskipUses` is more useful.

26.5 `-AskipDefs` and `-AonlyDefs` command-line options

You can suppress all errors and warnings in the *definition* of a given class, or suppress all errors and warnings except those in the definition of a given class. (Uses of the class are still type-checked, unless you also use the `-AskipUses` or `-AonlyUses` command-line option, see 26.4).

Set the `-AskipDefs` command-line option to a regular expression that matches class names (not file names) in whose definition warnings and errors should be suppressed. Or, set the `-AonlyDefs` command-line option to a regular expression that matches class names (not file names) whose definitions should be type-checked.

For example, if you use "`-AskipDefs=^mypackage\.`" on the command line (with appropriate quoting) when invoking `javac`, then the definitions of classes whose fully-qualified name starts with `mypackage.` will not be checked.

If you supply both `-AskipDefs` and `-AonlyDefs`, then `-AskipDefs` takes precedence.

Another way not to type-check a file is not to pass it on the compiler command-line: the Checker Framework type-checks only files that are passed to the compiler on the command line, and does not type-check any file that is not passed to the compiler. The `-AskipDefs` and `-AonlyDefs` command-line options are intended for situations in which the build system is hard to understand or change. In such a situation, a programmer may find it easier to supply an extra command-line argument, than to change the set of files that is compiled.

A common scenario for using the arguments is when you are starting out by type-checking only part of a legacy codebase. After you have verified the most important parts, you can incrementally check more classes until you are type-checking the whole thing.

26.6 `-Alint` command-line option

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 `redundantNullComparison` for the Nullness Checker (see Section 3.1) and `dotequals` for the Interning Checker (see Section 5.3). 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.

Only the last `-Alint` option is used; all previous `-Alint` options are silently ignored. In particular, this means that `-Alint=all -Alint=cast:redundant` is *not* equivalent to `-Alint=-all,cast:redundant`.

26.7 No `-processor` command-line option

You can also compile parts of your code without use of the `-processor` switch to `javac`. No checking is done during such compilations, so no warnings are issued related to pluggable type-checking.

26.8 Checker-specific mechanisms

Finally, some checkers have special rules. For example, the Nullness checker (Chapter 3) uses the special `castNonNull` method to suppress warnings (Section 3.4.1). This manual also explains special mechanisms for suppressing warnings issued by the Fenum Checker (Section 7.4) and the Units Checker (Section 15.5).

Chapter 27

Handling 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.

Also recall that you can convert checker errors into warnings via the `-Awarns` command-line option; see Section 2.2.2.

27.1 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 28 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 26).
3. Suppress all warnings related to uses of `unannotated_method` via the `skipUses` processor option (see Section 26.4). Since this can suppress more warnings than you may expect, it is usually better to annotate at least the method's signature. If you choose the boundary between the annotated and unannotated code wisely, then you only have to annotate the signatures of a limited number of classes/methods (e.g., the public interface to a library or package).

Chapter 28 discusses adding annotations to signatures when you do not have source code available. Section 26 discusses suppressing warnings.

27.2 Backward compatibility with earlier versions of Java

Sometimes, your code needs to be *compiled* by people who are using a Java 5/6/7 compiler, which does not support type annotations. You can handle this situation by writing annotations in comments (Sections 27.2.1–27.2.3).

If your code just needs to be *run* by people who are not using a Java 8 JVM, supply an appropriate `-target` command-line option to javac. As discussed in Section 27.2.4, the disadvantage is that this makes it more difficult for clients of your library to use pluggable type-checking to verify their own code against the `.class` or `.jar` files that you supply; Section 27.2.5 gives a partial solution.

27.2.1 Annotations in comments

A Java 4 compiler does not permit use of annotations. A Java 5/6/7 compiler only permits annotations on declarations — it does not permit annotations on generic arguments, casts, `extends` clauses, method receivers, etc.

So that your code can be compiled by any Java compiler (for any version of the Java language), you may write any single annotation inside a `/*...*/` Java comment, as in `List</*@NonNull*/ String>`. The Checker Framework compiler treats the code exactly as if you had not written the `/*` and `*/`. In other words, the Checker Framework compiler will recognize the annotation (when it is targeting a Java 8 or later JVM), but your code will still compile with any Java compiler.

By default, the Checker Framework 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*/` and `/*@ NonNull*/` and `/*@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 it's even better to disable inserting spaces: in the Formatter preferences, in the Comments tab, unselect the "enable block comment formatting" checkbox.

Compiler flag `-XDTA:noannotationsincomments` causes the compiler to ignore annotation comments. With this compiler flag, the Checker Framework compiler behaves like a standard Java 8 compiler that does not support annotations in comments. If your code already contains comments of the form `/*@...*/` that look like type annotations, and you want the Checker Framework compiler not to try to interpret them, then you can either selectively add spaces to the comments or use `-XDTA:noannotationsincomments` to turn off all annotation comments.

Note: Annotations in comments is a feature of the javac compiler that is distributed along with the Checker Framework. It is *not* supported by the mainline OpenJDK javac. This is the key difference between the Checker Framework compiler and the OpenJDK compiler.

Annotations in comments do not appear in Java 5/6/7 `.class` files

The Checker Framework compiler ignores annotations in comments when targeting a Java 5/6/7 JVM, for example when the `-target 7` command-line option is supplied.

It would be possible for the Checker Framework compiler to read the annotations in comments and place them in the Java 5/6/7 `.class` file so that they are available when type-checking client code. However, this would have two problems. First, it would only be use useful to the Checker Framework compiler, because a standard Java 8 compiler will not look for type annotations in Java 5/6/7 bytecode. Second, the type annotations make reference to parts of the Java 8 JDK, such as `ElementType.TYPE_USE`. Therefore, trying to run the `.class` file on a Java 5/6/7 JVM would cause warnings or crashes.

27.2.2 Import statements and receiver parameters in comments

There is a more powerful mechanism that permits arbitrary code to be written in a comment. Format the comment as "`/*»>...*/`", with the first three characters of the comment being greater-than signs. As with annotations in comments, the commented code is ignored by ordinary compilers but is treated like code by the Checker Framework compiler.

This mechanism is intended for two purposes. First, it supports the receiver (`this` parameter) syntax. For example, to specify a method that does not modify its receiver:

```
public boolean method1(/*>>> @ReadOnly MyClass this*/) { ... }
public boolean method2(/*>>> @ReadOnly MyClass this, */ String argument) { ... }
```

Second, it can be used for import statements:

```
/*>>>
import org.checkerframework.checker.nullness.qual.*;
import org.checkerframework.checker.regex.qual.*;
*/
```

If the import statements are *not* commented out, then every time you compile the code (even when not doing pluggable type-checking), the annotation definitions (e.g., the `checker.jar` or `checker-qual.jar` file) must be on the classpath. (This is done automatically if you use the Checker Framework comiler.) Commenting out the import statements also eliminates Eclipse warnings about unused import statements, if all uses of the imported qualifier are themselves in comments and thus invisible to Eclipse.

A third use is for writing multiple annotations inside one comment, as in `/*»> @NonNull @Interned */ String s;`. However, it is better style to write multiple annotations each inside its own comment, as in `/*@NonNull*/ /*@Interned*/ String s;`.

It would be possible to abuse the `/*»>...*/` mechanism to inject code only when using the Checker Framework compiler. Doing so is not a sanctioned use of the mechanism.

27.2.3 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 compilers that support type annotations and your code will only run on Java 8 or later JVMs. 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*\*/' \
  | xargs perl -pi.bak -e 's|/\*\s*@([^ */]+)\s*\*/|@\1|g'
```

You can customize this command:

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

The command does not handle the »> comments; you will need to adapt the above command to do so, or remove them in another way.

27.2.4 No modular type-checking when targeting Java 5/6/7

The Checker Framework's type annotations utilize a Java 8 feature that allows them to be placed on any type use, including generic type parameters as in `List<@NonNull String>`. A downside is that use of these type annotations creates a dependency on Java 8, which means that the compiled program requires a Java 8 or later JDK at run time.

To ensure that your program can run on a Java 5/6/7 JVM, use a command-line option such as `-target 7` when doing normal compilation to produce classfiles. Before doing so, you will do pluggable typechecking, using the `-target 8` command-line option (or no `-target` command-line option) to javac; you may wish to supply the `-proc:only` command-line argument so that the type-checking step does not overwrite existing classfiles.

Here are the disadvantages of this approach:

- It produces classfiles that contain no trace of your type annotations. This means that modular typechecking (also known as separate compilation) is not possible.
  You need to compile your entire application every time you do pluggable type-checking, rather than just compiling a subset of the files. Furthermore, clients of your code cannot do pluggable type-checking to verify that they are using your code correctly, unless they re-compile your code (or at least all the interfaces that they use) every time that they compile their own.
- It makes pluggable type-checking a different step than "real" compilation, rather than both happening at the same time. You will do pluggable type-checking first, and when it works or when you want to create a binary to distribute to others, you will compile with an ordinary Java compiler.

One way to enable clients to do pluggable type-checking is to provide a version of your library compiled for Java 8 or later, with the type annotations. Clients will do type-checking against this version of the library, but will do normal compilation and execution using the Java 5, 6, or 7 version of your library.

Section 27.2.5 gives an alternative approach with its own advantages and disadvantages.

27.2.5 Distributing declaration annotations instead of type annotations

If it is important to you to distribute Java 5/6/7 classfiles against which clients can do some type-checking, this section gives a way to do so.

The idea is to use annotations that are Java 5/6/7 declaration annotations. This approach requires you to use annotations that are declared in different packages than usual and that have slightly different names.

- At code locations that are legal for both declaration and type annotations (such as for fields, method returns, and method parameters), write annotations normally (not in comments).
- At locations where a declaration annotation is not permitted (such as generic type parameters and `extends` clauses), write annotations in comments.

Here are some disadvantages of this approach:

- You need to use nonstandard names for some annotations, and to remember which annotations to write in comments and which to write normally.
- It produces classfiles that contain only some of your type annotations — the ones that were not written in comments. If your code uses type annotations at locations such as generic type parameters and `extends` clauses, then modular type-checking will not observe them; the implications of that were described above.

Here are more details about the approach. Suppose you wish to run the Nullness Checker using Java 6 or 7 declaration annotations rather than type annotations. You have two options.

1. At locations where declaration annotations are possible, use aliased annotations from other projects. For example, the aliased annotations for the Nullness Checker are listed in Section 3.7.
   At locations where only type annotations are possible, use the "`*Type`" compatibility annotations from package `org.checkerframework.checker.nullness.compatqual` in comments. For example, the Nullness Checker declares these declaration annotations: `@NullableType`, `@NonNullType`, `@PolyNullType`, `@MonotonicNonNullType`, and `@KeyForType`.
2. At locations where declaration annotations are possible, use "`*Decl`" compatibility annotations from package `org.checkerframework.checker.nullness.compatqual`. For example, the Nullness Checker declares these declaration annotations: `@NullableDecl`, `@NonNullDecl`, `@PolyNullDecl`, `@MonotonicNonNullDecl`, and `@KeyForDecl`.
   At locations where only type annotations are possible, use the regular Checker Framework type annotations in comments.

Notice that in each case, the declaration annotations and type annotations have distinct names. This enables a programmer to import both sets of annotations without a name conflict. But, you must remember to use the correct name, depending on where the annotations are written.

Eventually, when backward compatibility with Java 7 and earlier is not important, you should refactor your codebase to use only the regular Checker Framework annotations, and not to write them in comments.

Chapter 28

Annotating libraries

If your code uses a library that does *not* contain type annotations, then the type-checker has no way to know the library's behavior. The type-checker makes conservative assumptions about unannotated bytecode: it assumes that every method parameter has the bottom type annotation and that every method return type has the top type annotation (see Section 25.3.5 for details and an example). These conservative library annotations invariably lead to checker warnings. This chapter describes how to eliminate the warnings by adding annotations to the library. (Alternately, you can instead suppress all warnings related to an unannotated library by use of the `-AskipUses` or `-AonlyUses` command-line option; see Section 26.4.)

(Note: This chapter uses "library" to refer to code that is provided in `.class` or `.jar` form. You should use this approach for parts of your own codebase if you typically compile different parts separately. If your codebase is typically compiled together and you are type-checking only part of it, you can use the approach described in this chapter, or you can use command-line arguments such as `-AskipUses` and `-AskipDefs` (see Sections 26.4–26.5). Also, recall that the Checker Framework analyzes all, and only, the source code that is passed to it. The Checker Framework is a plug-in to the javac compiler, and it never analyzes code that is not being compiled, though it does look up annotations in the class files for code that was previously compiled.)

You make the library's annotations known to the checkers by writing annotations in a copy of the library's source code (or in a "stub file" if you do not have access to the source code). Given the library annotations, you have two options:

1. You can compile the library to create `.class` and `.jar` files that contain the annotations. Then, when doing pluggable type-checking, you would put those files on the classpath. When running your code, you can use either version of the library: the one you created or the original distributed version.
   With this compilation approach, the syntax of the library annotations is validated ahead of time. Thus, this compilation approach is less error-prone, and the type-checker runs faster. You get correctness guarantees about the library in addition to your code. Section 28.1 describes how to compile a library.
2. You can supply the annotated library source code, or a very concise variant called a "stub file", textually to the Checker Framework.
   The stub file approach does not require you to compile the library source code. A stub file is applicable to multiple versions of a library, so the stub file does not need to to be updated when a new version of the library is released. When provided by the author of the checker, a stub file is used automatically, with no need for the user to supply a command-line option. The stub file reader approach has some limitations, notably using non-standard syntax in some locations (Section 28.2.5). Section 28.2 describes how to create and use stub files.

If you write any library annotations, please share them so that they can be distributed with the Checker Framework. Sharing your annotations is useful even if the library is only partially annotated.

28.1 Compiling partially-annotated libraries

If you completely annotate a library, then you can compile it using a pluggable type-checker, and include the resulting `.jar` file on your classpath. You get a guarantee that the library contains no errors.

The rest of this section tells you how to compile a library if you *partially* annotate it: that is, you write annotations for some of its classes but not others. (There is another type of partial annotation, which is when you annotate method signatures but do not type-check the bodies. To do that variety of partial annotation, simply suppress warnings; see Chapter 26. You can combine the two types of partial annotation.)

When compiling a partially-annotated library, the checker needs to use normal defaulting rules (Section 25.3.2) for code you have annotated and conservative defaulting rules (Section 25.3.5) for code you have not yet annotated. You use `@AnnotatedFor` to indicate which classes you have annotated.

28.1.1 The `-AuseSafeDefaultsForUnannotatedSourceCode` command-line argument

When compiling a library that is not fully annotated, use command-line argument `-AuseSafeDefaultsForUnannotatedSourceCode`. This causes the checker to behave normally for classes with a relevant `@AnnotatedFor` annotation. For all other classes, the checker uses conservative defaults (see Section 25.3.5) for any type use with no explicit user-written annotation, and the checker issues no warnings.

The `@AnnotatedFor` annotation, written on a class, indicates that the class has been annotated for certain type systems. For example, `@AnnotatedFor({"nullness", "regex"})` means that the programmer has written annotations for the Nullness and Regular Expression type systems. If one of those two type-checkers is run, the `-AuseSafeDefaultsForUnannotatedSourceCode` command-line argument has no effect and this class is treated normally: unannotated types are defaulted using normal source-code defaults and type-checking warnings are issued. `@AnnotatedFor`'s arguments are any string that may be passed to the `-processor` command-line argument: the fully-qualified class name for the checker, or a shorthand for built-in checkers (see Section 2.2.4).

Whenever you compile a class using the Checker Framework, including when using the `-AuseSafeDefaultsForUnannotatedSource` command-line argument, the resulting `.class` files are fully-annotated; each type use in the `.class` file has an explicit type qualifier for any checker that is run.

28.1.2 Workflow for creating or augmenting a partially-annotated library

This section describes the typical workflow for creating a partially-annotated library.

1. Read file `checker-framework/checker/lib/README` to find out whether an annotated version of the library already exists.
   If it does not already exist, fork the project (if its license permits forking). Add a note, perhaps in a README, indicating how to obtain the corresponding upstream version; that will enable others to see exactly what edits you have made.
   Adjust the library's build process, such as a Maven or Ant buildfile.
   (a) Every time the build system runs the compiler, it should:
   - passes the `-AuseSafeDefaultsForUnannotatedSourceCode` command-line option and
   - runs every pluggable type-checker for which any annotations exist, using `-processor` *`TypeSystem1`*`,`*`TypeSystem2`*
   (b) When the build system creates a `.jar` file, the resulting `.jar` file includes the contents of `checker-framework/checker/dist`

   You are not adding new build targets, but modifying existing targets. The reason to run every type-checker is to verify the annotations you wrote, and to use appropriate defaults for all unnanotated type uses. The reason to include the contents of `checker-qual.jar` is so that the resulting `.jar` file can be used whether or not the Checker Framework is being run.
2. Annotate some files.
   When you annotate a file, annotate the whole thing, not just a few of its methods. Once the file is fully annotated, add an `@AnnotatedFor({"`*`checkername`*`"})` annotation to its class(es), or augment an existing `@AnnotatedFor` annotation.

3. Build the library.
   Because of the changes that you made in step 1, this will run pluggable type-checkers. If there are any compiler warnings, fix them and re-compile.
   Now you have a `.jar` file that you can use while type-checking and at run time.
4. Tell other people about your work so that they can benefit from it.
   - Please inform the Checker Framework developers about your new annotated library by opening an issue. This will let us include your annotated `.jar` file in directory `checker-framework/checker/lib/` of the Checker Framework release.
   - Encourage the library's maintainers to accept your annotations into its main version control repository. This will make the annotations easier to maintain, the library will obtain the correctness guarantees of pluggable type-checking, and there will be no need for the Checker Framework to include an annotated version of the library.
     You will probably want to write the annotations in comments, so that it is still possible to compile the library without use of Java 8.
     If the library maintainers do not accept the annotations, then periodically, such as when a new version of the library is released, pull changes from upstream (the library's main version control system) into your fork, add annotations to any newly-added methods in classes that are annotated with `@AnnotatedFor`, rebuild to create an updated `.jar` file, and inform the Checker Framework developers by opening an issue or issuing a pull request.

28.2 Using stub classes

A stub file contains "stub classes" that contain annotated signatures, but no method bodies. A checker uses the annotated signatures at compile time, instead of or in addition to annotations that appear in the library.

Section 28.2.3 describes how to create stub classes. Section 28.2.1 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 to handle `java.lang.String` for the Interning Checker, but you might do something similar for a different class and/or checker.

28.2.1 Using a stub file

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

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

Each stub path entry is a file or a directory; specifying a directory is equivalent to specifying every file in it whose name ends with `.astub`. The stub path entries are delimited by `File.pathSeparator` (`':'` for Linux and Mac, `';'` for Windows).

A checker automatically reads the stub file `jdk.astub`, unless command-line option `-Aignorejdkastub` is supplied. (The checker author should place `jdk.astub` in the same directory as the Checker class, i.e., the subclass of `BaseTypeVisitor`.) Programmers should only use the `-Astubs` argument for additional stub files they create themselves.

If a method appears in more than one stub file (or twice in the same stub file), then the annotations are merged. If any of the methods have different annotations from the same hierarchy on the same type, then the annotation from the last declaration is used.

28.2.2 Stub file format

Every 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.

As an illustration, a stub file for the Interning type system (Chapter 5) could be:

```
import org.checkerframework.checker.interning.qual.Interned;
package java.lang;
@Interned class Class<T> { }
class String {
  @Interned String intern();
}
```

Note, annotations in comments are ignored.

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 library's `.class` files. (If you are using a stub class, then typically the library is unannotated.)

Declaration specifiers: Declaration specifiers (e.g., `public`, `final`, `volatile`) may be omitted.

Return types: The return type of a method does not need to match the real method. In particular, it is valid to use `java.lang.Object` for every method. This simplifies the creation of stub files.

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

Enum constants in annotations need to be either fully qualified or imported. For example, one has to either write the enum constant `ANY` in fully-qualified form:

```
@Source(sparta.checkers.quals.FlowPermission.ANY)
```

or correctly import the enum class:

```
import sparta.checkers.quals.FlowPermission;
...
@Source(FlowPermission.ANY)
```

or statically import the enum constants:

```
import static sparta.checkers.quals.FlowPermission.*;
...
@Source(ANY)
```

Importing all packages from a class (`import my.package.*;`) only considers annotations from that package; enum types need to be explicitly imported.

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.

28.2.3 Creating a stub file

If you have access to the Java source code

Every Java file is a stub file. If you have access to the Java file, then you can use the Java file as the stub file. Just add annotations to the signatures, leaving the method bodies unchanged. The stub file parser silently ignores any annotations that it cannot resolve to a type, so don't forget the `import` statement.

Optionally (but highly recommended!), run the type-checker to verify that your annotations are correct. When you run the type-checker on your annotations, there should not be any stub file that also contains annotations for the class. In particular, if you are type-checking the JDK itself, then you should use the `-Aignorejdkastub` command-line option.

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.

If you do not have access to the Java source code

If you do not have access to the library source code, then you can create a stub file from the class file (Section 28.2.3), and then annotate it. The rest of this section describes this approach.

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

   ```
   cd nullness-stub
   java org.checkerframework.framework.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 org.checkerframework.checker.interning.qual.*;
   ```

   The stub file parser silently ignores any annotations that it cannot resolve to a type, so don't forget the import statement. Use the `-AstubWarnIfNotFound` command-line option to see warnings if an entry could not be found.
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 28.2.2.

28.2.4 Troubleshooting stub libraries

Type-checking does not yield the expected results

By default, the stub parser silently ignores annotations on unknown classes and methods. The stub parser also silently ignores unknown annotations, so don't forget to `import` any annotations.

Use command-line option `-AstubWarnIfNotFound` to warn whenever some element of a stub file cannot be found.

The `@NoStubParserWarning` annotation on a package or type in a stub file overrides the `-AstubWarnIfNotFound` command-line option, and no warning will be issued.

Use command-line option `-AstubDebug` to output debugging messages while parsing stub files, including about unknown classes, methods, and annotations. This overrides the `@NoStubParserWarning` annotation.

Problems parsing stub libraries

When using command-line option `-AstubWarnIfNotFound`, an error is issued if a stub file has a typo or the API method does not exist.

Fix this error by removing the extra L in the method name:

```
StubParser: Method isLLowerCase(char) not found in type java.lang.Character
```

Fix this error by removing the method `enableForgroundNdefPush(...)` from the stub file, because it is not defined in class `android.nfc.NfcAdapter` in the version of the library you are using:

```
StubParser: Method enableForegroundNdefPush(Activity,NdefPushCallback)
      not found in type android.nfc.NfcAdapter
```

28.2.5 Limitations

The stub file reader has several limitations. We will fix these in a future release.

- The receiver is written after the method parameter list, instead of as an explicit first parameter. That is, instead of

  ```
  returntype methodname(@Annotations C this, params);
  ```

  in a stub file one has to write

```
returntype methodname(params) @Annotations;
```

- The stub file reader does not handle nested classes. To work around this, it permits a top-level class to be written with a `$` in its name, and applies the annotations to the appropriate nested class.
- Annotations must be written before the package name on a fully qualified types rather than directly on the type it qualifies. However, it is usually not necessary to write the fully qualified name.

```
void init(@Nullable java.security.SecureRandom random);
```

- Annotations can only use string, boolean, or integer literals; other literals are not yet supported.

If these limitations are a problem, then you should insert annotations in the library's `.class` files instead.

28.3 Troubleshooting/debugging annotated libraries

Sometimes, it may seem that a checker is treating a library as unannotated even though the library has annotations. The compiler has two flags that may help you in determining whether library files are read, and if they are read whether the library's annotations are parsed.

`-verbose` Outputs info about compile phases — when the compiler reads/parses/attributes/writes any file. Also outputs the classpath and sourcepath paths.

`-XDTA:parser` (which is equivalent to `-XDTA:reader` plus `-XDTA:writer`) Sets the internal `debugJSR308` flag, which outputs information about reading and writing.

Chapter 29

How to create a new checker

This chapter 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 chapter.

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

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

This checker is so short because it builds on the Subtyping Checker (Chapter 22). See Section 22.2 for more details about this particular checker. When you wish to create a new checker, it is often easiest to begin by building it declaratively on top of the Subtyping Checker, and then return to this chapter when you need more expressiveness or power than the Subtyping Checker affords.

You can also create your own checker by customizing a different existing checker. Specific checkers that are designed for extension (besides the Subtyping Checker) include the Fake Enumeration Checker (Chapter 7, page 56), the Units Checker (Chapter 15, page 86), and a typestate checker (Chapter 23.1, page 111). Or, you can copy and then modify a different existing checker — whether one distributed with the Checker Framework or a third-party one.

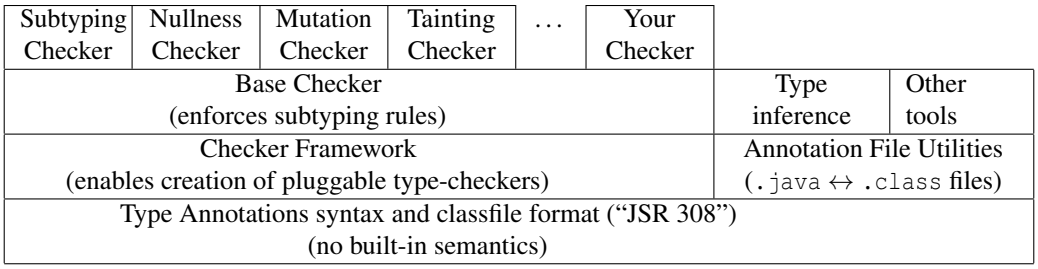
You can place your checker's source files wherever you like. When you compile your checker, `$CHECKERFRAMEWORK/framework/dist/framework.jar` and `$CHECKERFRAMEWORK/framework/dist/javac.jar` should be on your classpath. (If you wish to modify an existing checker in place, or to place the source code for your new checker in your own private copy of the Checker Framework source code, then you need to be able to re-compile the Checker Framework, as described in Section 32.3.)

The rest of this chapter 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 chapter 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/api/`.

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. For examples, see Chapters 23.1 and 23.

29.1 Relationship of the Checker Framework to other tools

This table shows the relationship among various tools. All of the tools support the Java 8 type annotation syntax. You use the Checker Framework to build pluggable type systems, and the Annotation File Utilities to manipulate `.java` and `.class` files.



The Base Checker enforces the standard subtyping rules on extended types. The Subtyping Checker is a simple use of the Base Checker that supports providing type qualifiers on the command line. You usually want to build your checker on the Base Checker.

29.2 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 29.3–29.7 describe the components of a type system as written using the Checker Framework:

29.3 **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`).

29.4 **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`. Optionally, write dataflow rules to enhance flow-sensitive type qualifier inference (Section 29.5).

29.6 **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 Base Checker (Chapter 22).
- 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.

29.7 **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.

29.3 Annotations: Type qualifiers and hierarchy

A type system designer specifies the qualifiers in the type system (Section 29.3.1) and the type hierarchy that relates them. The type hierarchy — the subtyping relationships among the qualifiers — can be defined either declaratively via meta-annotations (Section 29.3.2), or procedurally through subclassing `QualifierHierarchy` or `TypeHierarchy` (Section 29.3.3).

29.3.1 Defining the type qualifiers

Type qualifiers are defined as Java annotations [Dar06]. In Java, an annotation is defined using the Java `@interface` keyword. For example:

```
// Define an annotation for the @NonNull type qualifier.
@TypeQualifier
@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})
public @interface NonNull { }
```

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

Your type system should include a top qualifier and a bottom qualifier (Section 29.3.5). You should also define a polymorphic qualifier `@Poly`*`MyTypeSystem`* (Section 24.2).

29.3.2 Declaratively defining the qualifier 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_USE, ElementType.TYPE_PARAMETER})
  @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 19.2), `@Mutable` and `@Immutable` induce two mutually exclusive subtypes of the `@ReadOnly` qualifier.
  All type qualifiers, except for polymorphic qualifiers (see below and also Section 24.2), need to be properly annotated with `SubtypeOf`.
  The top qualifier is annotated with `@SubtypeOf( { } )`. The top qualifier is the qualifier that is a supertype of all other qualifiers. For example, `@Nullable` is the top qualifier of the Nullness type system, hence is defined as:

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

  If the top qualifier of the hierarchy is the unqualified type, then its children will use `@SubtypeOf(Unqualified.class)`, but no `@SubtypeOf( { } )` annotation on the top qualifier is necessary. For an example, see the `Encrypted` type system of Section 22.2.
- `@PolymorphicQualifier` denotes that a qualifier is a polymorphic qualifier. For example:

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

  For a description of polymorphic qualifiers, see Section 24.2. 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 29.3.3) by overriding the `isSubtype` method in the checker class (Section 29.7). However, the declarative mechanism is sufficient for most type systems.

29.3.3 Procedurally defining the qualifier hierarchy

While the declarative syntax suffices for many cases, more complex type hierarchies can be expressed by overriding, in your subclass of `BaseTypeVisitor`, 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<@QReadOnly Date>`).

29.3.4 Defining a default annotation

A type system applies a default qualifier where the user has not written a qualifier (and no implicit qualifier is applicable), as explained in Section 25.3.1.

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.addAbsoluteDefault` method. You may do this even if you have declaratively defined the qualifier hierarchy; see the Nullness Checker's implementation for an example.

29.3.5 Completeness of the type hierarchy

When you define a type system, its type hierarchy must be a complete lattice — that is, there must be a top type that is a supertype of all other types, and there must be a bottom type that is a subtype of all other types. Furthermore, it is best if the top type and bottom type are defined explicitly for the type system, rather than (say) reusing a qualifier from the Checker Framework such as `@Unqualified`.

It is possible that a single type-checker checks multiple type hierarchies. An example is the Nullness Checker, which has three separate type hierarchies, one each for nullness, initialization, and map keys. In this case, each type hierarchy would have its own top qualifier and its own bottom qualifier; they don't all have to share a single top qualifier or a single bottom qualifier.

Bottom qualifier Your type hierarchy must have a bottom qualifier — a qualifier that is a (direct or indirect) subtype of every other qualifier.

Your type system must give `null` the bottom type. (The only exception is if the type system has special treatment for `null` values, as the Nullness Checker does.) This legal code will not type-check unless `null` has the bottom type:

```
<T> T f() {
    return null;
}
```

You don't necessarily have to define a new bottom qualifier. You can use `org.checkerframework.framework.qual.Bottom` if your type system does not already have an appropriate bottom qualifier.

If your type system has a special bottom type that is used *only* for the `null` value, then users should never write the bottom qualifier explicitly. To ensure this, write `@Target({})` on the definition of the bottom qualifier.

The hierarchy shown in Figure 19.1 lacks a bottom qualifier, because there is no qualifier that is a subtype of both `@Immutable` and `@Mutable`. The actual IGJ hierarchy does contain 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 Your type hierarchy must have a top qualifier — a qualifier that is a (direct or indirect) supertype of every other qualifier. Here is the reason. The default type for local variables is the top qualifier (that type is then flow-sensitively refined depending on what values are stored in the local variable). If there is no single top qualifier, then there is no unambiguous choice to make for local variables.

Furthermore, it is most convenient to users if the top qualifier is defined by the type system. It is possible to use the framework's `@Unqualified` as the top type, but this is poor practice. Users lose flexibility in expressing defaults: there is no way for a user to change the default qualifier for just that type system. If a user specifies `@DefaultQualifier(Unqualified.class)`, then the default would apply to every type system that uses `@Unqualified`, which is unlikely to be desired.

29.4 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.

29.4.1 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 in `ImplicitFor`:

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

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

```
@TypeQualifier
@SubtypeOf( { Nullable.class } )
@ImplicitFor(
  types={TypeKind.PACKAGE},
  typeClasses={AnnotatedPrimitiveType.class},
  trees={
    Tree.Kind.NEW_CLASS,
    Tree.Kind.NEW_ARRAY,
    Tree.Kind.PLUS,
    // All literals except NULL_LITERAL:
```

```
    Tree.Kind.BOOLEAN_LITERAL, Tree.Kind.CHAR_LITERAL, Tree.Kind.DOUBLE_LITERAL, Tree.Kind.FLOAT_LITERAL,
    Tree.Kind.INT_LITERAL, Tree.Kind.LONG_LITERAL, Tree.Kind.STRING_LITERAL
  })
@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})
public @interface NonNull {  }


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

29.4.2 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.

29.5 Dataflow: enhancing flow-sensitive type qualifier inference

By default, every checker performs automatic type refinement, also known as flow inference, as described in Section 25.4.

In the uncommon case that you wish to disable flow inference in your checker, put the following two lines at the beginning of the constructor for your subtype of `BaseAnnotatedTypeFactory`:

```
// use true to enable flow inference, false to disable it
super(checker, false);
```

You can enhance the Checker Framework's built-in flow-sensitive type refinement, so that it is more powerful and is customized to your type system. In particular, your enhancement will yield a more refined type for certain expressions. However, most enhancements to type refinement are based on a run-time test specific to the type system and not all type-systems have applicable run-time tests. See Section 25.4.1 (page 134) to determine if run-time tests are applicable to your type system.

The Checker Framework's type refinement is implemented with a dataflow algorithm which can be customized to enhance the built-in type refinement. The next sections detail dataflow customization. It would also be helpful to read the Dataflow Manual, which gives a more in-depth description of the Checker Framework's dataflow framework.

The steps to customizing type refinement are:

1. 29.5.1 Create required classes and configure their use
2. 29.5.2 Override methods that handle `Node`s of interest
3. 29.5.3 Determine which expressions will be refined
4. 29.5.4 Implement the refinement

The Regex Checker's dataflow customization for the `RegexUtil.asRegex` run-time check is used as an example throughout the steps.

The `RegexUtil.asRegex` method is declared as:

```
@Regex(0) String asRegex(String s, int groups) { ... }
```

which means that an expression such as `RegexUtil.asRegex(myString, myInt)` has type `@Regex(0)` String. When `int` parameter `group` is known or can be inferred at compile time, a better estimate can be given. For example, `RegexUtil.asRegex(myString, 2)` has type `@Regex(2)` String.

29.5.1 Create required classes and configure their use

The following classes must be created to customize dataflow. These classes must be included on the classpath like other components of your checker.

1. **Create a class that extends `CFAbstractTransfer`**
   `CFAbstractTransfer` performs the default Checker Framework type refinement. The extended class will add functionality by overriding superclass methods.
   The Regex Checker's extended `CFAbstractTransfer` is `RegexTransfer`.
2. **Create a class that extends `CFAbstractAnalysis` and uses the extended `CFAbstractTransfer`**
   `CFAbstractTransfer` and its superclass, `Analysis`, are the central coordinating classes in the Checker Framework's dataflow algorithm. The `createTransferFunction` method must be overridden in an extended `CFAbstractTransfer` to return a new instance of the extended `CFAbstractTransfer`.
   The Regex Checker's extended `CFAbstractAnalysis` is `RegexAnalysis`, which overrides the `createTransferFunction` to return a new `RegexTransfer` instance:
   ```
   @Override
   public RegexTransfer createTransferFunction() {
       return new RegexTransfer(this);
   }
   ```
3. **Configure the checker's type factory to use the extended `CFAbstractAnalysis`**
   To configure your checker's type factory to use the new extended `CFAbstractAnalysis`, override the `createFlowAnalysis` method in your type factory to return a new instance of the extended `CFAbstractAnalysis`.
   ```
   @Override
   protected RegexAnalysis createFlowAnalysis(
           List<Pair<VariableElement, CFValue>> fieldValues) {

       return new RegexAnalysis(checker, this, fieldValues);
   }
   ```

29.5.2 Override methods that handle Nodes of interest

At this point, your checker is configured to use your extended `CFAbstractAnalysis`, but it uses only the default behavior. Next, in your extended `CFAbstractTransfer` override the visitor method that handles the `Node`s relevant to your run-time check or run-time operation can be used to refine types.

A `Node` is basically equivalent to a javac compiler `Tree`. A tree is a node in the abstract syntax tree of the program being checked. See 29.11 for more information about trees.

A `Node` generally maps one-to-one with a `Tree`. When dataflow processes a method, it translates `Tree`s into `Node`s and then calls the appropriate visit method on `CFAbstractTransfer` which then performs the dataflow analysis for the passed in `Node`.

Decide what `Node` kinds are of interest with respect to the run-time checks or run-time operations you are trying to support. The `Node` subclasses can be found in the `org.checkerframework.dataflow.cfg.node` package. Some examples are `EqualToNode`, `LeftShiftNode`, `VariableDeclarationNode`.

The Regex Checker refines the type of a run-time test method call, so `RegexTransfer` overrides the method that handles `MethodInvocationNode`s, `visitMethodInvocation`.

```
public TransferResult<CFValue, CFStore> visitMethodInvocation(
  MethodInvocationNode n, TransferInput<CFValue, CFStore> in)  { ... }
```

29.5.3 Determine the expressions to refine the types of

There are usually multiple expressions used in a run-time check or run-time operation; determine which expression the customization will refine. This is usually specific to the type system and run-time test.

Expressions are refined by modifying the return value of a visitor method in `CFAbstractTransfer`. `CFAbstractTransfer` visitor methods return a `TransferResult`. The constructor of a `TransferResult` takes two parameters: the resulting type for the `Node` being evaluated (the result type) and a map from expressions in scopes to estimates of their types (a `Store`).

For the program operation `op(a,b)`, an enhancement may improve the Checker Framework's types by:

1. Changing the resulting type to refine the estimate of the type of entire expression `op(a,b)`, or
2. Changing the store to refine the estimate of some other expression, such as `a` or `b`.

Changing the `TransferResult`'s result type changes the type that is returned by the `AnnotatedTypeFactory` for the tree corresponding to the `Node` that was visited. (Remember that `BaseTypeVisitor` uses the `AnnotatedTypeFactory` to look up the type of a `Tree`, and then performs checks on types of one or more `Trees`).

When `RegexTransfer` evaluates a `RegexUtils.asRegex` invocation, it updates the `TransferResult`'s result type. This changes the type of the `RegexUtils.asRegex` invocation when it's `Tree` is looked up by the `AnnotatedTypeFactory`. Regex Checker's `visitMethodInvocation` is shown in more detail in 29.5.4.

Updating the `Store` treats an expression as a having a refined type for the remainder of the method or conditional block. For example, when the Nullness Checker's dataflow evaluates `myvar != null`, it updates the `Store` to specify that the variable `myvar` should be treated as having type `@NonNull` for the rest of the then conditional block. Not all kinds of expressions can be refined; currently method return values, local variables, fields, and array values can be stored in the `Store`. Other kinds of expressions, like binary expressions or casts, cannot be stored in the `Store`.

Both the `Store` and the result type may be updated in the same `TransferResult`.

29.5.4 Implement the refinement

This section details implementing the visitor method `RegexTransfer.visitMethodInvocation` for the `RegexUtil.asRegex` run-time test. You can find other examples of visitor methods in `LockTransfer` and `FormatterTransfer`.

A general outline of the visit method is to:

1. Determine if the visited `Node` is of interest
2. Determine the refined type
3. Return a `TransferResult` with the refined types

1. **Determine if the visited `Node` is of interest**
   The visitor method for a `Node` is invoked for all instances of that `Node` kind in the program, so the `Node` must be inspected to determine if it is an instance of the desired run-time test or operation. For example, `visitMethodInvocation` is called when dataflow processes any method invocation, but the `RegexTransfer` should only refine the result of `RegexUtils.asRegex` invocations:

```
@Override
public TransferResult<CFValue, CFStore> visitMethodInvocation(...)
  ...
  MethodAccessNode target = n.getTarget();
  ExecutableElement method = target.getMethod();
  Node receiver = target.getReceiver();
  if (receiver instanceof ClassNameNode) {
    ClassNameNode cn = (ClassNameNode) receiver;
    String receiverName = cn.getElement().toString();

    // Is this a RegexUtil.isRegex(s, groups) method call?
```

```
if (isRegexUtil(receiverName)) {
  if (ElementUtils.matchesElement(method,
    null, IS_REGEX_METHOD_NAME, String.class, int.class)) {

    ...
```

2. **Determine the refined type**
   Some run-time tests, like the null comparison test, have a deterministic type refinement, e.g. the Nullness Checker always refines the argument in the expression to `@NonNull`. However, sometimes the refined type is dependent on the parts of run-time test or operation itself, such as arguments passed to it.
   For example, the refined type of `RegexUtils.asRegex` is dependent on the integer argument to the method call. The `RegexTransfer` uses this argument to build the resulting type `@Regex(i)`, where `i` is the value of the integer argument. Note that currently this code only uses the value of the integer argument if the argument was an integer literal. It could be extended to use the value of the argument if it was any compile-time constant or was inferred at compile time by another analysis, such as the 16.

   ```
   AnnotationMirror regexAnnotation;
   Node count = n.getArgument(1);
   if (count instanceof IntegerLiteralNode) {
     IntegerLiteralNode iln = (IntegerLiteralNode) count;
     Integer groupCount = iln.getValue();
     regexAnnotation = factory.createRegexAnnotation(groupCount);
   ```

   If the integer argument was not a literal integer, the `RegexTransfer` falls back to refining the type to just `@Regex(0)`.

   ```
   } else {
     regexAnnotation = AnnotationUtils.fromClass(factory.getElementUtils(), Regex.class);
   }
   ```

3. **Return a `TransferResult` with the refined types**
   As discussed in section 29.5.3, the type of an expression is refined by modifying the `TransferResult`. Since the `RegexTransfer` is updating the type of the run-time test itself, it will update the result type and not the `Store`. A `CFValue` is created to hold the type inferred. `CFValue` is a wrapper class for values being inferred by dataflow:

   ```
   CFValue newResultValue = analysis.createSingleAnnotationValue(regexAnnotation,
       result.getResultValue().getType().getUnderlyingType());
   ```

   Then, RegexTransfer's `visitMethodInvocation` creates and returns a `TransferResult` using `newResultValue` as the result type.

   ```
   return new RegularTransferResult<>(newResultValue, result.getRegularStore());
   ```

   Finally, when the Regex Checker encounters a `RegexUtils.asRegex` method call, the checker will refine the return type of the method if it can determine the value of the integer parameter at compile time.

29.6 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 Oracle'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 (type.argument.type.incompatible) 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

29.6.1 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[1], 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 30.6).

29.6.2 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 28), and leave the work to the standard type-checking rules.

29.7 The checker class: Compiler interface

A checker's entry point is a subclass of `SourceChecker`, and is usually a direct subclass of either `BaseTypeChecker` or `AggregateChecker`. 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 Checker and the Nullness Checker:

[1] Actually, there is a 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.

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

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

The checker class must indicate the annotations that make up the type hierarchy for this checker (including polymorphic qualifiers), either via a `@TypeQualifiers` annotation or by overriding the `createSupportedTypeQualifiers` method. Each argument to `@TypeQualifiers` or value returned by `createSupportedTypeQualifiers` is a class literal for a type qualifier whose definition bears the `@TypeQualifier` meta-annotation. An aggregate checker (which extends `AggregateChecker`) does not need to specify its type qualifiers, but each of its component checkers should do so.

The checker class bridges between the compiler and the rest of the checker. It invokes the type-rule check visitor on every Java source file being compiled, and provides a simple API, `SourceChecker.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. For more details about message keys, see Section 26.1.3 (page 141).

29.7.1 Bundling multiple checkers

Sometimes, multiple checkers work together and should always be run together. There are two different ways to bundle multiple checkers together, by creating an "aggregate checker" or a "compound checker".

1. An aggregate checker runs multiple independent, unrelated checkers. There is no communication or cooperation among them.
   The effect is the same as if a user passes multiple processors to the `-processor` command-line option.
   For example, instead of a user having to run

   ```
   javac -processor DistanceUnitChecker,VelocityUnitChecker,MassUnitChecker ... files ...
   ```

   the user can write

   ```
   javac -processor MyUnitCheckers ... files ...
   ```

   if you define an aggregate checker class. Extend `AggregateChecker` and override the `getSupportedTypeCheckers` method, like the following:

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

An example of an aggregate checker is `I18nChecker` (see Chapter 12.2, page 78), which consists of `I18nSubchecker` and `LocalizableKeyChecker`.

2. Use a compound checker to express dependencies among checkers. Suppose it only makes sense to run MyChecker if MyHelperChecker has already been run; that might be the case if MyHelperChecker computes some information that MyChecker needs to use.
   Override `MyChecker.getImmediateSubcheckerClasses` to return a list of the checkers that MyChecker depends on. Every one of them will be run before MyChecker is run. One of MyChecker's subcheckers may itself be a compound checker, and multiple checkers may declare a dependence on the same subchecker. The Checker Framework will run each checker once, and in an order consistent with all the dependences.
   A checker obtains information from its subcheckers (those that ran before it) by querying their `AnnotatedTypeFactory` to determine the types of variables.

29.7.2 Providing command-line options

A checker can provide two kinds of command-line options: boolean flags and named string values (the standard annotation processor options).

Boolean flags

To specify a simple boolean flag, add:

```
@SupportedLintOptions({"flag"})
```

to your checker subclass. The value of the flag can be queried using

```
checker.getLintOption("flag", false)
```

The second argument sets the default value that should be returned.
To pass a flag on the command line, call javac as follows:

```
javac -processor Mine -Alint=flag
```

Named string values

For more complicated options, one can use the standard annotation processing `@SupportedOptions` annotation on the checker, as in:

```
@SupportedOptions({"info"})
```

The value of the option can be queried using

```
checker.getOption("info")
```

To pass an option on the command line, call javac as follows:

```
javac -processor Mine -Ainfo=p1,p2
```

The value is returned as a single string and you have to perform the required parsing of the option.

29.8 Testing framework

The Checker Framework provides a convenient way to write tests for your checker. It is extensively documented in file `checker-framework/checker/tests/README`.

29.9 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.

29.9.1 Amount of detail in messages

- `-AprintAllQualifiers`: print all type qualifiers, including qualifiers like `@Unqualified` which are usually not shown. (Use the `@InvisibleQualifier` meta-annotation on a qualifier to hide it.)
- `-Adetailedmsgtext`: Output error/warning messages in a stylized format that is easy for tools to parse. This is useful for tools that run the Checker Framework and parse its output, such as IDE plugins. See the source code of `SourceChecker.java` for details about the format.
- `-AprintErrorStack`: print a stack trace whenever an internal Checker Framework error occurs.
- `-Anomsgtext`: use message keys (such as "`type.invalid`") rather than full message text when reporting errors or warnings. This is used by the Checker Framework's own tests, so they do not need to be changed if the English message is updated.

29.9.2 Stub and JDK libraries

- `-Aignorejdkastub`: ignore the `jdk.astub` file in the checker directory. Files passed through the `-Astubs` option are still processed. This is useful when experimenting with an alternative stub file.
- `-Anocheckjdk`: don't issue an error if no annotated JDK can be found.
- `-AstubDebug`: Print debugging messages while processing stub files.

29.9.3 Progress tracing

- `-Afilenames`: print the name of each file before type-checking it.
- `-Ashowchecks`: print debugging information for each pseudo-assignment check (as performed by `BaseTypeVisitor`; see Section 29.6.

29.9.4 Saving the command-line arguments to a file

- `-AoutputArgsToFile`: This saves the final command-line parameters as passed to the compiler in a file. This file can be used as a script (if the file is marked as executable on Unix, or if it includes a `.bat` extension on Windows) to re-execute the same compilation command. This is useful, for example, when debugging problems running the Checker Framework from Maven, since normally the command-line parameters used by Maven are not user-visible. Note that this argument cannot be included in a file containing command-line arguments passed to the compiler using the @argfile syntax. Please see Section 30.3.1 for more details on how to use this command-line parameter to debug compilation using Maven.
  Example usage: `-AoutputArgsToFile=/home/username/scriptfile`

29.9.5 Miscellaneous debugging options

- `-Aflowdotdir`: Directory for .dot files that visualize the control flow graph of all the methods and code fragments analyzed by the dataflow analysis. The graph also contains information about flow-sensitively refined types of various expressions at many program points.
- `-AresourceStats`: Whether to output resource statistics at JVM shutdown.

29.9.6 Examples

The following example demonstrates how these options are used:

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

[InterningChecker] InterningExampleWithWarnings.java
 success (line  18): STRING_LITERAL "foo"
     actual: DECLARED @org.checkerframework.checker.interning.qual.Interned java.lang.String
   expected: DECLARED @org.checkerframework.checker.interning.qual.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 @org.checkerframework.checker.interning.qual.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 Checker Framework javac.jar (resides in `.../checker-framework/checker/dist/javac`. 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.

You can also set up remote (or local) debugging using the following command as a template:

```
java -jar $CHECKERFRAMEWORK/framework/dist/framework.jar \
    -J-Xdebug -J-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005 \
    -processor org.checkerframework.checker.nullness.NullnessChecker \
    src/sandbox/FileToCheck.java
```

29.10 Documenting the checker

This section describes how to write a chapter for this manual that describes a new type-checker. This is a prerequisite to having your type-checker distributed with the Checker Framework, which is the best way for users to find it and for it to be kept up to date with Checker Framework changes. Even if you do not want your checker distributed with the Checker Framework, these guidelines may help you write better documentation.

When writing a chapter about a new type-checker, see the existing chapters for inspiration. (But recognize that the existing chapters aren't perfect: maybe they can be improved too.)

A chapter in the Checker Framework manual should generally have the following sections:

Chapter: Belly Rub Checker The text before the first section in the chapter should state the guarantee that the checker provides and why it is important. It should give an overview of the concepts. It should state how to run the checker.

Section: Belly Rub Annotations This section includes descriptions of the annotations with links to the Javadoc. Separate type annotations from declaration annotations, and put any type annotations that a programmer may not write (they are only used internally by the implementation) last within variety of annotation.

Draw a diagram of the type hierarchy. A textual description of the hierarchy is not sufficient; the diagram really helps readers to understand the system.

The Javadoc for the annotations deserves the same care as the manual chapter. Each annotation's Javadoc comment should use the `@checker_framework.manual` Javadoc taglet to refer to the chapter that describes the checker; see `ManualTaglet`.

Section: What the Belly Rub Checker checks This section gives more details about when an error is issued, with examples. This section may be omitted if the checker does not contain special type-checking rules — that is, if the checker only enforces the usual Java subtyping rules.

Section: Examples Code examples.

Sometimes you can omit some of the above sections. Sometimes there are additional sections, such as tips on suppressing warnings, comparisons to other tools, and run-time support.

You will create a new `belly-rub-checker.tex` file, then `\input` it at a logical place in `manual.tex` (not necessarily as the last checker-related chapter). Also add two references to the checker's chapter: one at the beginning of chapter 1, and identical text in Section 25.4.1 (both of these lists appear in the same order as the manual chapters, to help us notice if anything is missing).

Every chapter and (sub)*section should have a label defined *within* the `\section` command. Section labels should start with the checker name (as in `\label{bellyrub-examples}`) and not with "`sec:`". These conventions are for the benefit of the Hevea program that produces the HTML version of the manual.

Don't forget to write Javadoc for any annotations that the checker uses. That is part of the documentation and is the first thing that many users may see. Also ensure that the Javadoc links back to the manual, using the `@checker_framework.manual` custom Javadoc tag.

You should also integrate your new checker with the Eclipse plugin.

29.11 javac implementation survival guide

Since this section of the manual was written, the useful "The Hitchhiker's Guide to javac" has become available at `http://openjdk.java.net/groups/compiler/doc/hhgtjavac/index.html`. See it first, and then refer to this section. (This section of the manual should be revised, or parts eliminated, in light of that document.)

A checker built using the Checker Framework makes use of a few interfaces from the underlying compiler (Oracle's OpenJDK javac). This section describes those interfaces.

29.11.1 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. A checker should use the Checker 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 (see below). 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 unit in the source code, like a method declaration, statement, block, `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 a method invocation, and `ForEachTree` for an enhanced-for-loop statement.

You should limit your use of trees. A checker uses Trees mainly to traverse the source code and retrieve the types/elements corresponding to them. Then, the checker performs 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 determine 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 Oracle 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` suffix instead (e.g., `TypesUtils`, `TreeUtils`, `ElementUtils`), with one notable exception: `AnnotatedTypes`.

29.11.2 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 processor 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 8 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 Oracle's Tree API to access a program's AST. The Tree API is specific to the Oracle OpenJDK, so the Checker Framework only works with the OpenJDK 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 29.6.1.

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 document, 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`).

29.12 Integrating a checker with the Checker Framework

To integrate a new checker with the Checker Framework release, perform the following:

- Add a `XXX-tests` build target and ensure all tests pass.
- Make sure `all-tests` tests the new checker.
- Extend the `check-compilermsgs` target to include the compiler messages property file of the new checker in the `checker-args` list.
- Make sure `check-compilermsgs` and `check-purity` run without warnings or errors.

Chapter 30

Integration with external tools

This chapter discusses how to run a checker from the command line, from a build system, or from an IDE. You can skip to the appropriate section:

- javac (Section 30.1)
- Ant (Section 30.2)
- Maven (Section 30.3)
- Gradle (Section 30.4)
- IntelliJ IDEA (Section 30.5)
- Eclipse (Section 30.6)
- tIDE (Section 30.7)

If your build system or IDE is not listed above, you should customize how it runs the javac command on your behalf. See your build system or IDE documentation to learn how to customize it, adapting the instructions for javac in Section 30.1. 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 chapter also discusses type inference tools (see Section 30.8).

All examples in this chapter are in the public domain, with no copyright nor licensing restrictions.

30.1 Javac compiler

If you use the `javac` compiler from the command line, then you can instead use the "Checker Framework compiler", a variant of the OpenJDK `javac` that recognizes type annotations in comments and that includes the Checker Framework jar files on its path. The Checker Framework compiler is backward-compatible, so using it as your Java compiler, even when you are not doing pluggable type-checking, has no negative consequences.

You can use the Checker Framework compiler in three ways. You can use any one of them. However, if you are using the Windows command shell, you must use the last one.

- Option 1: Add directory `.../checker-framework-1.9.4/checker/bin` to your path, *before* any other directory that contains a `javac` executable. Now, whenever you run `javac`, you will use the updated compiler. If you are using the bash shell, a way to do this is to add the following to your `~/.profile` (or alternately `~/.bash_profle` or `~/.bashrc`) file:

  ```
  export CHECKERFRAMEWORK=${HOME}/checker-framework-1.9.4
  export PATH=${CHECKERFRAMEWORK}/checker/bin:${PATH}
  ```

  then log out and back in to ensure that the environment variable setting takes effect.
- Option 2: Whenever this document tells you to run `javac`, you can instead run `$CHECKERFRAMEWORK/checker/bin/javac`. You can simplify this by introducing an alias. Then, whenever this document tells you to run `javac`, instead use that alias. Here is the syntax for your `~/.bashrc` file:

```
export CHECKERFRAMEWORK=${HOME}/checker-framework-1.9.4
alias javacheck='$CHECKERFRAMEWORK/checker/bin/javac'
```

If you are using a Java 7 JVM, then add command-line arguments to so indicate:

```
export CHECKERFRAMEWORK=${HOME}/checker-framework-1.9.4
alias javacheck='$CHECKERFRAMEWORK/checker/bin/javac -source 7 -target 7'
```

If you do not add the `-source 7 -target 7` command-line arguments, you may get the following error when running a class that was compiled by javacheck:

```
UnsupportedClassVersionError: ... : Unsupported major.minor version 52.0
```

- Option 3: Whenever this document tells you to run `javac`, instead run checker.jar via `java` (not `javac`) as in:

  ```
  java -jar $CHECKERFRAMEWORK/checker/dist/checker.jar ...
  ```

  You can simplify the above command by introducing an alias. Then, whenever this document tells you to run `javac`, instead use that alias. For example:

  ```
  # Unix
  export CHECKERFRAMEWORK=${HOME}/checker-framework-1.9.4
  alias javacheck='java -jar $CHECKERFRAMEWORK/checker/dist/checker.jar'

  # Windows
  set CHECKERFRAMEWORK = C:\Program Files\checker-framework-1.9.4\
  doskey javacheck=java -jar %CHECKERFRAMEWORK%\checker\dist\checker.jar $*
  ```

  and add `-source 7 -target 7` if you use a Java 7 JVM.
  (Explanation for advanced users: More generally, anywhere that you would use `javac.jar`, you can substitute `$CHECKERFRAMEWORK/checker/dist/checker.jar`; the result is to use the Checker Framework compiler instead of the regular `javac`.)

To ensure that you are using the Checker Framework compiler, run `javac -version` (possibly using the full pathname to `javac` or the alias, if you did not add the Checker Framework `javac` to your path). The output should be:

```
javac 1.8.0-jsr308-1.9.4
```

30.2 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"/>

<property name="checkerframework" value="${env.CHECKERFRAMEWORK}" />

<!-- On Mac/Linux, use the javac shell script; on Windows, use javac.bat -->
<condition property="cfJavac" value="javac.bat" else="javac">
    <os family="windows" />
</condition>

<presetdef name="jsr308.javac">
  <javac fork="yes" executable="${checkerframework}/checker/bin/${cfJavac}" >
    <!-- JSR-308-related compiler arguments -->
    <compilerarg value="-version"/>
    <compilerarg value="-implicit:class"/>
  </javac>
</presetdef>
```

2. **Duplicate** the compilation target, then **modify** it slightly as indicated in this example:

```
<target name="check-nullness"
        description="Check for null pointer dereferences"
        depends="clean,...">
  <!-- use jsr308.javac instead of javac -->
  <jsr308.javac ... >
    <compilerarg line="-processor org.checkerframework.checker.nullness.NullnessChecker"/>
    <!-- optional, to not check uses of library methods: <compilerarg value="-AskipUses=^(java\.awt\.|javax\.swing\.)"/> -->
    <compilerarg line="-Xmaxerrs 10000"/>
    ...
  </jsr308.javac>
</target>
```

Fill in each ellipsis (...) from the original compilation target.
In the example, the target is named `check-nullness`, but you can name it whatever you like.

30.2.1 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 Checker Framework compiler that is distributed with the Checker Framework.
   The `-version` compiler argument is just for debugging; you may omit it.
   The `-implicit:class` compiler argument causes annotation processing to be performed on implicitly compiled files. (An implicitly compiled file is one that was not specified on the command line, but for which the source code is newer than the `.class` file.) This is the default, but supplying the argument explicitly suppresses a compiler warning.
2. 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.

30.3 Maven

If you use the Maven tool, then you can specify pluggable type-checking as part of your build process. This is done by pointing Maven to a script that makes Maven use the Type Annotations compiler. These instructions use the artifacts from Maven Central.

1. Declare a dependency on the type qualifier annotations. Find the existing `<dependencies>` section and add a new `<dependencies>` item:

```
<dependencies>
    ... existing <dependency> items ...

    <!-- annotations from the Checker Framework: nullness, interning, locking, ... -->
    <dependency>
        <groupId>org.checkerframework</groupId>
        <artifactId>checker-qual</artifactId>
        <version>1.9.4</version>
    </dependency>

</dependencies>
```

2. Direct the Maven compiler plugin to use the `javac_maven` script. Change the reference to the `maven-compiler-plugin` within the `<plugins>` section, or add it if it is not present.

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
        <fork>true</fork>
        <executable>$env.CHECKERFRAMEWORK/checker/bin/javac_maven</executable>
      </configuration>
    </plugin>
  </plugins>
</build>
```

   On Windows, the `javac_maven.bat` script is automatically used instead of the `javac_maven` script – it is not necessary to include the `.bat` extension in the absolute path above.
   This script assumes that the Checker Framework is the only annotation processor being run. If this is not the case, please modify the `javac_maven` script accordingly.
3. Create a text file named `argfile` in the same directory as the `pom.xml` and include in it the command-line parameters to pass to the Java compiler.
   Example contents of `argfile`:

```
-processor org.checkerframework.checker.nullness.NullnessChecker
-AsuppressWarnings=purity.invalid.overriding
-Alint
-AprintErrorStack
-Awarns
-Xmaxwarns 10000
```

   Due to limitations of the `pom.xml` syntax, some command-line options are difficult or impossible to pass via the `pom.xml`, hence the need for this file.
   If you would like to call this file something other than `argfile`, please modify the `javac_maven` script accordingly.

30.3.1 Debugging the Maven compiler command-line arguments

Maven will sometimes hide important Checker Framework and/or Java compiler debugging output. If Maven is not producing the expected output when using it with the Type Annotations compiler and Checker Framework, it is possible to output the compiler command-line arguments produced by Maven to a file. This file can then be used as a script to execute the compiler in the same way Maven would have but without running Maven. This is done through the `-AoutputArgsToFile` command-line parameter. To use it with Maven, modify (or copy) the `javac_maven` script such that the last line in the script ends with:

```
[...] "-AoutputArgsToFile=<path to filename>" "$@"
```

For `javac_maven.bat`, modify (or copy) it such that the last line ends with:

```
[...] -AoutputArgsToFile=<path to filename> %*
```

Please see Section 29.9.4 for more details on how to use the resulting file.

30.4 Gradle

If you fork the compilation task, Gradle lets you specify the executable to compile java programs.

To specify the appropriate executable, set `options.fork = true` and `compile.options.fork.executable = "$CHECKERFRAMEWORK/checker/bin/javac"`

To specify command-line arguments, set `compile.options.compilerArgs`. Here is a possible example:

```
allprojects {
  tasks.withType(JavaCompile).all { JavaCompile compile ->
    compile.options.debug = true
    compile.options.compilerArgs = [
      '-version',
      '-implicit:class',
      '-processor', 'org.checkerframework.checker.nullness.NullnessChecker'
    ]
    options.fork = true
    options.forkOptions.executable = "$CHECKERFRAMEWORK/checker/bin/javac"
  }
}
```

30.5 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/`.

30.6 Eclipse

There are two ways to run a checker from within the Eclipse IDE: via Ant or using an Eclipse plugin. These two methods are described below.

No matter what method you choose, we suggest that all Checker Framework annotations be written in the comments if you are using a version of Eclipse that does not support Java 8. This will avoid many text highlighting errors with versions of Eclipse that don't support Java 8 and type annotations.

Even in a version of Eclipse that supports Java 8's type annotations, you still need to run the Checker Framework via Ant or via the plug-in, rather than by supplying the `-processor` command-line option to the ejc compiler. The reason is that the Checker Framework is built upon javac, and ejc represents the Java program differently. (If both javac and ejc implemented JSR 198 [Cro06], then it would be possible to build a type-checking plug-in that works with both compilers.)

30.6.1 Using an Ant task

Add an Ant target as described in Section 30.2. You can run the Ant target by executing the following steps (instructions copied from `http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.user%2FgettingStarted%2Fqs-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.

30.6.2 Eclipse plugin for the Checker Framework

The Checker framework Eclipse Plugin enables the use of the Checker Framework within the Eclipse IDE. Its website (`http://types.cs.washington.edu/checker-framework/eclipse/`). The website contains instructions for installing and using the plugin.

30.7 tIDE

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

30.8 Type inference tools

30.8.1 Varieties of type inference

There are two different tasks that are commonly called "type inference".

1. Type inference during type-checking (Section 25.4): 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 30.8.2): 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 25.4). Every method signature (arguments and return values) and field must have already been explicitly annotated, either by the programmer or by a separate type-checking tool (Section 30.8.2). This approach enables modular checking (one class or method at a time) and gives documentation benefits. The programmer still has to put in some effort, but much less than without inference: typically, a programmer does not have to write any qualifiers inside the body of a method.

30.8.2 Type inference to annotate a program

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

Section 3.3.7 lists several tools that infer annotations for the Nullness Checker.

Section 20.2.2 lists a tool that infers annotations for the Javari Checker, which detects mutation errors.

Cascade [VPEJ14] is an Eclipse plugin that implements interactive type qualifier inference. Cascade is interactive rather than fully-automated: it makes it easier for a developer to insert annotations. Cascade starts with an unannotated program and runs a type-checker. For each warning it suggests multiple fixes, the developer chooses a fix, and Cascade applies it. Cascade works with any checker built on the Checker Framework. You can find installation instructions and a video tutorial at `https://github.com/reprogrammer/cascade`.

Chapter 31

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.

Contents:

31.1: Motivation for pluggable type-checking
31.1.1: I don't make type errors, so would pluggable type-checking help me?
31.1.2: When should I use type qualifiers, and when should I use subclasses?

31.2: Getting started
31.2.1: How do I get started annotating an existing program?
31.2.2: Which checker should I start with?
31.2.3: Should I use pluggable types or Java subtypes?

31.3: Usability of pluggable type-checking
31.3.1: Are type annotations easy to read and write?
31.3.2: Will my code become cluttered with type annotations?
31.3.3: Will using the Checker Framework slow down my program? Will it slow down the compiler?
31.3.4: How do I shorten the command line when invoking a checker?

31.4: How to handle warnings
31.4.1: What should I do if a checker issues a warning about my code?
31.4.2: What does a certain Checker Framework warning message mean?
31.4.3: Can a pluggable type-checker guarantee that my code is correct?
31.4.4: What guarantee does the Checker Framework give for concurrent code?
31.4.5: How do I make compilation succeed even if a checker issues errors?
31.4.6: Why does the checker always say there are 100 errors or warnings?
31.4.7: Why does the Checker Framework report an error regarding a type I have not written in my program?
31.4.8: How can I do run-time monitoring of properties that were not statically checked?

31.5: Syntax of type annotations
31.5.1: What is a "receiver"?
31.5.2: What is the meaning of an annotation after a type, such as `@NonNull Object @Nullable`?
31.5.3: What is the meaning of array annotations such as `@NonNull Object @Nullable []`?
31.5.4: What is the meaning of a type qualifier at a class declaration?
31.5.5: Why shouldn't a qualifier apply to both types and declarations?

31.6: Semantics of type annotations
31.6.1: Why are the type parameters to `List` and `Map` annotated as `@NonNull`?
31.6.2: How can I handle typestate, or phases of my program with different data properties?
31.6.3: Why are explicit and implicit bounds defaulted differently?

31.7: Creating a new checker
31.7.1: How do I create a new checker?
31.7.2: Why is there no declarative syntax for writing type rules?

31.8: Relationship to other tools
31.8.1: Why not just use a bug detector (like FindBugs)?
31.8.2: How does the Checker Framework compare with Eclipse's Null Analysis?
31.8.3: How does pluggable type-checking compare with JML?
31.8.4: Is the Checker Framework an official part of Java?
31.8.5: What is the relationship between the Checker Framework and JSR 305?
31.8.6: What is the relationship between the Checker Framework and JSR 308?

31.1 Motivation for pluggable type-checking

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

Occasionally, a developer says that he makes no errors that type-checking 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 type-checking 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, wrong units of measurement, and many other errors. Every programmer makes errors sometimes and works with other people who do. 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. You should continue to use these. But type-checking is a good complement to these. Type-checking 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 type-checking gives you more time to focus on those.

31.1.2 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 31.2.3.

31.2 Getting started

31.2.1 How do I get started annotating an existing program?

See Section 2.4.1.

31.2.2 Which checker should I start with?

You should start with a property that matters to you. Think about what aspects of your code cause the most errors, or cost the most time during maintenance, or are the most common to be incorrectly-documented. Focusing on what you care about will give you the best benefits.

When you first start out with the Checker Framework, it's usually best to get experience with an existing type-checker before you write your own new checker.

Many users are tempted to start with the Nullness Checker (see Chapter 3, page 24), since null pointer errors are common and familiar. The Nullness Checker works very well, but be warned of three facts that make the absence of null pointer exceptions challenging to verify.

1. Dereferences happen throughout your codebase, so there are a lot of potential problems. By contrast, fewer lines of code are related to locking, regular expressions, etc., so those properties are easier to check.
2. Programmers use `null` for many different purposes. More seriously, programmers write run-time tests against `null`, and those are difficult for any static analysis to capture.
3. The Nullness Checker interacts with initialization and map keys.

If null pointer exceptions are most important to you, then by all means use the Nullness Checker. But if you just want to try *some* type-checker, there are others that are easier to use.

we do not recommend indiscriminately running all the checkers on your code. The reason is that each one has a cost — not just at compile time, but also in terms of code clutter and human time to maintain the annotations. If the property is important to you, is difficult for people to reason about, or has caused problems in the past, then you should run that checker. For other properties, the benefits may not repay the effort to use it. You will be the best judge of this for your own code, of course.

The Linear Checker (see Chapter 18, page 96) has not been extensively tested. The IGJ Checker (see Chapter 19, page 98), Javari Checker (see Chapter 20, page 102), and some of the third-party checkers (see Chapter 23, page 111) have known bugs that limit their usability. (Report the ones that affect you, and the Checker Framework developers will prioritize fixing them.)

31.2.3 Should I use pluggable types or Java subtypes?

For some programming tasks, you can use either a Java subclass or a type qualifier. As an example that your code currently uses `String` to represent an address. You could use Java subclasses by creating a new `Address` class and refactor your code to use it, or you could use type qualifiers by creating an `@Address` annotation and applying it to some uses of `String` in your code. As another example, suppose that your code currently uses `MyClass` in two different ways that should not interact with one another. You could use Java subclasses by changing MyClass into an interface or abstract class, defining two subclasses, and ensuring that neither subclass ever refers to the other subclass nor to the parent class.

If Java subclasses solve your problem, then that is probably better. We do not encourage you to use type qualifiers as a poor substitute for classes. An advantage of using classes is that the Java type-checker always runs; by contrast, it is possible to forget to run the pluggable type-checker. However, here are some reasons type qualifiers may be a better choice.

Backward compatibility 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. For code that strictly cannot be changed, you can add annotations in comments (see Section 27.2.1), or you can write library annotations (see Chapter 28).

Broader applicability Type annotations can be applied to primitives and to final classes such as `String`, which cannot be subclassed.

Richer semantics and new supertypes Type qualifiers permit you to remove operations, with a compile-time guarantee. An example is that an immutable version of a type prohibits calling mutator methods (see Chapters 19

and 20). More generally, type qualifiers permit creating a new supertype, not just a subtype, of an existing Java type.

More precise type-checking The Checker Framework is able to verify the correctness of code that the Java type-checker would reject. Here are a few examples.

- It uses a dataflow analysis to determine a more precise type for variables after conditional tests or assignments.
- It treats certain Java constructs more precisely, such as reflection (see Chapter 21).
- It includes special-case logic for type-checking specific methods, such as the Nullness Checker's treatment of `Map.get`.

Efficiency Type qualifiers have no run-time representation. Therefore, there is no space overhead for separate classes or for wrapper classes for primitives. There is no run-time overhead for due to extra dereferences or dynamic dispatch for methods that could otherwise be statically dispatched.

Less code clutter The programmer does not have to convert primitive types to wrappers, which would make the code both uglier and slower. Thanks to defaults and type inference (Section 25.3.1), you may be able to write and think in terms of the original Java type, rather than having to explicitly write one of the subtypes in all locations.

31.3 Usability of pluggable type-checking

31.3.1 Are type annotations easy to read and write?

The papers "Practical pluggable types for Java" [PAC+08] and "Building and using pluggable type-checkers" [DDE+11] discuss 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.

31.3.2 Will my code become cluttered with type annotations?

In summary: annotations do not clutter code; they are used much less frequently than generic types, which Java programmers find acceptable; and they reduce the overall volume of documentation that a codebase needs.

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 interning annotations (`@Interned`)
- 1 annotation per 27 lines for immutability annotations (IGJ type system)

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.

Each annotation that a programmer writes replaces a sentence or phrase of English descriptive text that would otherwise have been written in the Javadoc. So, use of annotations actually reduces the overall size of the documentation, at the same time as making it machine-processable and less ambiguous.

31.3.3 Will using the Checker Framework slow down my program? Will it slow down the compiler?

Using the Checker Framework has no impact on the execution of your program: the compiler emits the identical bytecodes as the Java 8 compiler and so there is no run-time effect. Because there is no run-time representation of type qualifiers, there is no way to use reflection to query the qualifier on a given object, though you can use reflection to examine a class/method/field declaration.

Using the Checker Framework does increase compilation time. In theory it should only add a few percent overhead, but our current implementation can double the compilation time — or more, if you run many pluggable type-checkers at once. This is especially true if you run pluggable type-checking on every file (as we recommend) instead of just on the ones that have recently changed. Nonetheless, compilation with pluggable type-checking still feels like compilation, and you can do it as part of your normal development process.

31.3.4 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 org.checkerframework.checker.nullness.NullnessChecker ...`. See Section 2.2.3 for a way to avoid the need for the `-processor` command-line option.

31.4 How to handle warnings and errors

31.4.1 What should I do if a checker issues a warning about my code?

For a discussion of this issue, see Section 2.4.6.

31.4.2 What does a certain Checker Framework warning message mean?

Search through this manual for the text of the warning message. Oftentimes the manual explains it. If not, ask on the mailing list.

31.4.3 Can a pluggable type-checker guarantee that my code is correct?

Each checker looks for certain errors. You can use multiple checkers to detect more errors in your code, but you will never have a guarantee that your code is completely bug-free.

If the type-checker issues no warning, then you have a guarantee that your code is free of some particular error. There are some limitations to the guarantee.

Most importantly, if you run a pluggable checker on only part of a program, then you only get a guarantee that those parts of the program are error-free. For example, suppose you have type-checked a framework that clients are intended to extend. 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.

Section 2.3 states other limitations to a checker's guarantee, such as regarding concurrency. Java's type system is also unsound in certain situations, such as for arrays and casts (however, the Checker Framework is sound for arrays and casts). Java uses dynamic checks is some places it is unsound, so that errors are thrown at run time. The pluggable type-checkers do not currently have built-in dynamic checkers to check for the places they are unsound. Writing dynamic checkers would be an interesting and valuable project.

Other types of dynamism in a Java application do not jeopardize the guarantee, because the type-checker is conservative. For example, at a method call, dynamic dispatch chooses some implementation of the method, but it is impossible to know at compile time which one it will be. The type-checker gives a guarantee no matter what implementation of the method is invoked.

Even if a pluggable checker cannot give an ironclad guarantee of correctness, it is still useful. It can find errors, exclude certain types of possible problems (e.g., restricting the possible class of problems), improve documentation, and increase confidence in your software.

31.4.4 What guarantee does the Checker Framework give for concurrent code?

The Lock Checker (see Chapter 6) offers a way to detect and prevent certain concurrency errors.

By default, the Checker Framework assumes that the code that it is checking is sequential: that is, there are no concurrent accesses from another thread. This means that the Checker Framework is unsound for concurrent code, in the sense that it may fail to issue a warning about errors that occur only when the code is running in a concurrent setting. For example, the Nullness Checker issues no warning for this code:

```
if (myobject.myfield != null) {
  myobject.myfield.toString();
}
```

This code is safe when run on its own. However, in the presence of multithreading, the call to `toString` may fail because another thread may set `myobject.myfield` to `null` after the nullness check in the `if` condition, but before the `if` body is executed.

If you supply the `-AconcurrentSemantics` command-line option, then the Checker Framework assumes that any field can be changed at any time. This limits the amount of flow-sensitive type qualifier refinement (Section 25.4) that the Checker Framework can do.

31.4.5 How do I make compilation succeed even if a checker issues errors?

Section 2.2 describes the `-Awarns` command-line option that turns checker errors into warnings, so type-checking errors will not cause `javac` to exit with a failure status.

31.4.6 Why does the checker always say there are 100 errors or warnings?

By default, javac only reports the first 100 errors or warnings. Furthermore, once javac encounters an error, it doesn't try compiling any more files (but does complete compilation of all the ones that it has started so far).

To see more than 100 errors or warnings, use the javac options `-Xmaxerrs` and `-Xmaxwarns`. To convert Checker Framework errors into warnings so that javac will process all your source files, use the option `-Awarns`. See Section 2.2 for more details.

31.4.7 Why does the Checker Framework report an error regarding a type I have not written in my program?

Sometimes, a Checker Framework warning message will mention a type you have not written in your program. This is typically because a default has been applied where you did not write a type; see Section 25.3.1. In other cases, this is because flow-sensitive type refinement has given an expression a more specific type than you wrote or than was defaulted; see Section 25.4.

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

Some properties are not checked statically (see Chapter 26 for reasons that code might not be statically checked). In such cases, it would be desirable to check the property dynamically, at run time. Currently, the Checker Framework has no support for adding code to perform run-time checking.

Adding such support would be an interesting and valuable project. An example would be an option that causes the Checker Framework to automatically insert a run-time check anywhere that static checking is suppressed. If you are able to add run-time verification functionality, we would gladly welcome it as a contribution to the Checker Framework.

Some checkers have library methods that you can explicitly insert in your source code. Examples include the Nullness Checker's `NullnessUtils.castNonNull` method (see Section 3.4.1) and the Regex Checker's `RegexUtil` class (see Section 9.2.4). But, it would be better to have more general support that does not require the user to explicitly insert method calls.

31.5 Syntax of type annotations

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

31.5.1 What is a "receiver"?

The *receiver* of a method is the `this` formal parameter, sometimes also called the "current object". Within the method declaration, `this` is used to refer to the receiver formal parameter. At a method call, the receiver actual argument is written before the method name.

The method `compareTo` takes *two* formal parameters. At a call site like `x.compareTo(y)`, the two arguments are `x` and `y`. It is desirable to be able to annotate the types of both of the formal parameters, and doing so is supported by both Java's type annotations syntax and by the Checker Framework.

A type annotation on the receiver is treated exactly like a type annotation on any other formal parameter. At each call site, the type of the argument must be a consistent with (a subtype of or equal to) the declaration of the corresponding formal parameter. If not, the type-checker issues a warning.

Here is an example. Suppose that `@A Object` is a supertype of `@B Object` in the following declaration:

```
class MyClass {
  void requiresA(@A MyClass this) { ... }
  void requiresB(@B MyClass this) { ... }
}
```

Then the behavior of four different invocations is as follows:

```
@A MyClass myA = ...;
@B MyClass myB = ...;

myA.requiresA()    // OK
myA.requiresB()    // compile-time error
myB.requiresA()    // OK
myB.requiresB()    // OK
```

The invocation `myA.requiresB()` does not type-check because the actual argument's type is not a subtype of the formal parameter's type.

A top-level constructor does not have a receiver. An inner class constructor does have a receiver, whose type is the same as the containing outer class. The receiver is distinct from the object being constructed. In a method of a top-level class, the receiver is named `this`. In a constructor of an inner class, the receiver is named `Outer.this` and the result is named `this`.

31.5.2 What is the meaning of an annotation after a type, such as `@NonNull Object @Nullable`?

In a type such as `@NonNull Object @Nullable []`, it may appear that the `@Nullable` annotation is written *after* the type `Object`. In fact, `@Nullable` modifies `[]`. See the next FAQ, about array annotations (Section 31.5.3).

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

You should parse this as: (**`@NonNull Object`**) (**`@Nullable []`**). Each annotation precedes the component of the type that it qualifies.

Thus, `@NonNull Object @Nullable []` is a possibly-null array of non-null objects. Note that the first token in the type, "`@NonNull`", applies to the element type `Object`, not to the array type as a whole. The annotation `@Nullable` applies to the array (`[]`).

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

Some older tools interpret a declaration like `@NonEmpty String[] var` as "non-empty array of strings". This is in conflict with the Java type annotations specification, which defines it as meaning "array of non-empty strings". If you use one of these older tools, you will find this incompatibility confusing. You will have to live with it until the older tool is updated to conform to the Java specification, or until you transition to a newer tool that conforms to the Java specification.

31.5.4 What is the meaning of a type qualifier at a class declaration?

Writing an annotation on a class declaration makes that annotation implicit for all uses of the class (see Section 25.3). If you write `class @MyQual MyClass { ... }`, then every unannotated use of `MyClass` is `@MyQual MyClass`. A user is permitted to strengthen the type by writing a more restrictive annotation on a use of MyClass, such as `@MyMoreRestrictiveQual MyClass`.

31.5.5 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.) If you do not know what "receiver" means, see the next question.

Another example of a type-and-declaration annotation that represents poor design is JCIP's `@GuardedBy` annotation [GPB+06]. As discussed in Section 6.3.1, it means two different things when applied to a field or a method. To reduce confusion and increase expressiveness, the Lock Checker (see Chapter 6) uses the `@Holding` annotation for one of these meanings, rather than overloading `@GuardedBy` with two distinct meanings.

31.6 Semantics of type annotations

31.6.1 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 elements. Then, you can suppress the warning:

```
@SuppressWarnings("nullness:generic.argument") // any list passed to this
method will support null elements
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.

31.6.2 How can I handle typestate, or phases of my program with different data properties?

Sometimes, your program works in phases that have different behavior. For example, you might have a field that starts out null and becomes non-null at some point during execution, such as after a method is called. You can express this property as follows:

1. Annotate the field type as `@MonotonicNonNull`.
2. Annotate the method that sets the field as `@EnsuresNonNull("myFieldName")`. (If method `m1` calls method `m2`, which actually sets the field, then you would probably write this annotation on both `m1` and `m2`.)
3. Annotate any method that depends on the field being non-null as `@RequiresNonNull("myFieldName")`. The type-checker will verify that such a method is only called when the field isn't null — that is, the method is only called after the setting method.

You can also use a typestate checker (see Chapter 23.1, page 111), but they have not been as extensively tested.

31.6.3 Why are explicit and implicit bounds defaulted differently?

The following two bits of code have the same semantics under Java, but are treated differently by the Checker Framework's CLIMB-to-top defaulting rules (Section 25.3.2):

```
class MyClass<T> { ... }
class MyClass<T extends Object> { ... }
```

The difference is the annotation on the upper bound of the type argument `T`. They are treated in the following.

```
class MyClass<T>  ==  class MyClass<T extends @TOPTYPEANNO Object> { ... }
class MyClass<T extends Object>  ==  class MyClass<T extends @DEFAULTANNO Object>
```

`@TOPTYPEANNO` is the top annotation in the type qualifier hierarchy. For example, for the nullness type system, the top type annotation is `@Nullable`; as shown in Figure 3.1. `@DEFAULTANNO` is the default annotation for the type system. For example, for the nullness type system, the default type annotation is `@NonNull`.

In some type systems, the top qualifier and the default are the same. For such type systems, the two code snippets shown above are treated the same. An example is the regular expression type system; see Figure 9.1.

The CLIMB-to-top rule reduces the code edits required to annotate an existing program, and it treats types written in the program consistently.

When a user writes no upper bound, as in `class C<T> { ... }`, then Java permits the class to be instantiated with any type parameter. The Checker Framework behaves exactly the same, no matter what the default is for a particular type system – and no matter whether the user has changed the default locally.

When a user writes an upper bound, as in `class C<T extends OtherClass> { ... }`, then the Checker Framework treats this occurrence of `OtherClass` exactly like any other occurrence, and applies the usual defaulting rules. Use of `Object` is treated consistently with all other types in this location and all other occurrences of `Object` in the program.

It is uncommon for a user to write `Object` as an upper bound with no type qualifier: `class C<T extends Object> { ... }`. It is better style to write no upper bound or to write an explicit type annotation on `Object`.

31.7 Creating a new checker

31.7.1 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 29 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_USE, ElementType.TYPE_PARAMETER})
public @interface Encrypted { }
```

Section 22.2 explains this checker and tells you how to run it.

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

A type system implementer can declaratively specify the type qualifier hierarchy (Section 29.3.2) and the type introduction rules (Section 29.4.1). However, the Checker Framework uses a procedural syntax for specifying type-checking rules (Section 29.6). 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.

31.8 Relationship to other tools

31.8.1 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 32.5. 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].

31.8.2 How does the Checker Framework compare with Eclipse's null analysis?

Eclipse comes with a null analysis that can detect potential null pointer errors in your code. Eclipse's built-in analysis differs from the Checker Framework in several respects.

The Checker Framework's Nullness Checker (see Chapter 3, page 24) is more precise: it does a deeper semantic analysis, so it issues fewer false positives than Eclipse. For example, the Nullness Checker handles initialization and map key checking, it supports method pre- and post-conditions, and it includes a powerful dataflow analysis.

Eclipse assumes that all code is multi-threaded, which cripples its local type inference. By contrast, the Checker Framework allows the user to specify whether code will be run concurrently or not via the `-AconcurrentSemantics` command-line option (see Section 31.4.4).

The Checker Framework is easier to run in integration scripts or in environments where not all developers are using Eclipse.

Eclipse handles only nullness properties and is not extensible, whereas the Checker Framework comes with over 20 type-checkers (for a list, see Chapter 1, page 12) and is extensible to more properties.

There are also some benefits to Eclipse's Null Analysis. It is faster than the Checker Framework, in part because it is less featureful. It is built into Eclipse, so you do not have to download and install a separate Eclipse plugin as you do for the Checker Framework (see Section 30.6.2). Its IDE integration is tighter and slicker.

(If you know of other differences, please let us know at checker-framework-dev@googlegroups.com so we can update the manual.)

31.8.3 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. 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. Pluggable type-checking annotations are more concise and easier to understand.

JML is not as practical as pluggable type-checking. 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.

31.8.4 Is the Checker Framework an official part of Java?

The Checker Framework is not an official part of Java. The Checker Framework relies on type annotations, which are part of Java 8. See the Type Annotations (JSR 308) FAQ for more details.

31.8.5 What is the relationship between the Checker Framework and JSR 305?

JSR 305 aimed to define official Java names for some annotations, such as `@NonNull` and `@Nullable`. However, it did not aim to precisely define the semantics of those annotations nor to provide a reference implementation of an annotation processor that validated their use.

By contrast, the Checker Framework precisely defines the meaning of a set of annotations and provides powerful type-checkers that validate them. However, the Checker Framework is not an official part of the Java language; it chooses one set of names, but another tool might choose other names.

JSR 305 has been abandoned; there has been no activity by its expert group since 2009. In the future, the Java Community Process might standardize the names and meanings of specific annotations, after there is more experience with their use in practice.

The Checker Framework defines annotations `@NonNull` and `@Nullable` that are compatible with annotations defined by JSR 305, FindBugs, IntelliJ, and other tools; see Section 3.7.

31.8.6 What is the relationship between the Checker Framework and JSR 308?

JSR 308, also known as the Type Annotations specification, dictates the syntax of type annotations in Java SE 8: how they are expressed in the Java language.

JSR 308 does not define any type annotations such as `@NonNull`, and it does not specify the semantics of any annotations. Those tasks are left to third-party tools. The Checker Framework is one such tool.

The Checker Framework makes use of Java SE 8's type annotation syntax, but the Checker Framework can be used with previous versions of the Java language via the annotations-in-comments feature (Section 27.2.1).

Chapter 32

Troubleshooting and getting help

The manual might already answer your question, so first please look for your answer in the manual, including this chapter and the FAQ (Chapter 31). 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, please see Section 32.2. If you want to help out, you can give feedback (including on the documentation), choose a bug and fix it, or select a project from the ideas list at https://github.com/typetools/checker-framework/wiki/Ideas.

32.1 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.

32.1.1 Unable to run the checker, or checker crashes

If you are unable to run the checker, or if the checker or the compiler terminates with an error, then the problem may be a problem with your environment. (If the checker or the compiler crashes, that is a bug in the Checker Framework; please report it. See Section 32.2.) This section describes some possible problems and solutions.

- 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 1.3).
- If you get an error such as

  ```
  package org.checkerframework.checker.nullness.qual does not exist
  ```

  despite no apparent use of import org.checkerframework.checker.nullness.qual.*; in the source code, then perhaps jsr308_imports is set as a Java system property, a shell environment variable, or a command-line option. You should solve this by unsetting the variable/option, which it is deprecated.
  If the error is

  ```
  package org.checkerframework.checker.nullness.qual does not exist
  ```

  (note the extra apostrophe!), then you have probably misused quoting when supplying the (deprecated) jsr308_imports environment variable.
- If you get an error like one of the following,

  ```
  ...\build.xml:59: Error running ${env.CHECKERFRAMEWORK}\checker\bin\javac.bat compiler
  ```

```
.../bin/javac: Command not found
```

then the problem may be that you have not set the `CHECKERFRAMEWORK` environment variable, as described in Section 30.1. 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 likely **not** using the Checker Framework compiler. Use either `$CHECKERFRAMEWORK/checker/bin/javac` one of the alternatives described in Section 1.3.

- If you get the error

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

If you get an error such as

```
java.lang.NoClassDefFoundError: java/util/Objects
```

then you are trying to run the compiler using a JDK 6 or earlier JVM. Install and use a Java 7 or 8 JDK, at least for running the Checker Framework.
then an annotation is not present at run time that was present at compile time. For example, maybe when you compiled the code, the `@Nullable` annotation was available, but it was not available at run time. You can use JDK 8 at run time, or compile with a Java 6 or 7 compiler that will ignore the annotations in comments.

- A "class file for ... not found" error, especially for an inner class in the JDK, is probably due to a JDK version mismatch.
In general, Java issues a "class file for ... not found" error when your classpath contains code that was compiled with some library, but your classpath does not contain that library itself.
For example, suppose that when you run the compiler, you are using JDK 8, but some library on your classpath was compiled against JDK 6 or 7, and the compiled library refers to a class that only appears in JDK 6 or 7. (If only one version of Java existed, or the Checker Framework didn't try to support multiple different versions of Java, this would not be a problem.)
Examples of classes that were in JDK 7 but were removed in JDK 8 include:

```
class file for java.util.TimeZone$DisplayNames not found
```

Examples of classes that were in JDK 6 but were removed in JDK 7 include:

```
class file for java.io.File$LazyInitialization not found
class file for java.util.Hashtable$EmptyIterator not found
java.lang.NoClassDefFoundError: java/util/Hashtable$EmptyEnumerator
```

Examples of classes that were not in JDK 7 but were introduced in JDK 8 include:

```
The type java.lang.Class$ReflectionData cannot be resolved
```

Examples of classes that were not in JDK 6 but were introduced in JDK 7 include:

```
class file for java.util.Vector$Itr not found
```

There are even classes that were introduced within a single JDK release. Classes that appear in JDK 7 release 71 but not in JDK 7 release 45 include:

```
class file for java.lang.Class$ReflectionData not found
```

You may be able to solve the problem by running

```
cd checker
ant jdk.jar bindist
```

to re-generate files `checker/jdk/jdk{7,8}.jar` and `checker/bin/jdk{7,8}.jar`.
That usually works, but if not, then you should recompile the Checker Framework from source rather than using the pre-compiled distribution.

- A `NoSuchFieldError` such as this:

  ```
  java.lang.NoSuchFieldError: NATIVE_HEADER_OUTPUT
  ```

  Field `NATIVE_HEADER_OUTPUT` was added in JDK 8. The error message suggests that you're not executing with the right bootclasspath: some classes were compiled with the JDK 8 version and expect the field, but you're executing the compiler on a JDK without the field.
  One possibility is that you are not running the Checker Framework compiler — use `javac -version` to check this, then use the right one. (Maybe the Checker Framework javac is at the end rather than the beginning of your path.)
  If you are using Ant, then one possibility is that the javac compiler is using the same JDK as Ant is using. You can correct this by being sure to use `fork="yes"` (see Section 30.2) and/or setting the `build.compiler` property to `extJavac`.
  If you are building from source, you might need to rebuild the Annotation File Utilities before recompiling or using the Checker Framework.
- If you get an error that contains lines like these:

  ```
  Caused by: java.util.zip.ZipException: error in opening zip file
      at java.util.zip.ZipFile.open(Native Method)
      at java.util.zip.ZipFile.<init>(ZipFile.java:131)
  ```

  then one possibility is that you have installed the Checker Framework in a directory that contains special characters that Java's ZipFile implementation cannot handle. For instance, if the directory name contains "+", then Java 1.6 throws a ZipException, and Java 1.7 throws a FileNotFoundException and prints out the directory name with "+" replaced by blanks.
- If you get an error

  ```
  error: scoping construct for static nested type cannot be annotated
  ```

  then you have probably written something like `@Nullable java.util.List`. The correct syntax is `java.util.@Nullable List`. But, it's usually better to add `import java.util.List` to your source file, so that you can just write `@Nullable List`. Likewise, you must write `Outer.@Nullable StaticNestedClass` rather than `@Nullable Outer.StaticNestedClass`.
  Java 8 requires that a type qualifier be written directly on the type that it qualifies, rather than on a scoping mechanism that assists in resolving the name. Examples of scoping mechanisms are package names and outer classes of static nested classes.
  The reason for the Java 8 syntax is to avoid syntactic irregularity. When writing a member nested class (also known as an inner class), it is possible to write annotations on both the outer and the inner class: `@A1 Outer.@A2 Inner`. Therefore, when writing a static nested class, the annotations should go on the same place: `Outer.@A3 StaticNested` (rather than `@ConfusingAnnotation Outer.Nested` where `@ConfusingAnnotation` applies to `Outer` if `Nested` is a member class and applies to `Nested` if `Nested` is a static class). It's not legal to write an annotation on the outer class of a static nested class, because neither annotations nor instantiations of the outer class affect the static nested class.
  Similar arguments apply when annotating `package.Outer.Nested`.

32.1.2 Unexpected type-checking results

This section describes possible problems that can lead the type-checker to give unexpected results.

- If the Checker Framework is unable to verify a property that you know is true, then it is helpful to formulate an argument about why the property is true. Recall that the Checker Framework does modular verification, one procedure at a time; it observes the specifications, but not the implementations, of other methods.
  If any aspects of your argument are not expressed as annotations, then you may need to write more annotations.
  If any aspects of your argument are not expressible as annotations, then you may need to extend the type-checker.

- 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 Checker Framework 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 a checker issues a warning about a property that it accepted (or that was checked) on a previous line, then probably there was a side-effecting method call in between that could invalidate the property. For example, in this code:

  ```
  if (currentOutgoing != null && !message.isCompleted()) {
      currentOutgoing.continueBuffering(message);
  }
  ```

  the Nullness Checker will issue a warning on the second line:

  ```
  warning: [dereference.of.nullable] dereference of possibly-null reference currentOutgoing
      currentOutgoing.continueBuffering(message);
      ^
  ```

  If `currentOutgoing` is a field rather than a local variable, and `isCompleted()` is not a pure method, then a null pointer dereference can occur at the given location, because `isCompleted()` might set the field `currentOutgoing` to `null`.
  If you want to communicate that isCompleted() does not set the field `currentOutgoing` to `null`, you can use `@Pure`, `@SideEffectFree`, or `@EnsuresNonNull` on the declaration of `isCompleted()`; see Sections 25.4.3 and 3.2.2.
- If a checker issues a type-checking error for a call that the library's documentation states is correct, then maybe that library method has not yet been annotated, so default annotations are being used.
  To solve the problem, add the missing annotations to the library (see Chapter 28). Depending on the checker, the annotations might be expressed in the form of stub files (which appear together with the checker's source code, such as in file `checker/src/org/checkerframework/checker/interning/jdk.astub` for the Interning Checker) or in the form of annotated libraries (which appear under `checker/jdk/`, such as at `checker/jdk/nullness/src/` for the Nullness Checker.
- 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.
  To solve the problem, add the missing annotations to the library, as described in the previous item.
  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 related to a bounded type parameter and a literal such as `null`, the problem may be missing defaulting. Here is an example:

  ```
  mypackage/MyClass.java:2044: warning: incompatible types in assignment.
        T retval = null;
                   ^
    found   : null
    required: T extends @MyQualifier Object
  ```

  A value that can be assigned to a variable of type `T extends @MyQualifier Object` only if that value is of the bottom type, since the bottom type is the only one that is a subtype of every subtype of `T extends @MyQualifier Object`. The value `null` satisfies this for the Java type system, and it must be made to satisfy it for the pluggable type system as well. The typical way to address this is to write the meta-annotation `@ImplicitFor(trees=Tree.Kind.NULL_LITERAL)` on the definition of the bottom type qualifier.

- An error such as

  ```
  MyFile.java:123: error: incompatible types in argument.
                          myModel.addElement("Scanning directories...");
                                             ^
    found   : String
    required: ? extends Object
  ```

  may stem from use of raw types. ("`String`" might be a different type and might have type annotations.) If your declaration was

  ```
  DefaultListModel myModel;
  ```

  then it should be

  ```
  DefaultListModel<String> myModel;
  ```

  Running the regular Java compiler with the `-Xlint:unchecked` command-line option will help you to find and fix problems such as raw types.
- The error

  ```
  error: annotation type not applicable to this kind of declaration
      ... List<@NonNull String> ...
  ```

  indicates that you are using a definition of `@NonNull` that is a declaration annotation, which cannot be used in that syntactic location. For example, many legacy annotations such as those listed in Figure 3.2 are declaration annotations. You can fix the problem by instead using a definition of `@NonNull` that is a type annotation, such as the Checker Framework's annotations; often this only requires changing an `import` statement. Alternately, if you wish to continue using the legacy annotations in declaration locations, see Section 27.2.5.
- This compile-time error

  ```
  unknown enum constant java.lang.annotation.ElementType.TYPE_USE
  ```

  indicates that you are compiling using a Java 6 or 7 JDK, but your code references an enum constant that is only defined in the Java 8 JDK. The problem might be that your code uses a library that references the enum constant. In particular, the type annotations shipped with the Checker Framework reference `ElementType.TYPE_USE`. You can use the Checker Framework, but still compile and run your code in a Java 6 or 7 JVM, by following the instructions in Section 27.2.
  If you ignore the error and run your code in a Java 6 or 7 JVM, then you will get a run-time error:

  ```
  java.lang.ArrayStoreException: sun.reflect.annotation.EnumConstantNotPresentExceptionProxy
  ```
- If Eclipse gives the warning

  ```
  The annotation @NonNull is disallowed for this location
  ```

  then you have the wrong version of the `org.eclipse.jdt.annotation` classes. Eclipse includes two incompatible versions of these annotations. You want the one with a name like `org.eclipse.jdt.annotation_2.0.0.....jar`, which you can find in the `plugins` subdirectory under the Eclipse installation directory. Add this .jar file to your build path.

32.1.3 Unable to build the checker, or to run programs

An error like this

```
Unsupported major.minor version 52.0
```

means that you have compiled some files into the Java 8 format (version 52.0), but you are trying to run them with Java 7 or earlier. Likewise, "Unsupported major.minor version 51.0" means that you have compiled some files into the Java 7 format (version 51.0), but you are trying to run them with Java 6 or earlier. Here are ways to solve the problem:

- Use a newer JVM (run `java -version` to determine the version you are using)
- Use the Checker Framework to type-check your code, then afterword produce a classfile that targets an earlier JVM by supplying arguments such as `javac -source 7 -target 7 ...`.

32.1.4 Classfile version warning

The following warning is innocuous and you can ignore it, or you can suppress it using the `-Xlint:-classfile` command-line argument to javac:

```
RuntimeVisibleTypeAnnotations attribute introduced in version 52.0 class files is ignored in version 51.0 c
```

This warning results when you compile a library using the Checker Framework compiler, then use a normal Java compiler to compile client code that uses the library. The Checker Framework compiler puts Java 8 type annotations even in Java 7 classfiles, for the benefit of modular typechecking. The Checker Framework compiler reads these annotations in Java 7, and other compilers ignore them.

32.2 How to report problems (bug reporting)

If you have a problem with any checker, or with the Checker Framework, please file a bug at `https://github.com/typetools/checker-framework/issues`. (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 -AprintErrorStack -AprintAllQualifiers` 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 (a screenshot is not as useful).
- 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. Please attach the files, rather than pasting their contents into the body of your bug report or email message.
- 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, and why — remember, a bug is a difference between desired and actual outcomes.
- Indicate what you have already done to try to understand the problem. Did you do any additional experiments? What parts of the manual did read, and what else did you search for in the manual? This information will prevent you being given redundant suggestions.

A particularly useful format for a test case is as a new file, or a diff to an existing file, for the existing Checker Framework test suite. For instance, for the Nullness Checker, see directory `checker-framework/checker/tests/nullness/`. But, please report your bug even if you do not report it in this format.

32.3 Building from source

The Checker Framework release (Section 1.3) contains everything that most users need, both to use the distributed checkers and to write your own checkers. This section describes how to compile its binaries from source. You will be using the latest development version of the Checker Framework, rather than an official release.

32.3.1 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://bitbucket.org/typetools/jsr308-langtools jsr308-langtools
git clone https://github.com/typetools/checker-framework.git checker-framework
git clone https://github.com/typetools/annotation-tools.git annotation-tools
```

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

32.3.2 Build the Type Annotations compiler

The Checker Framework compiler is built upon a compiler called the Type Annotations compiler. The Type Annotations compiler is a variant of the OpenJDK javac that supports annotations in comments. The Checker Framework compiler is a small wrapper around the Type Annotations compiler, which adds annotated JDKs and the Checker Framework jars to the classpath.

1. Set the `JAVA_HOME` environment variable to the location of your JDK 7 or 8 installation (not the JRE installation, and not JDK 6 or earlier). This needs to be an Oracle JDK. (The `JAVA_HOME` environment variable might already be set, because it is needed for Ant to work.)
   In the bash shell, the following command *sometimes* works (it might not because `java` might be the version in the JDK or in the JRE):
   ```
   export JAVA_HOME=${JAVA_HOME:-$(dirname $(dirname $(dirname $(readlink -f $(/usr/bin/which java)))))
   ```
2. Compile the Type Annotations tools:
   ```
   cd $JSR308/jsr308-langtools/make
   ant clean-and-build-all-tools
   ```
3. Add the `jsr308-langtools/dist/bin` directory to the front of your PATH environment variable. Example command:
   ```
   export PATH=$JSR308/jsr308-langtools/dist/bin:${PATH}
   ```
   You may wish to later put the Checker Framework `javac` even earlier on your path. The Checker Framework's `javac` ensures that the Checker Framework is on your classpath and bootclasspath, but is otherwise identical to the Type Annotations compiler.

32.3.3 Build the Annotation File Utilities

This is simply done by:

```
cd $JSR308/annotation-tools
ant
```

You do not need to add the Annotation File Utilities to the path, as the Checker Framework build finds it using relative paths.

32.3.4 Build the Checker Framework

1. Run `ant` to create `checker.jar`:
   ```
   cd $JSR308/checker-framework/checker
   ant
   ```
2. Add `tools.jar` and `checker.jar` to your classpath. (If you do not do this, you will have to supply the `-cp` option whenever you run `javac` and use a checker plugin.) Example command:
   ```
   export CLASSPATH=${CLASSPATH}:$JAVA_HOME/lib/tools.jar:$JSR308/checker-framework/checker/dist/checker.jar
   ```

3. Test that everything works:
   - Run `ant all-tests` in the `checker` directory:
     ```
     cd $JSR308/checker-framework/checker
     ant all-tests
     ```
   - Run the Nullness Checker examples (see Section 3.5, page 30).

32.3.5 Build the Checker Framework Manual (this document)

1. To build the manual you will need HEVEA (`http://hevea.inria.fr/`) installed.
2. Run `make` in the `checker/manual` directory to build both the PDF and HTML versions of the manual.

32.4 Publications

Here are two technical papers about the Checker Framework itself:

- "Practical pluggable types for Java" [PAC+08] (ISSTA 2008, `http://homes.cs.washington.edu/~mernst/pubs/pluggable-checkers-issta2008.pdf`) describes the design and implementation of the Checker Framework. The paper also describes case studies in which the Nullness, Interning, Javari, and IGJ Checkers found previously-unknown errors in real software. The case studies also yielded new insights about type systems.
- "Building and using pluggable type-checkers" [DDE+11] (ICSE 2011, `http://homes.cs.washington.edu/~mernst/pubs/pluggable-checkers-icse2011.pdf`) discusses further experience with the Checker Framework, increasing the number of lines of verified code to 3 million. The case studies are of the Fake Enum, Signature String, Interning, and Nullness Checkers. The paper also evaluates the ease of pluggable type-checking with the Checker Framework: type-checkers were easy to write, easy for novices to use, and effective in finding errors.

Here are some papers about type systems that were implemented and evaluated using the Checker Framework:

Nullness (Chapter 3) See the two papers about the Checker Framework, described above.

Rawness initialization (Section 3.8.6) "Inference of field initialization" (ICSE 2011, `http://homes.cs.washington.edu/~mernst/pubs/initialization-icse2011-abstract.html`) describes inference for the Rawness Initialization Checker.

Interning (Chapter 5) See the two papers about the Checker Framework, described above.

Fake enumerations (Chapter 7) See the ICSE 2011 paper about the Checker Framework, described above.

Regular expressions (Chapter 9) "A type system for regular expressions" [SDE12] (FTfJP 2012, `http://homes.cs.washington.edu/~mernst/pubs/regex-types-ftfjp2012-abstract.html`) describes the Regex Checker.

Format Strings (Chapter 10) "A type system for format strings" [WKSE14] (ISSTA 2014, `http://homes.cs.washington.edu/~mernst/pubs/format-string-issta2014-abstract.html`) describes the Format String Checker.

Signature strings (Chapter 13) See the ICSE 2011 paper about the Checker Framework, described above.

GUI Effects (Chapter 14) "JavaUI: Effects for controlling UI object access" [GDEG13] (ECOOP 2013, `http://homes.cs.washington.edu/~mernst/pubs/gui-thread-ecoop2013-abstract.html`) describes the GUI Effect Checker.

"Verification games: Making verification fun" (FTfJP 2012, `http://homes.cs.washington.edu/~mernst/pubs/verigames-ftfjp2012-abstract.html`) describes a general inference approach that, at the time, had only been implemented for the Nullness Checker (Section 3).

IGJ and OIGJ immutability (Chapter 19) "Object and reference immutability using Java generics" [ZPA+07] (ESEC/FSE 2007, `http://homes.cs.washington.edu/~mernst/pubs/immutability-generics-fse2007-abstract.html`) and "Ownership and immutability in generic Java" [ZPL+10] (OOPSLA 2010, `http://homes.cs.washington.edu/~mernst/pubs/ownership-immutability-oopsla2010-abstract.html`) describe the IGJ and OIGJ immutability type systems. For further case studies, also see the ISSTA 2008 paper about the Checker Framework, described above.

Javari immutability (Chapter 20) "Javari: Adding reference immutability to Java" [TE05] (OOPSLA 2005, http://homes.cs.washington.edu/~mernst/pubs/ref-immutability-oopsla2005-abstract.html) describes the Javari type system. For inference, see "Inference of reference immutability" [QTE08] (ECOOP 2008, http://homes.cs.washington.edu/~mernst/pubs/infer-refimmutability-ecoop2008-abstract.html) and "Parameter reference immutability: Formal definition, inference tool, and comparison" [AQKE09] (J.ASE 2009, http://homes.cs.washington.edu/~mernst/pubs/mutability-jase2009-abstract.html). For further case studies, also see the ISSTA 2008 paper about the Checker Framework, described above.

Thread locality (Section 23.3) "Loci: Simple thread-locality for Java" [WPM+09] (ECOOP 2009, http://janvitek.github.io/pubs/ecoop09.pdf)

Generic Universe Types (Section 23.5) "Tunable static inference for Generic Universe Types" (ECOOP 2011, http://homes.cs.washington.edu/~mernst/pubs/tunable-typeinf-ecoop2011-abstract.html) describes inference for the Generic Universe Types type system.
Another implementation of Universe Types and ownership types is described in "Inference and checking of object ownership" [HDME12] (ECOOP 2012, http://homes.cs.washington.edu/~mernst/pubs/infer-ownership-ecoop2012-a
html).

Approximate data (Section 23.6) "EnerJ: Approximate Data Types for Safe and General Low-Power Computation" [SDF+11] (PLDI 2011, http://adriansampson.net/media/papers/enerj-pldi2011.pdf)

Information flow and tainting (Section 23.8) "Collaborative Verification of Information Flow for a High-Assurance App Store" [EJM+14] (CCS 2014, http://homes.cs.washington.edu/~mernst/pubs/infoflow-ccs2014.pdf) describes the SPARTA information flow type system.

ReIm immutability "ReIm & ReImInfer: Checking and inference of reference immutability and method purity" [HMDE12] (OOPSLA 2012, http://homes.cs.washington.edu/~mernst/pubs/infer-refimmutability-oopsla2012-abstract.html) describes the ReIm immutability type system.

In addition to these papers that discuss use the Checker Framework directly, other academic papers use the Checker Framework in their implementation or evaluation. Most educational use of the Checker Framework is never published, and most commercial use of the Checker Framework is never discussed publicly.

(If you know of a paper or other use that is not listed here, please inform the Checker Framework developers so we can add it.)

32.5 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 alarms, also known as false positive warnings; see Section 26.
- 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 (that is, null pointer dereferences). None of the other tools found any errors.

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

32.6 Credits, changelog, and license

The key developers of the Checker Framework are Mahmood Ali, Telmo Correa, Werner M. Dietl, Michael D. Ernst, and Matthew M. Papi. Many other developers have also contributed, for example by writing the checkers that are distributed with the Checker Framework. Many, many users to list have provided valuable feedback, for which we are grateful.

Differences from previous versions of the checkers and framework can be found in the `changelog.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.txt`.

Two different licenses apply to different parts of the Checker Framework.

- The Checker Framework itself is licensed under the GNU General Public License (GPL), version 2. The GPL is the same license that OpenJDK is licensed under. That means that type-checking your code using the Checker Framework is no more dangerous (from an intellectual property point of view) than compiling your code using javac.
- The more permissive MIT License applies to code that you might want to include in your own program, such as the annotations.

For details, see file `LICENSE.txt`.

Bibliography

[AQKE09] Shay Artzi, Jaime Quinonez, Adam Kieżun, and Michael D. Ernst. Parameter reference immutability: Formal definition, inference tool, and comparison. *Automated Software Engineering*, 16(1):145–192, March 2009.

[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.

[DDE+11] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kıvanç Muşlu, and Todd Schiller. Building and using pluggable type-checkers. In *ICSE'11, Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690, Waikiki, Hawaii, USA, May 25–27, 2011.

[EJM+14] Michael D. Ernst, René Just, Suzanne Millstein, Werner M. Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu. Collaborative verification of information flow for a high-assurance app store. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, Scottsdale, AZ, USA, November 4–6, 2014.

[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.

[GDEG13] Colin S. Gordon, Werner Dietl, Michael D. Ernst, and Dan Grossman. JavaUI: Effects for controlling UI object access. In *ECOOP 2013 — Object-Oriented Programming, 27th European Conference*, pages 179–204, Montpellier, France, July 3–5, 2013.

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

[HDME12] Wei Huang, Werner Dietl, Ana Milanova, and Michael D. Ernst. Inference and checking of object ownership. In *ECOOP 2012 — Object-Oriented Programming, 26th European Conference*, pages 181–206, Beijing, China, June 14–16, 2012.

[HMDE12] Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. ReIm & ReImInfer: Checking and inference of reference immutability and method purity. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2012)*, pages 879–896, Tucson, AZ, USA, October 23–25, 2012.

[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.

[SDE12] Eric Spishak, Werner Dietl, and Michael D. Ernst. A type system for regular expressions. In *FTfJP 2012: 14th Workshop on Formal Techniques for Java-like Programs*, pages 20–26, Beijing, China, June 12, 2012.

[SDF+11] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI 2011, Proceedings of the ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation*, pages 164–174, San Jose, CA, USA, June 6–8, 2011.

[SM11] Alexander J. Summers and Peter Müller. Freedom before commitment: A lightweight type system for object initialisation. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2011)*, pages 1013–1032, Portland, OR, USA, October 25–27, 2011.

[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.

[VPEJ14] Mohsen Vakilian, Amarin Phaosawasdi, Michael D. Ernst, and Ralph E. Johnson. Cascade: A universal type qualifier inference tool. Technical report, University of Illinois at Urbana-Champaign, Urbana, IL, USA, September 2014.

[WKSE14] Konstantin Weitz, Gene Kim, Siwakorn Srisakaokul, and Michael D. Ernst. A type system for format strings. In *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 127–137, San Jose, CA, USA, July 23–25, 2014.

[WPM+09] Tobias Wrigstad, Filip Pizlo, Fadi Meawad, Lei Zhao, and Jan Vitek. Loci: Simple thread-locality for Java. In *ECOOP 2009 — Object-Oriented Programming, 23rd European Conference*, pages 445–469, Genova, Italy, July 8–10, 2009.

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

[ZPL+10] Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. Ownership and immutability in generic Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2010)*, pages 598–617, Revo, NV, USA, October 19–21, 2010.