The Checker Framework Manual: Custom pluggable types for Java

https://checkerframework.org/

Version 2.5.1 (1 May 2018)

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 . . . 13

2 Using a checker **15**
2.1 Writing annotations . . . 15
2.2 Running a checker . . . 16
2.2.1 Using annotated libraries . . . 16
2.2.2 Distributing your annotated project . . . 17
2.2.3 Summary of command-line options . . . 17
2.2.4 Checker auto-discovery . . . 19
2.2.5 Shorthand for built-in checkers . . . 19
2.3 What the checker guarantees . . . 19
2.4 Tips about writing annotations . . . 20
2.4.1 Write annotations before you run a checker . . . 20
2.4.2 How to get started annotating legacy code . . . 20
2.4.3 Annotations indicate normal behavior . . . 21
2.4.4 Subclasses must respect superclass annotations . . . 22
2.4.5 Annotations on constructor invocations . . . 23
2.4.6 What to do if a checker issues a warning about your code . . . 23

3 Nullness Checker **25**
3.1 What the Nullness Checker checks . . . 25
3.2 Nullness annotations . . . 26
3.2.1 Nullness qualifiers . . . 26
3.2.2 Nullness method annotations . . . 27
3.2.3 Initialization qualifiers . . . 27
3.2.4 Map key qualifiers . . . 27
3.3 Writing nullness annotations . . . 28
3.3.1 Implicit qualifiers . . . 28
3.3.2 Default annotation . . . 28
3.3.3 Conditional nullness . . . 28
3.3.4 Nullness and arrays . . . 29
3.3.5 Run-time checks for nullness . . . 29
3.3.6 Additional details . . . 29
3.3.7 Inference of `@NonNull` and `@Nullable` annotations . . . 30
3.4 Suppressing nullness warnings . . . 30
3.4.1 Suppressing warnings with assertions and method calls . . . 30

3.5 Examples . . . 31
3.5.1 Tiny examples . . . 31
3.5.2 Example annotated source code . . . 31
3.6 Tips for getting started . . . 31
3.7 Nullness_Lite: An Unsound Option of the Nullness Checker for fewer false positives . . . 32
3.7.1 Introduction . . . 32
3.7.2 Who wants to use Nullness_Lite option? . . . 32
3.7.3 How to use the Nullness_Lite? . . . 33
3.7.4 More information . . . 33
3.8 Other tools for nullness checking . . . 33
3.8.1 Which tool is right for you? . . . 33
3.8.2 Incompatibility note about FindBugs `@Nullable` . . . 34
3.8.3 Relationship to `Optional<T>` . . . 35
3.9 Initialization Checker . . . 35
3.9.1 Initialization qualifiers . . . 38
3.9.2 How an object becomes initialized . . . 38
3.9.3 Partial initialization . . . 39
3.9.4 Method calls from the constructor . . . 40
3.9.5 Initialization of circular data structures . . . 41
3.9.6 How to handle warnings . . . 43
3.9.7 More details about initialization checking . . . 44
3.9.8 Rawness Initialization Checker . . . 44

4 Map Key Checker **50**
4.1 Map key annotations . . . 50
4.2 Examples . . . 51
4.3 Inference of @KeyFor annotations . . . 51

5 Optional Checker for possibly-present data **53**
5.1 How to run the Optional Checker . . . 53
5.2 Optional annotations . . . 53

6 Interning Checker **55**
6.1 Interning annotations . . . 56
6.2 Annotating your code with `@Interned` . . . 56
6.2.1 Implicit qualifiers . . . 56
6.2.2 InternedDistinct: values not equals() to any other value . . . 56
6.3 What the Interning Checker checks . . . 57
6.3.1 Limitations of the Interning Checker . . . 58
6.4 Examples . . . 58
6.5 Other interning annotations . . . 58

7 Lock Checker **59**
7.1 What the Lock Checker guarantees . . . 59
7.2 Lock annotations . . . 59
7.2.1 Type qualifiers . . . 59
7.2.2 Declaration annotations . . . 61
7.3 Type-checking rules . . . 61
7.3.1 Polymorphic qualifiers . . . 61
7.3.2 Dereferences . . . 62
7.3.3 Primitive types, boxed primitive types, and Strings . . . 62
7.3.4 Overriding . . . 62

7.3.5 Side effects 62
7.4 Examples 62
7.4.1 Examples of @GuardedBy 63
7.4.2 @GuardedBy({“a”, “b”}) is not a subtype of @GuardedBy({“a”}) 64
7.4.3 Examples of @Holding 64
7.4.4 Examples of @EnsuresLockHeld and @EnsuresLockHeldIf 65
7.4.5 Example of @LockingFree, @ReleasesNoLocks, and @MayReleaseLocks 65
7.4.6 Polymorphism and method formal parameters with unknown guards 66
7.5 More locking details 67
7.5.1 Two types of locking: monitor locks and explicit locks 67
7.5.2 Held locks and held expressions; aliasing 67
7.5.3 Run-time checks for locking 67
7.5.4 Discussion of default qualifier 68
7.5.5 Discussion of `@Holding` 68
7.6 Other lock annotations 69
7.6.1 Relationship to annotations in *Java Concurrency in Practice* 69
7.7 Possible extensions 69

8 Index Checker for sequence bounds (arrays and strings) 70
8.1 Index Checker structure and annotations 70
8.2 Lower bounds 71
8.3 Upper bounds 72
8.4 Sequence minimum lengths 73
8.5 Sequences of the same length 74
8.6 Binary search indices 75
8.7 Substring indices 76
8.7.1 The need for the `@SubstringIndexFor` annotation 76
8.8 Inequalities 77
8.9 Annotating fixed-size data structures 77

9 Fake Enum Checker for fake enumerations 79
9.1 Fake enum annotations 79
9.2 What the Fenum Checker checks 80
9.3 Running the Fenum Checker 80
9.4 Suppressing warnings 81
9.5 Example 81
9.6 The fake enumeration pattern 82
9.7 References 82

10 Tainting Checker 83
10.1 Tainting annotations 83
10.2 Tips on writing `@Untainted` annotations 83
10.3 `@Tainted` and `@Untainted` can be used for many purposes 84

11 Regex Checker for regular expression syntax 85
11.1 Regex annotations 85
11.2 Annotating your code with `@Regex` 85
11.2.1 Implicit qualifiers 85
11.2.2 Capturing groups 86
11.2.3 Concatenation of partial regular expressions 86
11.2.4 Testing whether a string is a regular expression 87
11.2.5 Suppressing warnings 87

12 Format String Checker 88
12.1 Formatting terminology 88
12.2 Format String Checker annotations 88
12.2.1 Conversion Categories 89
12.2.2 Subtyping rules for `@Format` 91
12.3 What the Format String Checker checks 91
12.3.1 Possible false alarms 92
12.3.2 Possible missed alarms 92
12.4 Implicit qualifiers 93
12.5 @FormatMethod 93
12.6 Testing whether a format string is valid 93

13 Internationalization Format String Checker (I18n Format String Checker) 95
13.1 Internationalization Format String Checker annotations 95
13.2 Conversion categories 96
13.3 Subtyping rules for `@I18nFormat` 96
13.4 What the Internationalization Format String Checker checks 97
13.5 Resource files 98
13.6 Running the Internationalization Format Checker 99
13.7 Testing whether a string has an i18n format type 99
13.8 Examples of using the Internationalization Format Checker 99

14 Property File Checker 101
14.1 General Property File Checker 101
14.2 Internationalization Checker (I18n Checker) 102
14.2.1 Internationalization annotations 102
14.2.2 Running the Internationalization Checker 102
14.3 Compiler Message Key Checker 102

15 Signature String Checker for string representations of types 104
15.1 Signature annotations 104
15.2 What the Signature Checker checks 106

16 GUI Effect Checker 107
16.1 GUI effect annotations 108
16.2 What the GUI Effect Checker checks 108
16.3 Running the GUI Effect Checker 108
16.4 Annotation defaults 108
16.5 Polymorphic effects 109
16.5.1 Defining an effect-polymorphic type 109
16.5.2 Using an effect-polymorphic type 109
16.5.3 Subclassing a specific instantiation of an effect-polymorphic type 109
16.5.4 Subtyping with polymorphic effects 110
16.6 References 111

17 Units Checker 112
17.1 Units annotations 112
17.2 Extending the Units Checker 113
17.3 What the Units Checker checks 114
17.4 Running the Units Checker 114
17.5 Suppressing warnings 114
17.6 References 115

18 Signedness Checker **116**
18.1 Annotations . . . 116
18.1.1 Default qualifiers . . . 117
18.2 Prohibited operations . . . 117
18.3 Rationale . . . 117
18.4 Utility routines for manipulating unsigned values . . . 118

19 Constant Value Checker **119**
19.1 Annotations . . . 119
19.1.1 Type Annotations . . . 119
19.1.2 Compile-time execution of expressions . . . 121
19.1.3 `@StaticallyExecutable` methods and the classpath . . . 121
19.2 Warnings . . . 121
19.3 Unsoundly ignoring overflow . . . 122

20 Aliasing Checker **123**
20.1 Aliasing annotations . . . 123
20.2 Leaking contexts . . . 124
20.3 Restrictions on where `@Unique` may be written . . . 125
20.4 Aliasing type refinement . . . 125

21 Reflection resolution **127**
21.1 MethodVal and ClassVal Checkers . . . 127
21.1.1 ClassVal Checker . . . 127
21.1.2 MethodVal Checker . . . 128
21.1.3 MethodVal and ClassVal inference . . . 129
21.2 Reflection resolution example . . . 130

22 Subtyping Checker **132**
22.1 Using the Subtyping Checker . . . 132
22.2 Subtyping Checker example . . . 133
22.3 Type aliases and typedefs . . . 135

23 Third-party checkers **137**
23.1 Typestate checkers . . . 137
23.1.1 Comparison to flow-sensitive type refinement . . . 137
23.2 Units and dimensions checker . . . 138
23.3 Thread locality checker . . . 138
23.4 Safety-Critical Java checker . . . 138
23.5 Generic Universe Types checker . . . 138
23.6 EnerJ checker . . . 138
23.7 CheckLT taint checker . . . 138
23.8 SPARTA information flow type-checker for Android . . . 138
23.9 Immutability checkers: IGJ, OIGJ, and Javari . . . 139
23.10 Read Checker for CERT FIO08-J . . . 139
23.11 SQL checker that supports multiple dialects . . . 139
23.12 Glacier: Class immutability . . . 139

24 Generics and polymorphism **140**
24.1 Generics (parametric polymorphism or type polymorphism) . . . 140
24.1.1 Raw types . . . 140
24.1.2 Restricting instantiation of a generic class . . . 140
24.1.3 Type annotations on a use of a generic type variable . . . 142
24.1.4 Annotations on wildcards . . . 142
24.1.5 Examples of qualifiers on a type parameter . . . 143
24.1.6 Covariant type parameters . . . 143
24.1.7 Method type argument inference and type qualifiers . . . 144
24.1.8 The Bottom type . . . 144
24.2 Qualifier polymorphism . . . 144
24.2.1 Examples of using polymorphic qualifiers . . . 145
24.2.2 Relationship to subtyping and generics . . . 145
24.2.3 The `@PolyAll` qualifier applies to every type system . . . 146
24.2.4 Using multiple polymorphic qualifiers in a method signature . . . 146
24.2.5 Using a single polymorphic qualifier in a method signature . . . 147

25 Advanced type system features **149**
25.1 Invariant array types . . . 149
25.2 Context-sensitive type inference for array constructors . . . 149
25.3 The effective qualifier on a type (defaults and inference) . . . 150
25.3.1 Default qualifier for unannotated types . . . 151
25.3.2 Defaulting rules and CLIMB-to-top . . . 152
25.3.3 Inherited defaults . . . 153
25.3.4 Inherited wildcard annotations . . . 153
25.3.5 Default qualifiers for `.class` files (conservative library defaults) . . . 154
25.4 Automatic type refinement (flow-sensitive type qualifier inference) . . . 154
25.4.1 Type refinement examples . . . 155
25.4.2 Types that are not refined . . . 156
25.4.3 Run-time tests and type refinement . . . 156
25.4.4 Fields and flow-sensitive analysis . . . 157
25.4.5 Side effects, determinism, purity, and flow-sensitive analysis . . . 158
25.4.6 Assertions . . . 160
25.5 Writing Java expressions as annotation arguments . . . 160
25.6 Field invariants . . . 161
25.7 Unused fields . . . 162
25.7.1 `@Unused` annotation . . . 162

26 Suppressing warnings **164**
26.1 `@SuppressWarnings` annotation . . . 164
26.1.1 `@SuppressWarnings` syntax . . . 165
26.1.2 Where `@SuppressWarnings` can be written . . . 165
26.1.3 Good practices when suppressing warnings . . . 165
26.2 `@AssumeAssertion` string in an `assert` message . . . 166
26.2.1 Suppressing warnings and defensive programming . . . 167
26.3 `-AsuppressWarnings` command-line option . . . 167
26.4 `-AskipUses` and `-AonlyUses` command-line options . . . 168
26.5 `-AskipDefs` and `-AonlyDefs` command-line options . . . 168
26.6 `-Alint` command-line option . . . 168
26.7 Don't run the processor . . . 169
26.8 Checker-specific mechanisms . . . 169

27 Handling legacy code 170
27.1 Checking partially-annotated programs: handling unannotated code 170
27.1.1 Declaration annotations 170

28 Type inference 172
28.1 Local type inference during type-checking 172
28.2 Type inference to annotate a program 172
28.2.1 Type inference tools 173
28.3 Whole-program inference 173
28.3.1 Whole-program inference ignores some code 174
28.3.2 Manually checking whole-program inference results 174
28.3.3 How whole-program inference works 175

29 Annotating libraries 176
29.1 Tips for annotating a library 176
29.1.1 Don't change the code 177
29.1.2 Library annotations should reflect the specification, not the implementation 177
29.1.3 Report bugs upstream 177
29.1.4 Fully annotate the library, or indicate which parts you did not 177
29.1.5 Verify your annotations 177
29.2 Creating an annotated library 178
29.3 Creating an annotated JDK 179
29.4 Compiling partially-annotated libraries 179
29.4.1 The `-AuseDefaultsForUncheckedCode=source,bytecode` command-line argument 180
29.5 Using stub classes 180
29.5.1 Using a stub file 180
29.5.2 Stub file format 181
29.5.3 Creating a stub file 181
29.5.4 Troubleshooting stub libraries 182
29.6 Troubleshooting/debugging annotated libraries 182

30 How to create a new checker 184
30.1 How checkers build on the Checker Framework 185
30.2 The parts of a checker 185
30.3 Compiling and using a custom checker 186
30.3.1 Tips for creating a checker 186
30.4 Annotations: Type qualifiers and hierarchy 187
30.4.1 Defining the type qualifiers 188
30.4.2 Declaratively defining the qualifier hierarchy 189
30.4.3 Procedurally defining the qualifier hierarchy 189
30.4.4 Defining a default annotation 190
30.4.5 Relevant Java types 190
30.4.6 Do not re-use type qualifiers 190
30.4.7 Completeness of the type hierarchy 190
30.4.8 Annotations whose argument is a Java expression (dependent type annotations) 191
30.5 The checker class: Compiler interface 192
30.5.1 Indicating supported annotations 192
30.5.2 Bundling multiple checkers 193
30.5.3 Providing command-line options 193
30.6 Visitor: Type rules 194
30.6.1 AST traversal 195
30.6.2 Avoid hardcoding 195

30.7 Type factory: Implicit annotations (type introduction rules) . . . 195
30.7.1 Declaratively specifying implicit annotations . . . 195
30.7.2 Procedurally specifying implicit annotations . . . 196
30.8 Dataflow: enhancing flow-sensitive type qualifier inference . . . 196
30.8.1 Determine expressions to refine the types of . . . 196
30.8.2 Create required class . . . 197
30.8.3 Override methods that handle Nodes of interest . . . 197
30.8.4 Implement the refinement . . . 198
30.8.5 Disabling flow-sensitive inference . . . 199
30.9 Annotated JDK . . . 199
30.10 Testing framework . . . 199
30.11 Debugging options . . . 200
30.11.1 Amount of detail in messages . . . 200
30.11.2 Stub and JDK libraries . . . 200
30.11.3 Progress tracing . . . 200
30.11.4 Saving the command-line arguments to a file . . . 200
30.11.5 Visualizing the dataflow graph . . . 200
30.11.6 Miscellaneous debugging options . . . 201
30.11.7 Examples . . . 201
30.11.8 Using an external debugger . . . 201
30.12 Documenting the checker . . . 201
30.13 javac implementation survival guide . . . 202
30.13.1 Checker access to compiler information . . . 202
30.13.2 How a checker fits in the compiler as an annotation processor . . . 203
30.14 Integrating a checker with the Checker Framework . . . 204

31 Integration with external tools **205**
31.1 Android Studio 3.0 and the Android Gradle Plugin 3.0 . . . 205
31.2 Ant task . . . 206
31.2.1 Explanation . . . 207
31.3 Eclipse . . . 207
31.3.1 Using an Ant task . . . 207
31.3.2 Troubleshooting Eclipse . . . 208
31.4 Gradle . . . 208
31.5 IntelliJ IDEA . . . 208
31.6 Javac compiler . . . 209
31.7 Maven . . . 209
31.8 NetBeans . . . 212
31.8.1 Adding a checker via the Project Properties window . . . 212
31.8.2 Adding a checker via an ant target . . . 212
31.9 tIDE . . . 214
31.10 Type inference tools . . . 214

32 Frequently Asked Questions (FAQs) **215**
32.1 Motivation for pluggable type-checking . . . 216
32.1.1 I don't make type errors, so would pluggable type-checking help me? . . . 216
32.1.2 Should I use pluggable types (type qualifiers) or Java subtypes? . . . 217
32.2 Getting started . . . 218
32.2.1 How do I get started annotating an existing program? . . . 218
32.2.2 Which checker should I start with? . . . 218
32.2.3 How can I join the checker-framework-dev mailing list? . . . 218
32.3 Usability of pluggable type-checking . . . 218

32.3.1 Are type annotations easy to read and write? 218
32.3.2 Will my code become cluttered with type annotations? 219
32.3.3 Will using the Checker Framework slow down my program? Will it slow down the compiler? 219
32.3.4 How do I shorten the command line when invoking a checker? 219
32.3.5 Method pre-condition contracts, including formal parameter annotations, make no sense for public methods 219
32.4 How to handle warnings and errors 220
32.4.1 What should I do if a checker issues a warning about my code? 220
32.4.2 What does a certain Checker Framework warning message mean? 220
32.4.3 Can a pluggable type-checker guarantee that my code is correct? 220
32.4.4 What guarantee does the Checker Framework give for concurrent code? 221
32.4.5 How do I make compilation succeed even if a checker issues errors? 221
32.4.6 Why does the checker always say there are 100 errors or warnings? 221
32.4.7 Why does the Checker Framework report an error regarding a type I have not written in my program? 221
32.4.8 How can I do run-time monitoring of properties that were not statically checked? 221
32.5 False positive warnings 222
32.5.1 What is a “false positive” warning? 222
32.5.2 How can I improve the Checker Framework to eliminate a false positive warning? 222
32.5.3 Why doesn’t the Checker Framework infer types for fields and method return types? 222
32.5.4 Why doesn’t the Checker Framework track relationships between variables? 223
32.5.5 Why isn’t the Checker Framework path-sensitive? 225
32.6 Syntax of type annotations 225
32.6.1 What is a “receiver”? 225
32.6.2 What is the meaning of an annotation after a type, such as `@NonNull Object @Nullable`? 226
32.6.3 What is the meaning of array annotations such as `@NonNull Object @Nullable []`? 226
32.6.4 What is the meaning of varargs annotations such as `@English String @NonEmpty ...`? 226
32.6.5 What is the meaning of a type qualifier at a class declaration? 227
32.6.6 Why shouldn’t a qualifier apply to both types and declarations? 227
32.6.7 How do I annotate a fully-qualified type name? 228
32.6.8 What is the difference between type annotations and declaration annotations? 228
32.7 Semantics of type annotations 228
32.7.1 How can I handle typestate, or phases of my program with different data properties? 228
32.7.2 Why are explicit and implicit bounds defaulted differently? 229
32.7.3 Why are type annotations declared with `@Retention(RetentionPolicy.RUNTIME)`? 229
32.8 Creating a new checker 230
32.8.1 How do I create a new checker? 230
32.8.2 What properties can and cannot be handled by type-checking? 230
32.8.3 Why is there no declarative syntax for writing type rules? 230
32.9 Tool questions 230
32.9.1 How does pluggable type-checking work? 230
32.9.2 What classpath is needed to use an annotated library? 231
32.9.3 Why do `.class` files contain more annotations than the source code? 231
32.9.4 Is there a type-checker for managing checked and unchecked exceptions? 231
32.10 Relationship to other tools 231
32.10.1 Why not just use a bug detector (like FindBugs)? 231
32.10.2 How does the Checker Framework compare with Eclipse’s null analysis? 232
32.10.3 How does the Checker Framework compare with the JDK’s `Optional` type? 232
32.10.4 How does pluggable type-checking compare with JML? 232
32.10.5 Is the Checker Framework an official part of Java? 233
32.10.6 What is the relationship between the Checker Framework and JSR 305? 233

32.10.7 What is the relationship between the Checker Framework and JSR 308? 233

33 Troubleshooting, getting help, and contributing 234
33.1 Common problems and solutions 234
33.1.1 Unable to compile the Checker Framework 234
33.1.2 Unable to run the checker, or checker crashes 234
33.1.3 Unexpected type-checking results 237
33.1.4 Unexpected compilation output when running javac without a pluggable type-checker 239
33.2 How to report problems (bug reporting) 240
33.2.1 Problems with annotated libraries 240
33.3 Building from source 241
33.3.1 Install prerequisites 241
33.3.2 Obtain the source 241
33.3.3 Build the Checker Framework 242
33.3.4 Build the Checker Framework Manual (this document) 242
33.3.5 Configure Eclipse to edit the Checker Framework 242
33.3.6 Enable Travis continuous integration builds 243
33.3.7 Code style 243
33.4 Contributing fixes (creating a pull request) 244
33.5 Publications 244
33.6 Comparison to other tools 245
33.7 Credits and changelog 246
33.8 License 246

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 25)
2. Initialization Checker to ensure all fields are set in the constructor (see Chapter 3.9, page 35)
3. Map Key Checker to track which values are keys in a map (see Chapter 4, page 50)
4. Optional Checker for errors in using the `Optional` type (see Chapter 5, page 53)
5. Interning Checker for errors in equality testing and interning (see Chapter 6, page 55)
6. Lock Checker for concurrency and lock errors (see Chapter 7, page 59)
7. Index Checker for array accesses (see Chapter 8, page 70)
8. Fake Enum Checker to allow type-safe fake enum patterns and type aliases or typedefs (see Chapter 9, page 79)
9. Tainting Checker for trust and security errors (see Chapter 10, page 83)
10. Regex Checker to prevent use of syntactically invalid regular expressions (see Chapter 11, page 85)
11. Format String Checker to ensure that format strings have the right number and type of `%` directives (see Chapter 12, page 88)
12. Internationalization Format String Checker to ensure that i18n format strings have the right number and type of `{}` directives (see Chapter 13, page 95)
13. Property File Checker to ensure that valid keys are used for property files and resource bundles (see Chapter 14, page 101)
14. Internationalization Checker to ensure that code is properly internationalized (see Chapter 14.2, page 102)
15. Signature String Checker to ensure that the string representation of a type is properly used, for example in `Class.forName` (see Chapter 15, page 104)
16. GUI Effect Checker to ensure that non-GUI threads do not access the UI, which would crash the application (see Chapter 16, page 107)
17. Units Checker to ensure operations are performed on correct units of measurement (see Chapter 17, page 112)
18. Signedness Checker to ensure unsigned and signed values are not mixed (see Chapter 18, page 116)
19. Constant Value Checker to determine whether an expression's value can be known at compile time (see Chapter 19, page 119)
20. Aliasing Checker to identify whether expressions have aliases (see Chapter 20, page 123)
21. Subtyping Checker for customized checking without writing any code (see Chapter 22, page 132)
22. Third-party checkers that are distributed separately from the Checker Framework (see Chapter 23, page 137)

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 30.

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 24–33), so that you will know what is available and can find it later. Skip Chapter 30 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; see Chapter 28.2, page 172.

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 try the Checker Framework without installing it, use the Checker Framework Live Demo webpage.)

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 33.3, page 241).

Requirement: You must have **JDK 8** installed. You can get the JDK from Oracle or elsewhere.

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

1. Download the Checker Framework distribution:
   `https://checkerframework.org/checker-framework-2.5.1.zip`
2. Unzip it to create a `checker-framework` directory.
3. Configure your IDE, build system, or command shell to include the Checker Framework on the classpath. Choose the appropriate section of Chapter 31.

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 (`https://checkerframework.org/tutorial/`), which walks you through how to use the Checker Framework on the command line. There is also a Nullness Checker tutorial (`https://github.com/glts/safer-spring-petclinic/wiki`) by David Bürgin.

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 (`https://checkerframework.org/tutorial/`) that gives more extensive instructions for using the Checker Framework on

the command line, and a Nullness Checker tutorial (`https://github.com/glts/safer-spring-petclinic/wiki`) by David Bürgin.

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 *`javac`* is set as in Section 31.6.
   (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 31).

   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
```

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 Section 3.3.7.) 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 29: Annotating libraries
- Chapter 30: How to create a new checker
- Chapter 31: Integration with external tools

Finally, there is a tutorial (`https://checkerframework.org/tutorial/`) that walks you through using the Checker Framework on the command line, and a separate Nullness Checker tutorial (`https://github.com/glts/safer-spring-petclinic/wiki`) by David Bürgin.

2.1 Writing annotations

The syntax of type annotations in Java is specified by the Java Language Specification (Java SE 8 edition).

Java 5 defines declaration annotations such as `@Deprecated`, which apply to a class, method, or field, but do not apply to the method's return type or the field's type. They are typically written on their own line in the source code.

Java 8 defines type annotations, which you write immediately before any use of a type, including in generics and casts. Because array levels are types and receivers have types, you can also write type annotations on them. Here are a few examples of type annotations:

```
@Interned String intern() { ... }             // return value
int compareTo(@NonNull String other) { ... }    // parameter
String toString(@Tainted 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 = (@Initialized Date) beingConstructed;  // cast
```

You only need to write annotations on method signatures and fields. Annotations within method bodies are inferred for you; for more details, see Section 25.4.

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. A concrete example (using the Nullness Checker) is:

```
javac -processor nullness MyFile.java
```

where `javac` is as specified in Section 31.6.

You can also run a checker from within your favorite IDE or build system. See Chapter 31 for details about Ant (Section 31.2), Maven (Section 31.7), Gradle (Section 31.4), IntelliJ IDEA (Section 31.5), Eclipse (Section 31.3), and tIDE (Section 31.9), and about customizing other IDEs and build tools.

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 Using annotated libraries

When your code uses a library that is not currently being compiled, the Checker Framework looks up the library's annotations in its class files.

Some projects are already distributed with type annotations by their maintainers, so you do not need to do anything special. Over time, this should become more common.

For some other libraries, the Checker Framework developers have provided an annotated version of the library. During type-checking, you should use the annotated version of the library to improve type-checking results (to issue fewer false positive warnings). When doing ordinary compilation or while running your code, you can use either the annotated library or the regular distributed version of the library — they behave identically.

The annotated libraries appear in the `org.checkerframework.annotatedlib` group in the Central Repository.

- If your project stores `.jar` files locally, then download the `.jar` file from the Central Repository.
- If your project manages dependencies using a tool such as Gradle or Maven, then update your buildfile to use the `org.checkerframework.annotatedlib` group. For example, in `build.gradle`, change

  ```
  api group: 'org.apache.bcel', name: 'bcel', version: '6.2'
  ```

  to

  ```
  api group: 'org.checkerframework.annotatedlib', name: 'bcel', version: '6.2'
  ```

  Use the same version number. (Sometimes you will use a slightly larger number, if the Checker Framework developers have improved the type annotations since the last release by the upstream maintainers.) If a newer version of the upstream library is available, then open an issue requesting that the `org.checkerframework.annotatedlib` version be updated.

There are two special cases. The annotated JDK is automatically put on your classpath; you don't have to do anything special for it. For the Javadoc classes in the JDK's com.sun.javadoc package, use the stub file `checker/resources/javadoc.astub`.

If a library you wish to use is not available, then you can request that it be annotated, or you can volunteer to write the annotations (see Chapter 29, page 176), which will help you and all other Checker Framework users.

2.2.2 Distributing your annotated project

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. Developers perform pluggable type-checking in-house to detect errors and verify their absence. When you create `.class` files, run 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.

2.2.3 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

- `-AsuppressWarnings` Suppress all warnings matching the given key; see Section 26.3.
- `-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.
- `-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 25.4.5.
- `-AassumeAssertionsAreEnabled`, `-AassumeAssertionsAreDisabled` Whether to assume that assertions are enabled or disabled; see Section 25.4.6.
- `-AignoreRangeOverflow` Ignore the possibility of overflow for range annotations such as `@IntRange`; see Section 19.3.
- `-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. If you use this, don't supply `-Werror`, which is a javac argument to halt compilation if a warning is issued.

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 25.4.5.
- `-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.
- `-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.
- `-AuseDefaultsForUncheckedCode` Enables/disables unchecked code defaults. Takes arguments "source,bytecode". "-source,-bytecode" is the (unsound) default setting. "bytecode" specifies whether the checker should apply unchecked code defaults to bytecode; see Section 25.3.5. Outside the scope of any relevant `@AnnotatedFor` annotation, "source" specifies whether unchecked code default annotations are applied to source code and suppress all type-checking warnings; see Section 29.4.
- `-AconcurrentSemantics` Whether to assume concurrent semantics (field values may change at any time) or sequential semantics; see Section 32.4.4.

- -AconservativeUninferredTypeArguments Whether an error should be issued if type arguments could not be inferred and whether method type arguments that could not be inferred should use conservative defaults. By default, such type arguments are (largely) ignored in later checks. Passing this option uses a conservative value instead. See Issue 979.

Type-checking modes: enable/disable functionality

- -Alint Enable or disable optional checks; see Section 26.6.
- -AsuggestPureMethods Suggest methods that could be marked @SideEffectFree, @Deterministic, or @Pure; see Section 25.4.5.
- -AresolveReflection Determine the target of reflective calls, and perform more precise type-checking based no that information; see Section 21. -AresolveReflection=debug causes debugging information to be output.
- -Ainfer Output suggested annotations for method signatures and fields. These annotations may reduce the number of type-checking errors when running type-checking in the future; see Section 28.3.
- -AshowSuppressWarningKeys With each warning, show all possible keys to suppress that warning.

Partially-annotated libraries

- -Astubs List of stub files or directories; see Section 29.5.1.
- -AstubWarnIfNotFound Warn if a stub file entry could not be found; see Section 29.5.1.
- -AstubWarnIfOverwritesBytecode Warn if a stub file entry overwrite bytecode information; see Section 29.5.1.
- -AuseDefaultsForUncheckedCode=source Outside the scope of any relevant @AnnotatedFor annotation, use unchecked code default annotations and suppress all type-checking warnings; see Section 29.4.

Debugging

- -AprintAllQualifiers, -AprintVerboseGenerics, -Adetailedmsgtext, -AprintErrorStack, -Anomsgtext Amount of detail in messages; see Section 30.11.1.
- -Aignorejdkastub, -Anocheckjdk, -AstubDebug Stub and JDK libraries; see Section 30.11.2.
- -Afilenames, -Ashowchecks, -AshowInferenceSteps Progress tracing; see Section 30.11.3.
- -AoutputArgsToFile Output the compiler command-line arguments to a file. Useful when the command line is generated and executed by a tool, such as a build system. This produces a standalone command line that can be executed independently of the tool that generated it can make it easier to reproduce, report, and debug issues. For example, the command line can be modified to enable attaching a debugger. See Section 30.11.4.
- -Aflowdotdir, -Averbosecfg, -Acfgviz Draw a visualization of the CFG (control flow graph); see Section 30.11.5.
- -AresourceStats, -AatfDoNotCache, -AatfCacheSize Miscellaneous debugging options; see Section 30.11.6.

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 9) 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 an "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 2.2.4
- -implicit:class Suppresses warnings about implicitly compiled files (not named on the command line); see Section 31.2
- -J Supply an argument to the JVM that is running javac; for example, -J-Xmx2500m to increase its maximum heap size
- -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.

The Checker Framework does not support -source 1.4 or earlier. You must supply -source 1.5 or later, or no -source command-line argument, when running javac.

2.2.4 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.

2.2.5 Shorthand for built-in checkers

Ordinarily, javac's -processor flag requires fully-qualified class names. 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 6) 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 Chapter 26).
  - Suppression of warnings, such as via the @SuppressWarnings annotation (see Chapter 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 2.2.3. 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 33.2).

2.4 Tips about writing annotations

Section 29.1 gives additional tips that are specific to annotating a third-party library.

2.4.1 Write annotations before you run a checker

Before you run a checker, annotate the code, based on its documentation. Then, run the checker to uncover bugs in the code or the documentation.

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

2.4.2 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

Start small. Focus on one specific property that matters to you; in other words, run just one checker rather than multiple ones. You may choose a different checker for different programs. Focus on the most mission-critical or error-prone part of your code; don't try to annotate your whole program at first.

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.

When annotating, be systematic; we recommend annotating an entire class at a time (not just some of the methods) so that you don't lose track of your work or redo work. For example, working class-by-class avoids confusion about whether an unannotated type means you determined that the default is desirable, or it means you didn't yet examine that type. 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. For example, within a package annotate supertypes before you annotated classes that extend or implement them. The reason for this rule is that it is easiest to annotate a class if the code it depends on has already been annotated.

Don't overuse pluggable type-checking. If the regular Java type system can verify a property using Java subclasses, then that is a better choice than pluggable type-checking (see Section 32.1.2).

Annotations are a specification

When you write annotations, you are writing a specification, and you should think about them that way. Start out by understanding the program so that you can write an accurate specification. Sections 2.4.3 and 2.4.4 give more tips about writing specifications.

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. For now, just 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 (that is, of changing the specification for a given method or class): how many other specification changes (added annotations) will it require, and will it conflict with other code? It's best to reason about the desired design, but you can also do an experiment. Suppose you are considering adding an annotation to a method parameter. One approach is to manually examine all callees. A more automated approach is 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.

Chapter 29 tells you how to annotate libraries that your code uses. Section 2.4.6 and Chapter 26 tell you what to do when you are unable to eliminate checker warnings by adding annotations.

Write good code

Avoid complex code, which is more error-prone. If you write your code to be simple and clean enough for the type-checker to verify, then it will also be easier for programmers to understand.

Your code should compile cleanly under the regular Java compiler. If you are not willing to write code that type-checks in Java, then there is little point in using an even more powerful, restrictive type system. As a specific example, your code should not use raw types like `List`; use parameterized types like `List<String>` instead (Section 24.1.1).

Do not write unnecessary annotations.

- 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. You should add annotations inside method bodies only if the checker is unable to infer the correct annotation.
- Do not write annotations that are redundant with defaults. For example, when checking nullness (Chapter 3, page 25), the default annotation is `@NonNull`, in most locations other than some type bounds (Section 25.3.2). When you are starting out, it might seem helpful to write redundant annotations as a reminder, but that's like when beginning programmers write a comment about every simple piece of code:

  ```
  // The below code increments variable i by adding 1 to it.
  i++;
  ```

  As you become comfortable with pluggable type-checking, you will find redundant annotations to be distracting clutter, so avoid putting them in your code in the first place.
- Avoid writing `@SuppressWarnings` annotations unless there is no alternative. It is tempting to think that your code is right and the checker's warnings are false positives. Sometimes they are, but slow down and convince yourself of that before you dismiss them. Section 2.4.6 discusses what to do when a checker issues a warning about your code.

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 specify such parameters as `@NonNull`, indicating the intended use of the method. When you specify the parameter as the `@NonNull` annotation, the checker is able to issue compile-time warnings about possible run-time exceptions, which is its purpose. Specifying the parameter as `@Nullable` would suppress such warnings, which is undesirable. (Since `@NonNull` is the default, you don't have to write anything in the source code to specify the parameter as non-null. You are allowed to write a redundant `@NonNull` annotation, but it is discouraged.)

If a method can possibly throw an 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 in `com/google/common/collect/Multiset.java`:

```
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 @Untainted Date()
(@Untainted 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 {
  @Untainted MyClass() { ... }
}
```

In general, you should only use an annotation on a constructor invocation when you know that the cast is guaranteed to succeed.

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

When you run a type-checker on your code, it is likely to issue warnings or errors. Don't panic! There are three general causes for the warnings:

- There is a bug in your code, such as a possible null dereference. Fix your code to prevent that crash.
- There is a weakness in the annotations you wrote. In other words, the specification you wrote is incorrect or inadequate. Improve the annotations, usually by writing more annotations in order to better express the specification.
- There is a weakness in the type-checker. Your code is safe — it never suffers the error at run time — but the checker cannot prove this fact. If possible, rewrite your code to be simpler for the checker to analyze; this is likely to make it easier for people to understand, too. If that is not possible, suppress the warning (see Chapter 26); be sure to include a code comment explaining how you know the code is correct even though the type-checker cannot deduce that fact.

For each warning issued by the checker, you need to determine which of the above categories it falls into. Here is an effective methodology to do so. It relies mostly on manual code examination, but you may also find it useful to write test cases for your code or do other kinds of analysis, to verify your reasoning.

1. Write an explanation of why your code is correct and it never suffers the error at run time. In other words, this is an English proof that the type-checker's warning is incorrect.
   Don't skip any steps in your proof. (For example, don't write an unsubstantiated claim such as "`x` is non-null here"; instead, give a justification.) Don't let your reasoning rely on facts that you do not write down explicitly. For example, remember that calling a method might change the values of object fields; your proof might need to state that certain methods have no side effects.
   If you cannot write a proof, then there is a bug in your code (you should fix the bug) or your code is too complex for you to understand (you should improve its documentation and/or design).
2. Translate the proof into annotations. Here are some examples.
   - If your proof includes "variable `x` is never `null` at run time", then annotate <x>'s type with `@NonNull`.
   - If your proof includes "method `foo` always returns a legal regular expression", then annotate `foo`'s return type with `@Regex`.
   - If your proof includes "if method `join`'s first argument is non-null, then `join` returns a non-null result", then annotate `join`'s first parameter and return type with `@PolyNull`.
   - If your proof includes "method `processOptions` has already been called and it set field `tz1`", then annotate `processOptions`'s declaration with `@EnsuresNonNull("tz1")`.
   - If your proof includes "method `isEmpty` returned false, so its argument must have been non-null", then annotate `isEmpty`'s declaration with `@EnsuresNonNullIf(expression="#1",result=false)`.

   All of these are examples of correcting weaknesses in the annotations you wrote. The Checker Framework provides many other powerful annotations; you may be surprised how many proofs you can express in annotations. If you need to annotate a method that is defined in a library that your code uses, see Chapter 29, page 176, Annotating Libraries.
   If there are complex facts in your proof that cannot be expressed as annotations, then that is a weakness in the type-checker. For example, the Nullness Checker cannot express "in list `lst`, elements stored at even indices are always non-`null`, but elements stored at odd elements might be `null`." In this case, you have two choices. First, you can suppress the warning (Chapter 26, page 164); be sure to write a comment explaining your reasoning for suppressing the warning. You may wish to submit a feature request (Section 33.2) asking for annotations that handle your use case. Second, you can rewrite the code to make the proof simpler; in the above example, it might be better to use a list of pairs rather than a heterogeneous list.
3. At this point, all the steps in your proof have been formalized as annotations. Re-run the checker and repeat the process for any new or remaining warnings.
   If every step of your proof can be expressed in annotations, but the checker cannot make one of the deductions (it cannot follow one of the steps), then that is a weakness in the type-checker. First, double-check your reasoning. Then, suppress the warning, along with a comment explaining your reasoning (Chapter 26, page 164). The comment is an excerpt from your English proof, and the proof guides you to the best place to suppress the warning. Finally, please submit a bug report so that the checker can be improved in the future (Section 33.2).

If you have trouble understanding a Checker Framework warning message, you can search for its text in this manual. Also see Section 33.1.3 and Chapter 33, Troubleshooting. In particular, Section 33.1.3 explains this same methodology in different words.

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 3.9), and the Map Key Checker (Chapter 4, page 50). 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.

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 3.9.8) and of map key annotations (see Chapter 4, page 50).

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. Options that control soundness:
   - 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.
2. Options that warn about poor code style:
   - 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.

3.2 Nullness annotations

The Nullness Checker uses three separate type hierarchies: one for nullness, one for rawness (Section 3.9.8), and one for map keys (Chapter 4, page 50) 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.)

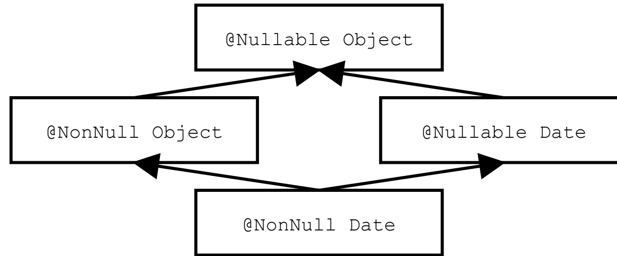


Figure 3.1: Partial type hierarchy for the Nullness type system. Java's `Object` is expressed as `@Nullable Object`. Programmers can omit most type qualifiers, because the default annotation (Section 3.3.2) is usually correct. The Nullness Checker verifies three type hierarchies: this one for nullness, one for initialization (Section 3.9), and one for map keys (Chapter 4, page 50).

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 3.9.

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

@Raw
@NonRaw
@PolyRaw

Figure 3.7 shows its type hierarchy. For details, see Section 3.9.8.

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 50) 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

The Nullness Checker adds implicit qualifiers, reducing the number of annotations that must appear in your code (see Section 25.3). For example, enum types are implicitly non-null, so you never need to write `@NonNull MyEnumType`.

For more information about implicitly-added nullness qualifiers, see the implementation of `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 by writing a `@DefaultQualifier` annotation on a package, class, or method. In the example below, fields are defaulted to `@Nullable` instead of `@NonNull`.

```
@DefaultQualifier(value = Nullable.class, locations = TypeUseLocation.FIELD)
class MyClass {
    Object nullableField = null;
    @NonNull Object nonNullField = new Object();
}
```

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<T> {
    @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

Suppose that you declare an array to contain non-null elements. Currently, the Nullness Checker does not verify that all the elements of the array are initialized to a non-null value. This is an unsoundness in the checker.

To make the Nullness Checker conservatively reject code that may leave a non-null value in an array, use the command-line option `-Alint=forbidnonnullarraycomponents`. The option is currently disabled because it makes the checker issue many false positive errors.

When the `-Alint=forbidnonnullarraycomponents` option is supplied, the following is not allowed:

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

(recall that `Object` means the same thing as `@NonNull Object`).

Instead, your code needs to create a nullable or lazy-nonnull array, initialize each component, and then assign 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.

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`. The refinement lasts until the end of the scope of the test or until `x` may be side-effected. 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 Chapter 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>`.

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 `NullnessUtil.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 need to include `checker-qual.jar` on your classpath at run time. If this is a problem, you can copy 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(Object)` 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 docs/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 docs/examples/NullnessExampleWithWarnings.java
```

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 2.4.2.

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 25.4.5).
- 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 Nullness_Lite: An Unsound Option of the Nullness Checker for fewer false positives

3.7.1 Introduction

Nullness_Lite is a lite option of the Nullness Checker in the Checker Framework to detect nullness bugs in java source files. It disables the following features of Nullness Checker to obtain more desireable traits. To be specific, the Nullness Checker with Nullness_Lite option enabled will be faster and easier to use by delibrately giving up soundness.

Featrue Disabled	Substitute Behavior of the Nullness_Lite option
Initialization Checker	Assume all values (fields & class variables) initialized
Map Key Checker	Assume all keys are in the map and `Map.get(key)` returns `@NonNull`
Invalidation of dataflow	Assume no aliasing and all methods are `@SideEffectFree`
Boxing of primitives	Assume the boxing of primitives return the same object and `BoxedClass.valueOf()` are `@Pure`

3.7.2 Who wants to use Nullness_Lite option?

Java developers who would like to get a trustable analysis on their source files, but hesitate to spend time running full verification tool such as Nullness Checker. They can run the Nullenss_Lite option instead to get a fast glimpse on their files before they upgrade to the Nullness Checker.

3.7.3 How to use the Nullness_Lite?

Please follow the instructions for installation first. To compile the Nullness Checker, please follow the instructions in Section 2.2 for information about running a checker.

Users run Nullness_Lite by passing an extra argument when running Nullness Checker from the command line:

```
javac -processor nullness -ANullnessLite <MyFile.java>
```

* Notice that the behavior is undefined if `-ANullnessLite option` is passed to a different checker.

3.7.4 More information

- See the implementation
- See the evaluation on JUnit4

Refer to the complete Nullness_Lite user manual on GitHub for more details.

3.8 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. In particular, IDE tools such as Eclipse and IntelliJ should be viewed as bug-finding tools rather than verification tools, since they give up precision, soundness, or both in favor of being fast and easy to use. Also see Section 33.6 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, the Checker Framework can type-check your code. It treats annotations from other tools as if you had written the corresponding annotation from the Nullness Checker, as described in Figure 3.2. If the other annotation is a declaration annotation, it may be moved; see Section 27.1.1.

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 3.8.2).

Java permits you to import at most one annotation of a given name. For example, if you use both `android.annotation.NonNull` and `lombok.NonNull` in your source code, then you must write at least one of them in fully-qualified form, as `@android.annotation.NonNull` rather than as `@NonNull`.

3.8.1 Which tool is right for you?

Different tools are appropriate in different circumstances. Here is a brief comparison with FindBugs, but similar points apply to other tools. (For example, the NullAway and Eradicate tools are more like sound type-checking than FindBugs, but all of those tools accept unsoundness — that is, false negatives or missed warnings — in exchange for analysis speed.)

The Checker Framework has a more powerful nullness analysis; FindBugs misses some real errors. FindBugs requires you to annotate your code, but usually not 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.

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

Other annotation		Checker Framework annotation
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.PossiblyNull		
edu.umd.cs.findbugs.annotations.UnknownNullness		
javax.annotation.Nullable		
javax.annotation.CheckForNull	⇒	org.checkerframework.checker.nullness.qual.Nullable
org.eclipse.jdt.annotation.Nullable		
org.eclipse.jgit.annotations.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		
org.springframework.lang.Nullable		

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

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 33.2) so that we can enhance the Checker Framework.

3.8.2 Incompatibility note about FindBugs `@Nullable`

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

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

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

```
// declare getObject() to possibly return null
```

```
@Nullable Object getObject() { ... }

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

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

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.)

3.8.3 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.

See the article "Nothing is better than Java's `Optional` class" for more details and explanation of the benefits of `@Nullable` over `Optional`.

Java's `Optional` class provides utility routines to reduce clutter when using `Optional`. The Nullness Checker provides an `Opt` class that provides all the same methods, but written for regular possibly-null Java references.

3.9 Initialization Checker

The Initialization Checker determines whether an object is initialized or not. For any object that is not fully initialized, the Nullness Checker treats its fields as possibly-null — even fields annotated as `@NonNull`.

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();
  }
```

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

    public MyClass(String arg1, String arg2, String arg3) {
      this.field1 = arg1;
      this.field2 = arg2;
      this.field3 = arg3;
      checkRep();
    }

    // Verify that the representation invariants are satisfied.
    // Works as long as the MyClass fields are initialized, even if the reciever's
    // class is a subclass of MyClass and not all of the subclass fields are initialized.
    private void checkRep(@UnderInitialization(MyClass.class) MyClass this) {
      ...
    }


  }
```

Note that it would not be sound to type @code this as fully @link Initialized anywhere in a constructor (with the exception of final classes, for which the set of all fields is known), because there might be subclasses with uninitialized fields. The following example shows why:

```
class A {
   @NonNull String a;
   public A() {
       a = "";
       // Now, all fields of A are initialized.
       // However, if this constructor is invoked as part of 'new B()', then
       // the fields of B are not yet initialized.
       // If we would type 'this' as @Initialized, then the following call is valid:
       foo();
   }
   void foo() {}
}

class B extends A {
   @NonNull String b;
   @Override
   void foo() {
       // Dereferencing 'b' is ok, since 'this' is @Initialized and 'b' @NonNull.
       // However, when executing 'new B()', this line throws a null-pointer exception.
       b.toString();
   }
}
```

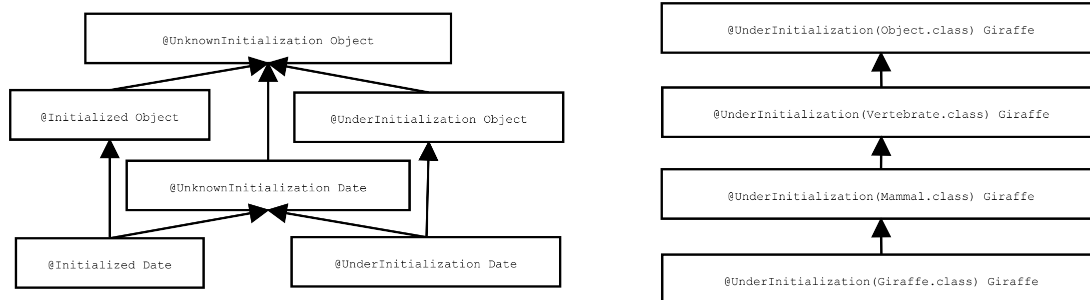


Figure 3.3: Partial type hierarchy for the Initialization type system. `@UnknownInitialization` and `@UnderInitialization` each take an optional parameter indicating how far initialization has proceeded, and the right side of the figure illustrates its type hierarchy in more detail.

3.9.1 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 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.

3.9.2 How an object becomes initialized

Within the constructor, `this` starts out with `@UnderInitialization` type. As soon as all of the `@NonNull` fields in class *C* have been initialized, then `this` is treated as `@UnderInitialization(`*C*`)`. This means that `this` is still being initialized, but all initialization of *C*'s fields is complete, including all fields of supertypes. Eventually, when all constructors complete, the type is `@Initialized`.

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.

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.

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.

3.9.3 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 aField;
  A() {
    aField = new Object();
  }
}
class B extends A {
  Object bField;
  B() {
    super();
    bField = 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.

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`

3.9.4 Method calls from the constructor

Consider the following incorrect program.

```
class A {
  Object aField;
  A() {
    aField = new Object();
    process(5);  // illegal call
  }
  public void process(int arg) { ... }
}
```

The call to `process()` is not legal. `process()` is declared to be called on a fully-initialized receiver, which is the default if you do not write a different initialization annotation. At the call to `process()`, all the fields of `A` have been set, but `this` is not fully initialized because fields in subclasses of `A` have not yet been set. The type of `this` is `@UnderInitialization(A.class)`, meaning that `this` is partially-initialized, with the `A` part of initialization done but the initialization of subclasses not yet complete.

The Initialization Checker output indicates this problem:

```
Client.java:7: error: [method.invocation.invalid] call to process(int) not allowed on the given receiver.
    process(5);  // illegal call
           ^
  found   : @UnderInitialization(A.class) A
  required: @Initialized A
```

Here is a subclass and client code that crashes with a `NullPointerException`.

x.f	f is @NonNull	f is @Nullable
x is @Initialized	@Initialized @NonNull	@Initialized @Nullable
x is @UnderInitialization	@UnknownInitialization @Nullable	@UnknownInitialization @Nullable
x is @UnknownInitialization	@UnknownInitialization @Nullable	@UnknownInitialization @Nullable

Figure 3.5: Initialization rules for reading a `@NotOnlyInitialized` field `f`.

```
class B extends A {
  List<Integer> processed;
  B() {
    super();
    processed = new ArrayList<Integer>();
  }
  @Override
  public void process(int arg) {
    super();
    processed.add(arg);
  }
}
class Client {
  public static void main(String[] args) {
    new B();
  }
}
```

You can correct the problem in multiple ways.

One solution is to not call methods that can be overridden from the constructor: move the call to `process()` to after the constructor has completed.

Another solution is to change the declaration of `process()`:

```
public void process(@UnderInitialization(A.class) A this, int arg) { ... }
```

If you choose this solution, you will need to rewrite the definition of `B.process()` so that it is consistent with the declared receiver type.

A non-solution is to prevent subclasses from overriding `process()` by using `final` on the method. This doesn't work because even if `process()` is not overridden, it might call other methods that are overridden.

As final classes cannot have subclasses, they can be handled more flexibly: once all fields of the final class have been initialized, `this` is fully initialized.

3.9.5 Initialization of circular data structures

There is one final aspect of the initialization type system to be considered: the rules governing reading and writing to objects that are currently under initialization (both reading from fields of objects under initialization, as well as writing objects under initialization to fields). By default, only fully-initialized objects can be stored in a field of another object. If this was the only option, then it would not be possible to create circular data structures (such as a doubly-linked list) where fields have a `@NonNull` type. However, the annotation `@NotOnlyInitialized` can be used to indicate that a field can store objects that are currently under initialization. In this case, the rules for reading and writing to that field become a little bit more interesting, to soundly support circular structures.

The rules for reading from a `@NotOnlyInitialized` field are summarized in Figure 3.5. Essentially, nothing is known about the initialization status of the value returned unless the receiver was `@Initialized`.

Similarly, Figure 3.6 shows under which conditions an assignment `x.f = y` is allowed for a `@NotOnlyInitialized` field `f`. If the receiver `x` is `@UnderInitialization`, then any `y` can be of any initialization state. If `y` is known to be fully initialized, then any receiver is allowed. All other assignments are disallowed.

x.f = y	y is @Initialized	y is @UnderInitialization	y is @UnknownInitialization
x is @Initialized	yes	no	no
x is @UnderInitialization	yes	yes	yes
x is @UnknownInitialization	yes	no	no

Figure 3.6: Rules for deciding when an assignment `x.f = y` is allowed for a `@NotOnlyInitialized` field `f`.

These rules allow for the safe initialization of circular structures. For instance, consider a doubly linked list:

```
class List<T> {
  @NotOnlyInitialized
  Node<T> sentinel;

  public List() {
    this.sentinel = new Node<T>(this);
  }

  void insert(@Nullable T data) {
    this.sentinel.insertAfter(data);
  }

  public static void main() {
    List<Integer> l = new List<Integer>();
    l.insert(1);
    l.insert(2);
  }
}

class Node<T> {
  @NotOnlyInitialized
  Node<T> prev;

  @NotOnlyInitialized
  Node<T> next;

  @NotOnlyInitialized
  List parent;

  @Nullable
  T data;

  // for sentinel construction
  Node(@UnderInitialization List parent) {
    this.parent = parent;
    this.prev = this;
    this.next = this;
  }

  // for data node construction
  Node(Node<T> prev, Node<T> next, @Nullable T data) {
    this.parent = prev.parent;
    this.prev = prev;
```

```
    this.next = next;
    this.data = data;
  }

  void insertAfter(@Nullable T data) {
    Node<T> n = new Node<T>(this, this.next, data);
    this.next.prev = n;
    this.next = n;
  }
}
```

3.9.6 How to handle warnings

“error: the constructor does not initialize fields: . . . ”

Like any warning, “error: the constructor does not initialize fields: . . . ” indicates that either your annotations are incorrect or your code is buggy. You can fix either the annotations or the code.

- 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.

“call to . . . is not allowed on the given receiver”

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 32.6.1) 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 by using `@UnderInitialization` or `@UnknownInitialization`.

```
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`.

3.9.7 More details about initialization checking

Suppressing warnings You can suppress warnings related to partially-initialized objects with `@SuppressWarnings("initialization")`. This can be placed on a single field; on a constructor; or on a class to suppress all initialization warnings for all constructors.

To disable initialization checking, supply the command-line argument `-AsuppressWarnings=uninitialized`. Do *not* use `-AsuppressWarnings=initialization`, because doing so will disable all nullness checking as well as all initialization checking. (That is because of an implementation detail of the Nullness and Initialization Checkers: they are actually the same checker, rather than being two separate checkers that are aggregated together.)

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.

3.9.8 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 3.9) uses the three annotations `@Initialized`, `@UnknownInitialization`, and `@UnderInitialization`. We recommend that you use it.

The other (described here in Section 3.9.8) 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.NullnessRawnessChecker` 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();
    this.f = new Object();
  }
  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();
  }
```

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
```

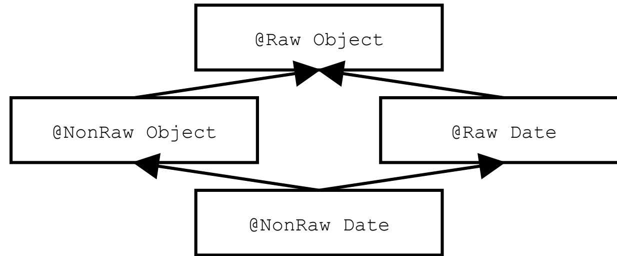


Figure 3.7: Partial type hierarchy for the Rawness Initialization type system.

```
  @EnsuresNonNull({"field2", "field3"})
  private void init_other_fields(@Raw MyClass this) {
    field2 = new Object();
    field3 = new Object();
  }
}
```

Rawness qualifiers

The rawness hierarchy is shown in Figure 3.7. The rawness hierarchy contains these qualifiers:

@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 3.8 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 in class *C* have been initialized, then `this` is treated as `@NonRaw(C)`. This means that `this` is still being initialized, but all initialization of *C*'s fields is complete, including all fields of supertypes. Eventually, when all constructors complete, the type is simply `@NonRaw`.

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.

Declarations	Expression	Expression's nullness type, or checker error

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

`@NonNull @NonRaw C a;`	`a`
	`a.f`
	`a.g`
		`@NonNull`
		`@NonNull`
		`@Nullable`

`@NonNull @Raw C b;`	`b`
	`b.f`
	`b.g`
		`@NonNull`
		`@MonotonicNonNull`
		`@Nullable`

`@Nullable @NonRaw C c;`	`c`
	`c.f`
	`c.g`
		`@Nullable`
		error: deref of nullable
		error: deref of nullable

`@Nullable @Raw C d;`	`d`
	`d.f`
	`d.g`
		`@Nullable`
		error: deref of nullable
		error: deref of nullable

Figure 3.8: 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.

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.

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!

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 25) 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 164.

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.

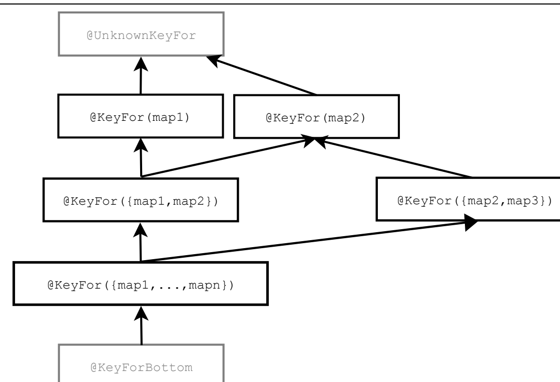


Figure 4.1: The subtyping relationship of the Map Key Checker's qualifiers. `@KeyFor(A)` is a supertype of `@KeyFor(B)` if and only if `A` is a subset of `B`. Qualifiers in gray are used internally by the type system but should never be written by a programmer.

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: 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

Optional Checker for possibly-present data

Java 8 introduced the `Optional` class, a container that is either empty or contains a non-null value.

Using `Optional` is intended to help programmers remember to check whether data is present or not. However, `Optional` itself is prone to misuse. The article Nothing is better than the `Optional` type gives reasons to use regular nullable references rather than `Optional`. However, if you do use `Optional`, then the Optional Checker will help you avoid `Optional`'s pitfalls.

Stuart Marks gave 7 rules to avoid problems with Optional:

1. Never, ever, use `null` for an `Optional` variable or return value.
2. Never use `Optional.get()` unless you can prove that the Optional is present.
3. Prefer alternative APIs over `Optional.isPresent()` and `Optional.get()`.
4. It's generally a bad idea to create an `Optional` for the specific purpose of chaining methods from it to get a value.
5. If an Optional chain has a nested `Optional` chain, or has an intermediate result of `Optional`, it's probably too complex.
6. Avoid using `Optional` in fields, method parameters, and collections.
7. Don't use an `Optional` to wrap any collection type (`List`, `Set`, `Map`). Instead, use an empty collection to represent the absence of values.

Rule #1 is guaranteed by the Nullness Checker (Chapter 3, page 25). Rules #2–#7 are guaranteed by the Optional Checker, described in this chapter. (Exception: Rule #5 is not yet implemented and will be checked by the Optional Checker in the future.)

Use of the Optional Checker guarantees that your program will not suffer a `NullPointerException` nor a `NoSuchElementException` when calling methods on an expression of `Optional` type.

5.1 How to run the Optional Checker

```
javac -processor optional MyFile.java ...
```

5.2 Optional annotations

These qualifiers make up the Optional type system:

`@MaybePresent` The annotated `Optional` container may or may not contain a value. This is the default type.
`@Present` The annotated `Optional` container definitely contains a (non-null) value.
`@PolyPresent` indicates qualifier polymorphism (see Section 24.2).

The subtyping hierarchy of the Optional Checker's qualifiers is shown in Figure 5.1.

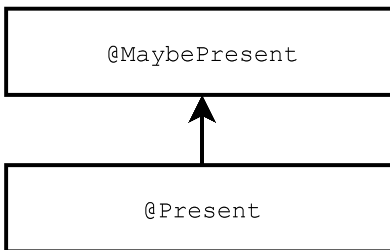


Figure 5.1: The subtyping relationship of the Optional Checker's qualifiers.

Chapter 6

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. (Not every value of the given type is necessarily interned; it is possible for two objects of the class to be `equals()` to one another, even if one of them 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 searching for the index of an 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.

To run the Interning Checker, supply the `-processor org.checkerframework.checker.interning.InterningChecker` command-line option to javac. For examples, see Section 6.4.

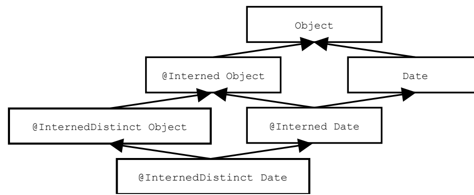


Figure 6.1: Type hierarchy for the Interning type system.

6.1 Interning annotations

These qualifiers are part of the Interning type system:

@Interned indicates a type that includes only interned values (no non-interned values).

@InternedDistinct indicates a type such that each value is not `equals()` to any other Java value. For details, see Section 6.2.2.

@UnknownInterned indicates a type whose values might or might not be interned. It is used internally by the type system and is not written by programmers.

@PolyInterned indicates qualifier polymorphism (see Section 24.2).

@UsesObjectEquals is a class annotation (not a 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.

6.2 Annotating your code with @Interned

In order to perform checking, you must annotate your code with the `@Interned` type annotation. A type annotated with `@Interned` contains the canonical representation of an object:

```
          String s1 = ...;  // type is (uninterned) "String"
@Interned String s2 = ...;  // type is "@Interned String"
```

The Interning Checker 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.

6.2.1 Implicit qualifiers

The Interning Checker adds implicit qualifiers, reducing the number of annotations that must appear in your code (see Section 25.3). 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`.

6.2.2 InternedDistinct: values not equals() to any other value

The `@InternedDistinct` annotation represents values that are not `equals()` to any other value. Suppose expression `e` has type `@InternedDistinct`. Then `e.equals(x) == (e == x)`. Therefore, it is legal to use == whenever at least one of the operands has type `@InternedDistinct`.

`@InternedDistinct` is stronger (more restrictive) than `@Interned`. For example, consider these variables:

```
@Interned String i = "22";
          String s = new Integer(22).toString();
```

The variable `i` is not `@InternedDistinct` because `i.equals(s)` is true.

`@InternedDistinct` is not as restrictive as stating that all objects of a given Java type are interned.

The `@InternedDistinct` annotation is rarely used, because it arises from coding paradigms that are tricky to reason about. One use is on static fields that hold canonical values of a type. Given this declaration:

```
class MyType {
  final static @InternedDistinct MyType SPECIAL = new MyType(...);
  ...
}
```

it would be legal to write `myValue == MyType.SPECIAL` rather than `myValue.equals(MyType.SPECIAL)`.

The `@InternedDistinct` is trusted (not verified), because it would be too complex to analyze the `equals()` method to ensure that no other value is `equals()` to a `@InternedDistinct` value. You will need to manually verify that it is only written in locations where its contract is satisfied. For example, here is one set of guidelines that you could check manually:

- The constructor is private.
- The factory method returns the canonical version for certain values.
- The class is final, so that subclasses cannot violate these properties.

6.3 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. As a special case, the operation is permitted if either argument is of `@InternedDistinct` type
2. When a non-`@Interned` type is used where an `@Interned` type is expected.

This example shows both sorts of problems:

```
                    Date  date;
          @Interned Date idate;
  @InternedDistinct Date ddate;
  ...
  if (date == idate) ...  // error: reference equality test is unsafe
  idate = date;           // error: idate’s referent might no longer be interned
  ddate = idate;          // error: idate’s referent might be equals() to some other value
```

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

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.

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

Figure 6.2: Correspondence between other interning annotations and the Checker Framework's annotations.

6.3.1 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. (The Interning Checker should make use of the Value 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.)

6.4 Examples

To try the Interning Checker on a source file that uses the `@Interned` qualifier, use the following command:

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

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`.

6.5 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, the Checker Framework can type-check your code. It treats annotations from other tools as if you had written the corresponding annotation from the Interning Checker, as described in Figure 6.2. If the other annotation is a declaration annotation, it may be moved; see Section 27.1.1.

Chapter 7

Lock Checker

The Lock Checker prevents certain concurrency errors by enforcing a locking discipline. A locking discipline indicates which locks must be held when a given operation occurs. You express the locking discipline by declaring a variable's type to have the qualifier `@GuardedBy("lockexpr")`. This indicates that the variable's value may be dereferenced only if the given lock is held.

To run the Lock Checker, supply the `-processor org.checkerframework.checker.lock.LockChecker` command-line option to javac. The `-AconcurrentSemantics` command-line option is always enabled for the Lock Checker (see Section 32.4.4).

7.1 What the Lock Checker guarantees

The Lock Checker gives the following guarantee. Suppose that expression *e* has type `@GuardedBy({"x", "y.z"})`. Then the value computed for *e* is only dereferenced by a thread when the thread holds locks `x` and `y.z`. Dereferencing a value is reading or writing one of its fields. The guarantee about *e*'s value holds not only if the expression *e* is dereferenced directly, but also if the value was first copied into a variable, returned as the result of a method call, etc. Copying a reference is always permitted by the Lock Checker, regardless of which locks are held.

A lock is held if it has been acquired but not yet released. Java has two types of locks. A monitor lock is acquired upon entry to a `synchronized` method or block, and is released on exit from that method or block. An explicit lock is acquired by a method call such as `Lock.lock()`, and is released by another method call such as `Lock.unlock()`. The Lock Checker enforces that any expression whose type implements `Lock` is used as an explicit lock, and all other expressions are used as monitor locks.

Ensuring that your program obeys its locking discipline is an easy and effective way to eliminate a common and important class of errors. If the Lock Checker issues no warnings, then your program obeys its locking discipline. However, your program might still have other types of concurrency errors. For example, you might have specified an inadequate locking discipline because you forgot some `@GuardedBy` annotations. Your program might release and re-acquire the lock, when correctness requires it to hold it throughout a computation. And, there are other concurrency errors that cannot, or should not, be solved with locks.

7.2 Lock annotations

This section describes the lock annotations you can write on types and methods.

7.2.1 Type qualifiers

`@GuardedBy`*(exprSet)* If a variable `x` has type `@GuardedBy("expr")`, then a thread may dereference the value referred to by `x` only when the thread holds the lock that *expr* currently evaluates to.

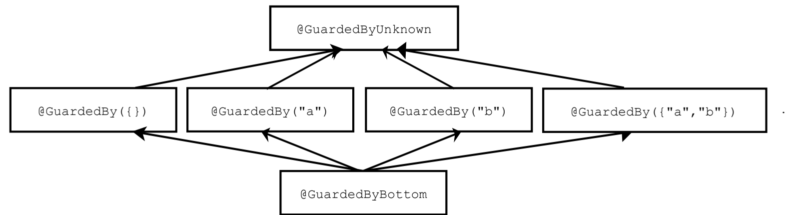


Figure 7.1: The subtyping relationship of the Lock Checker's qualifiers. `@GuardedBy({})` is the default type qualifier for unannotated types (except all CLIMB-to-top locations other than upper bounds and exception parameters — see Section 25.3.2).

The `@GuardedBy` annotation can list multiple expressions, as in `@GuardedBy({"`*expr1*`", "`*expr2*`"})`, in which case the dereference is permitted only if the thread holds all the locks.

Section 25.5 explains which expressions the Lock Checker is able to analyze as lock expressions. These include `<self>`, i.e. the value of the annotated reference (non-primitive) variable. For example, `@GuardedBy("<self>") Object o` indicates that the value referenced by `o` is guarded by the intrinsic (monitor) lock of the value referenced by `o`.

`@GuardedBy({})`, which means the value is always allowed to be dereferenced, is the default type qualifier that is used for all locations where the programmer does not write an explicit locking type qualifier (except all CLIMB-to-top locations other than upper bounds and exception parameters — see Section 25.3.2). (Section 7.5.4 discusses this choice.) It is also the conservative default type qualifier for method parameters in unannotated libraries (see Chapter 29, page 176).

`@GuardedByUnknown` If a variable `x` has type `@GuardedByUnknown`, then it is not known which locks protect `x`'s value. Those locks might even be out of scope (inaccessible) and therefore unable to be written in the annotation. The practical consequence is that the value referred to by `x` can never be dereferenced.

Any value can be assigned to a variable of type `@GuardedByUnknown`. In particular, if it is written on a formal parameter, then any value, including one whose locks are not currently held, may be passed as an argument.

`@GuardedByUnknown` is the conservative default type qualifier for method receivers in unannotated libraries (see Chapter 29, page 176).

`@GuardedByBottom` If a variable `x` has type `@GuardedByBottom`, then the value referred to by `x` is `null` and can never be dereferenced.

Figure 7.1 shows the type hierarchy of these qualifiers. All `@GuardedBy` annotations are incomparable: if *exprSet1* $\neq$ *exprSet2*, then `@GuardedBy(`*exprSet1*`)` and `@GuardedBy(`*exprSet2*`)` are siblings in the type hierarchy. You might expect that `@GuardedBy({"x", "y"}) T` is a subtype of `@GuardedBy({"x"}) T`. The first type requires two locks to be held, and the second requires only one lock to be held and so could be used in any situation where both locks are held. The type system conservatively prohibits this in order to prevent type-checking loopholes that would result from aliasing and side effects — that is, from having two mutable references, of different types, to the same data. See Section 7.4.2 for an example of a problem that would occur if this rule were relaxed.

Polymorphic type qualifiers

`@GuardSatisfied`(*index*) If a variable `x` has type `@GuardSatisfied`, then all lock expressions for `x`'s value are held.

As with other qualifier-polymorphism annotations (Section 24.2), the *index* argument indicates when two values are guarded by the same (unknown) set of locks.

`@GuardSatisfied` is only allowed in method signatures: on formal parameters (including the receiver) and return types. It may not be written on fields. Also, it is a limitation of the current design that `@GuardSatisfied` may not be written on array elements or on local variables.

A return type can only be annotated with `@GuardSatisfied(index)`, not `@GuardSatisfied`.

See Section 7.4.6 for an example of a use of `@GuardSatisfied`.

7.2.2 Declaration 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.

Method pre-conditions and post-conditions

`@Holding`(String[] locks) All the given lock expressions are held at the method call site.

`@EnsuresLockHeld`(String[] locks) The given lock expressions are locked upon method return if the method terminates successfully. This is useful for annotating a method that acquires a lock such as `ReentrantLock.lock()`.

`@EnsuresLockHeldIf`(String[] locks, boolean result) If the annotated method returns the given boolean value (true or false), the given lock expressions are locked upon method return if the method terminates successfully. This is useful for annotating a method that conditionally acquires a lock. See Section 7.4.4 for examples.

Side effect specifications

`@LockingFree` The method does not acquire or release locks, directly or indirectly. The method is not `synchronized`, it contains no `synchronized` blocks, it contains no calls to `lock` or `unlock` methods, and it contains no calls to methods that are not themselves `@LockingFree`.

Since `@SideEffectFree` implies `@LockingFree`, if both are applicable then you only need to write `@SideEffectFree`.

`@ReleasesNoLocks` The method maintains a strictly nondecreasing lock hold count on the current thread for any locks that were held prior to the method call. The method might acquire locks but then release them, or might acquire locks but not release them (in which case it should also be annotated with `@EnsuresLockHeld` or `@EnsuresLockHeldIf`).

This is the default for methods being type-checked that have no `@LockingFree`, `@MayReleaseLocks`, `@SideEffectFree`, or `@Pure` annotation.

`@MayReleaseLocks` The method may release locks that were held prior to the method being called. You can write this when you are certain the method releases locks, or when you don't know whether the method releases locks. This is the conservative default for methods in unannotated libraries (see Chapter 29, page 176).

7.3 Type-checking rules

In addition to the standard subtyping rules enforcing the subtyping relationship described in Figure 7.1, the Lock Checker enforces the following additional rules.

7.3.1 Polymorphic qualifiers

`@GuardSatisfied` The overall rules for polymorphic qualifiers are given in Section 24.2.

Here are additional constraints for (pseudo-)assignments:

- If the left-hand side has type `@GuardSatisfied` (with or without an index), then all locks mentioned in the right-hand side's `@GuardedBy` type must be currently held.
- A formal parameter with type qualifier `@GuardSatisfied` without an index cannot be assigned to.
- If the left-hand side is a formal parameter with type `@GuardSatisfied(`*`index`*`)`, the right-hand-side must have identical `@GuardSatisfied(`*`index`*`)` type.

If a formal parameter type is annotated with `@GuardSatisfied` without an index, then that formal parameter type is unrelated to every other type in the `@GuardedBy` hierarchy, including other occurrences of `@GuardSatisfied` without an index.

`@GuardSatisfied` may not be used on formal parameters, receivers, or return types of a method annotated with `@MayReleaseLocks`.

7.3.2 Dereferences

`@GuardedBy` An expression of type `@GuardedBy(`*`eset`*`)` may be dereferenced only if all locks in *eset* are held.

`@GuardSatisfied` An expression of type `@GuardSatisfied` may be dereferenced.

Not `@GuardedBy` or `@GuardSatisfied` An expression whose type is not annotated with `@GuardedBy` or `@GuardSatisfied` may not be dereferenced.

7.3.3 Primitive types, boxed primitive types, and Strings

Primitive types, boxed primitive types (such as `java.lang.Integer`), and type `java.lang.String` are implicitly annotated with `@GuardedBy({})`. It is an error for the programmer to annotate any of these types with an annotation from the `@GuardedBy` type hierarchy, including `@GuardedBy({})`.

7.3.4 Overriding

Overriding methods annotated with `@Holding` If class *B* overrides method *m* from class *A*, then the expressions in *B*'s `@Holding` annotation must be a subset of or equal to that of *A*'s `@Holding` annotation..

Overriding methods annotated with side effect annotations If class *B* overrides method *m* from class *A*, then the side effect annotation on *B*'s declaration of *m* must be at least as strong as that in *A*'s declaration of *m*. From weakest to strongest, the side effect annotations processed by the Lock Checker are:

```
@MayReleaseLocks
@ReleasesNoLocks
@LockingFree
@SideEffectFree
@Pure
```

7.3.5 Side effects

Releasing explicit locks Any method that releases an explicit lock must be annotated with `@MayReleaseLocks`. The Lock Checker issues a warning if it encounters a method declaration annotated with `@MayReleaseLocks` and having a formal parameter or receiver annotated with `@GuardSatisfied`. This is because the Lock Checker cannot guarantee that the guard will be satisfied throughout the body of a method if that method may release a lock.

No side effects on lock expressions If expression *expr* is used to acquire a lock, then *expr* must evaluate to the same value, starting from when *expr* is used to acquire a lock until *expr* is used to release the lock. An expression is used to acquire a lock if it is the receiver at a call site of a `synchronized` method, is the expression in a `synchronized` block, or is the argument to a `lock` method.

Locks are released after possible side effects After a call to a method annotated with `@LockingFree`, `@ReleasesNoLocks`, `@SideEffectFree`, or `@Pure`, the Lock Checker's estimate of held locks after a method call is the same as that prior to the method call. After a call to a method annotated with `@MayReleaseLocks`, the estimate of held locks is conservatively reset to the empty set, except for those locks specified to be held after the call by an `@EnsuresLockHeld` or `@EnsuresLockHeldIf` annotation on the method. Assignments to variables also cause the estimate of held locks to be conservatively reduced to a smaller set if the Checker Framework determines that the assignment might have side-effected a lock expression. For more information on side effects, please refer to Section 25.4.5.

7.4 Examples

The Lock Checker guarantees that a value that was computed from an expression of `@GuardedBy` type is dereferenced only when the current thread holds all the expressions in the `@GuardedBy` annotation.

7.4.1 Examples of @GuardedBy

The following example demonstrates the basic type-checking rules.

```
class MyClass {
  final ReentrantLock lock; // Initialized in the constructor

  @GuardedBy("lock") Object x = new Object();
  @GuardedBy("lock") Object y = x; // OK, since dereferences of y will require "lock" to be held.
  @GuardedBy({}) Object z = x; // ILLEGAL since dereferences of z don't require "lock" to be held.
  @GuardedBy("lock") Object myMethod() { // myMethod is implicitly annotated with @ReleasesNoLocks.
     return x; // OK because the return type is annotated with @GuardedBy("lock")
  }

  [...]

  void exampleMethod() {
     x.toString(); // ILLEGAL because the lock is not known to be held
     y.toString(); // ILLEGAL because the lock is not known to be held
     myMethod().toString(); // ILLEGAL because the lock is not known to be held
     lock.lock();
     x.toString();  // OK: the lock is known to be held
     y.toString();  // OK: the lock is known to be held, and toString() is annotated with @SideEffectFree.
     myMethod().toString(); // OK: the lock is known to be held, since myMethod
                            // is implicitly annotated with @ReleasesNoLocks.
  }
}
```

Note that the expression `new Object()` is inferred to have type `@GuardedBy("lock")` because it is immediately assigned to a newly-declared variable having type annotation `@GuardedBy("lock")`. You could explicitly write `new @GuardedBy("lock") Object()` but it is not required.

The following example demonstrates that using `<self>` as a lock expression allows a guarded value to be dereferenced even when the original variable name the value was originally assigned to falls out of scope.

```
class MyClass {
  private final @GuardedBy("<self>") Object x = new Object();
  void method() {
    x.toString(); // ILLEGAL because x is not known to be held.
    synchronized(x) {
      x.toString(); // OK: x is known to be held.
    }
  }

  public @GuardedBy("<self>") Object get_x() {
    return x; // OK, since the return type is @GuardedBy("<self>").
  }
}

class MyOtherClass {
  void method() {
    MyClass m = new MyClass();
    final @GuardedBy("<self>") Object o = m.get_x();
    o.toString(); // ILLEGAL because o is not known to be held.
```

```
    synchronized(o) {
      o.toString(); // OK: o is known to be held.
    }
  }
}
```

7.4.2 @GuardedBy({“a”, “b”}) is not a subtype of @GuardedBy({“a”})

@GuardedBy(exprSet)

The following example demonstrates the reason the Lock Checker enforces the following rule: if *exprSet1* $\neq$ *exprSet2*, then `@GuardedBy(exprSet1)` and `@GuardedBy(exprSet2)` are siblings in the type hierarchy.

```
class MyClass {
    final Object lockA = new Object();
    final Object lockB = new Object();
    @GuardedBy("lockA") Object x = new Object();
    @GuardedBy({"lockA", "lockB"}) Object y = new Object();
    void myMethod() {
        y = x;      // ILLEGAL; if legal, later statement x.toString() would cause trouble
        synchronized(lockA) {
          x.toString();  // dereferences y's value without holding lock lockB
        }
    }
}
```

If the Lock Checker permitted the assignment `y = x;`, then the undesired dereference would be possible.

7.4.3 Examples of @Holding

The following example shows the interaction between `@GuardedBy` and `@Holding`:

```
  void helper1(@GuardedBy("myLock") Object a) {
    a.toString(); // ILLEGAL: the lock is not held
    synchronized(myLock) {
      a.toString();  // OK: the lock is held
    }
  }
  @Holding("myLock")
  void helper2(@GuardedBy("myLock") Object b) {
    b.toString(); // OK: the lock is held
  }
  void helper3(@GuardedBy("myLock") Object d) {
    d.toString(); // ILLEGAL: the lock is not held
  }
  void myMethod2(@GuardedBy("myLock") Object e) {
    helper1(e);  // OK to pass to another routine without holding the lock
                 // (but helper1's body has an error)
    e.toString(); // ILLEGAL: the lock is not held
    synchronized (myLock) {
      helper2(e); // OK: the lock is held
      helper3(e); // OK, but helper3's body has an error
    }
  }
```

7.4.4 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 final ReentrantLock lock; // Initialized in the constructor

    ...

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

    ...
}
```

7.4.5 Example of @LockingFree, @ReleasesNoLocks, and @MayReleaseLocks

`@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`. `@ReleasesNoLocks` has a weaker guarantee than `@LockingFree`, and `@MayReleaseLocks` provides no guarantees.

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

[...]

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

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

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

@ReleasesNoLocks
void myLockingMethod() {
  lock.lock();
}

@MayReleaseLocks
void clientMethod() {
  if (lock.tryLock()) {
    x.toString(); // OK: the lock is held
    myMethod();
    x.toString(); // OK: the lock is still held since myMethod is locking-free
    mySideEffectFreeMethod();
    x.toString(); // OK: the lock is still held since mySideEffectFreeMethod is side-effect-free
    myUnlockingMethod();
    x.toString(); // ILLEGAL: myUnlockingMethod may have released a lock
  }
  if (lock.tryLock()) {
    x.toString(); // OK: the lock is held
    myLockingMethod();
    x.toString(); // OK: the lock is held
  }
  if (lock.isHeldByCurrentThread()) {
    x.toString(); // OK: the lock is known to be held
  }
}
```

7.4.6 Polymorphism and method formal parameters with unknown guards

The polymorphic `@GuardSatisfied` type annotation allows a method body to dereference the method's formal parameters even if the `@GuardedBy` annotations on the actual parameters are unknown at the method declaration site.

The declaration of `StringBuffer.append(String str)` is annotated as:

```
@LockingFree
public @GuardSatisfied(1) StringBuffer append(@GuardSatisfied(1) StringBuffer this,
                                              @GuardSatisfied(2) String str)
```

The method manipulates the values of its arguments, so all their locks must be held. However, the declaration does not know what those are and they might not even be in scope at the declaration. Therefore, the declaration cannot use `@GuardedBy` and must use `@GuardSatisfied`. The arguments to `@GuardSatisfied` indicate that the receiver and result (which are the same value) are guarded by the same (unknown, possibly empty) set of locks, and the `str` parameter may be guarded by a different set of locks.

The `@LockingFree` annotation indicates that this method makes no use of locks or synchronization.

Given these annotations on `append`, the following code type-checks:

```
final ReentrantLock lock1, lock2; // Initialized in the constructor
@GuardedBy("lock1") StringBuffer filename;
@GuardedBy("lock2") StringBuffer extension;
...
lock1.lock();
lock2.lock();
filename = filename.append(extension);
```

7.5 More locking details

This section gives some details that are helpful for understanding how Java locking and the Lock Checker works.

7.5.1 Two types of locking: monitor locks and explicit locks

Java provides two types of locking: monitor locks and explicit locks.

- A `synchronized(E)` block acquires the lock on the value of *E*; similarly, a method declared using the `synchronized` method modifier acquires the lock on the method receiver when called. (More precisely, the current thread locks the monitor associated with the value of *E*; see JLS §17.1.) The lock is automatically released when execution exits the block or the method body, respectively. We use the term "monitor lock" for a lock acquired using a `synchronized` block or `synchronized` method modifier.
- A method call, such as `Lock.lock()`, acquires a lock that implements the `Lock` interface. The lock is released by another method call, such as `Lock.unlock()`. We use the term "explicit lock" for a lock expression acquired in this way.

You should not mix the two varieties of locking, and the Lock Checker enforces this. To prevent an object from being used both as a monitor and an explicit lock, the Lock Checker issues a warning if a `synchronized(E)` block's expression `E` has a type that implements `Lock`.

7.5.2 Held locks and held expressions; aliasing

Whereas Java locking is defined in terms of values, Java programs are written in terms of expressions. We say that a lock expression is held if the value to which the expression currently evaluates is held.

The Lock Checker conservatively estimates the expressions that are held at each point in a program. The Lock Checker does not track aliasing (different expressions that evaluate to the same value); it only considers the exact expression used to acquire a lock to be held. After any statement that might side-effect a held expression or a lock expression, the Lock Checker conservatively considers the expression to be no longer held.

Section 25.5 explains which Java expressions the Lock Checker is able to analyze as lock expressions.

The `@LockHeld` and `@LockPossiblyHeld` type qualifiers are used internally by the Lock Checker and should never be written by the programmer. If you see a warning mentioning `@LockHeld` or `@LockPossiblyHeld`, please contact the Checker Framework developers as it is likely to indicate a bug in the Checker Framework.

7.5.3 Run-time checks for locking

When you perform a run-time check for locking, such as `if (explicitLock.isHeldByCurrentThread()){...}` or `if (Thread.holdsLock(monitorLock)){...}`, then the Lock Checker considers the lock expression to be held within the scope of the test. For more details, see Section 25.4.

7.5.4 Discussion of default qualifier

The default qualifier for unannotated types is `@GuardedBy({})`. This default forces you to write explicit `@GuardSatisfied` in method signatures in the common case that clients ensure that all locks are held.

It might seem that `@GuardSatisfied` would be a better default for method signatures, but such a default would require even more annotations. The reason is that `@GuardSatisfied` cannot be used on fields. If `@GuardedBy({})` is the default for fields but `@GuardSatisfied` is the default for parameters and return types, then getters, setters, and many other types of methods do not type-check without explicit lock qualifiers.

7.5.5 Discussion of `@Holding`

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

Synchronization protocol `@Holding` can specify a synchronization protocol that is not expressible as locks over the parameters to a method. For example, a global lock or a lock on a different object might need to be held. By requiring locks to be held, you can create 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 not already acquired.

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

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

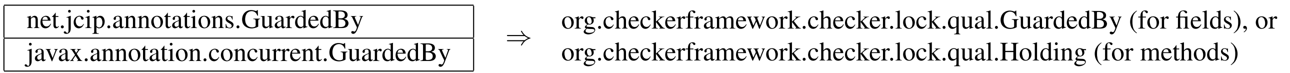
```
@Holding("myLock")
void myMethod() {
  ...
}
```

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

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

It is better to write a `@Holding` annotation rather than writing the extra synchronized block. Here are reasons:

- The annotation documents the fact that the lock is intended to already be held; that is, the method's contract requires that the lock be held when the method is called.
- 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 version with a 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. The lock re-acquisition consumes time, even if it succeeds.



net.jcip.annotations.GuardedBy	⇒	org.checkerframework.checker.lock.qual.GuardedBy (for fields), or
javax.annotation.concurrent.GuardedBy		org.checkerframework.checker.lock.qual.Holding (for methods)

Figure 7.2: Correspondence between other lock annotations and the Checker Framework's annotations.

7.6 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, the Checker Framework can type-check your code. It treats annotations from other tools as if you had written the corresponding annotation from the Lock Checker, as described in Figure 7.2. If the other annotation is a declaration annotation, it may be moved; see Section 27.1.1.

7.6.1 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 occur 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.). Another important distinction is that the Lock Checker's annotations express and enforce a locking discipline over values, just like the JLS expresses Java's locking semantics; by contrast, JCIP's annotations express a locking discipline that protects variable names and does not prevent race conditions. 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" and `@GuardedBy` creates preconditions for doing so. Variables can be accessed by reading or writing them (putfield, getfield), and methods can be accessed by calling them (invokevirtual, invokeinterface). This informal intuition is inappropriate for a tool that requires precise semantics.)

7.7 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.

Chapter 8

Index Checker for sequence bounds (arrays and strings)

The Index Checker warns about potentially out-of-bounds accesses to sequence data structures, such as arrays and strings.

The Index Checker prevents `IndexOutOfBoundsException`s that result from an index expression that might be negative or might be equal to or larger than the sequence's length. It also prevents `NegativeArraySizeException`s that result from a negative array dimension in an array creation expression. (A caveat: the Index Checker does not check for arithmetic overflow. If an expression overflows, the Index Checker might fail to warn about a possible exception. This is unlikely to be a problem in practice unless you have an array whose length is `Integer.MAX_VALUE`.)

The programmer can write annotations that indicate which expressions are indices for which sequences. The Index Checker prohibits any operation that may violate these properties, and the Index Checker takes advantage of these properties when verifying indexing operations. Typically, a programmer writes few annotations, because the Index Checker infers properties of indexes from the code around them. For example, it will infer that `x` is positive within the `then` block of an `if (x > 0)` statement. The programmer does need to write field types and method pre-conditions or post-conditions. For instance, if a method's formal parameter is used as an index for `myArray`, the programmer might need to write an `@IndexFor("myArray")` annotation on the formal parameter's types.

The Index Checker checks fixed-size data structures, whose size is never changed after creation. A fixed-size data structure has no `add` or `remove` operation. Examples are strings and arrays, and you can add support for other fixed-size data structures (see Section 8.9).

To run the Index Checker, run the command

```
javac -processor index MyJavaFile.java
```

Recall that in Java, type annotations are written before the type; in particular, array annotations appear immediately before "`[]`". Here is how to declare a length-9 array of positive integers:

```
@Positive int @ArrayLen(9) []
```

Multi-dimensional arrays are similar. Here is how to declare a length-2 array of length-4 arrays:

```
String @ArrayLen(2) [] @ArrayLen(4) []
```

8.1 Index Checker structure and annotations

Internally, the Index Checker computes information about integers that might be indices:

- the lower bound on an integer, such as whether it is known to be positive (Section 8.2)

- the upper bound on an integer, such as whether it is less than the length of a given sequence (Section 8.3)
- whether an integer came from calling the JDK's binary search routine on an array (Section 8.6)
- whether an integer came from calling a string search routine (Section 8.7)

and about sequence lengths:

- the minimum length of a sequence, such "`myArray` contains at least 3 elements" (Section 8.4)
- whether two sequences have the same length (Section 8.5)

The Index Checker checks of all these properties at once, but this manual discusses each type system in a different section. There are some annotations that are shorthand for writing multiple annotations, each from a different type system:

`@IndexFor(String[] names)` The value is a valid index for the named sequences. For example, the `String.charAt(int)` method is declared as

```
class String {
  char charAt(@IndexFor("this") index) { ... }
}
```

More generally, a variable declared as `@IndexFor("someArray") int i` has type `@IndexFor("someArray") int` and its run-time value is guaranteed to be non-negative and less than the length of `someArray`. You could also express this as `@NonNegative @LTLengthOf("someArray") int i`, but `@IndexFor("someArray") int i` is more concise.

`@IndexOrHigh(String[] names)` The value is non-negative and is less than or equal to the length of each named sequence. This type combines `@NonNegative` and `@LTEqLengthOf`.
For example, the `Arrays.fill` method is declared as

```
class Arrays {
  void fill(Object[] a, @IndexFor("#1") int fromIndex, @IndexOrHigh("#1") int toIndex, Object val)
}
```

`@LengthOf(String[] names)` The value is exactly equal to the length of the named sequences. In the implementation, this type aliases `@IndexOrHigh`, so writing it only adds documentation (although future versions of the Index Checker may use it to improve precision).

`@IndexOrLow(String[] names)` The value is -1 or is a valid index for each named sequence. This type combines `@GTENegativeOne` and `@LTLengthOf`.

`@PolyIndex` indicates qualifier polymorphism. This type combines `@PolyLowerBound` and `@PolyUpperBound`. For a description of qualifier polymorphism, see Section 24.2.

`@PolyLength` is a special polymorphic qualifier that combines `@PolySameLen` and `@PolyValue` from the Constant Value Checker (see Chapter 19, page 119). `@PolyLength` exists as a shorthand for these two annotations, since they often appear together.

8.2 Lower bounds

The Index Checker issues an error when a sequence is indexed by an integer that might be negative. The Lower Bound Checker uses a type system (Figure 8.1) with the following qualifiers:

`@Positive` The value is 1 or greater, so it is not too low to be used as an index. Note that this annotation is trusted by the Constant Value Checker, so if the Constant Value Checker is run on code containing this annotation, the Lower Bound Checker must be run on the same code in order to guarantee soundness.

`@NonNegative` The value is 0 or greater, so it is not too low to be used as an index.

`@GTENegativeOne` The value is -1 or greater. It may not be used as an index for a sequence, because it might be too low. ("`GTE`" stands for "Greater Than or Equal to".)

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

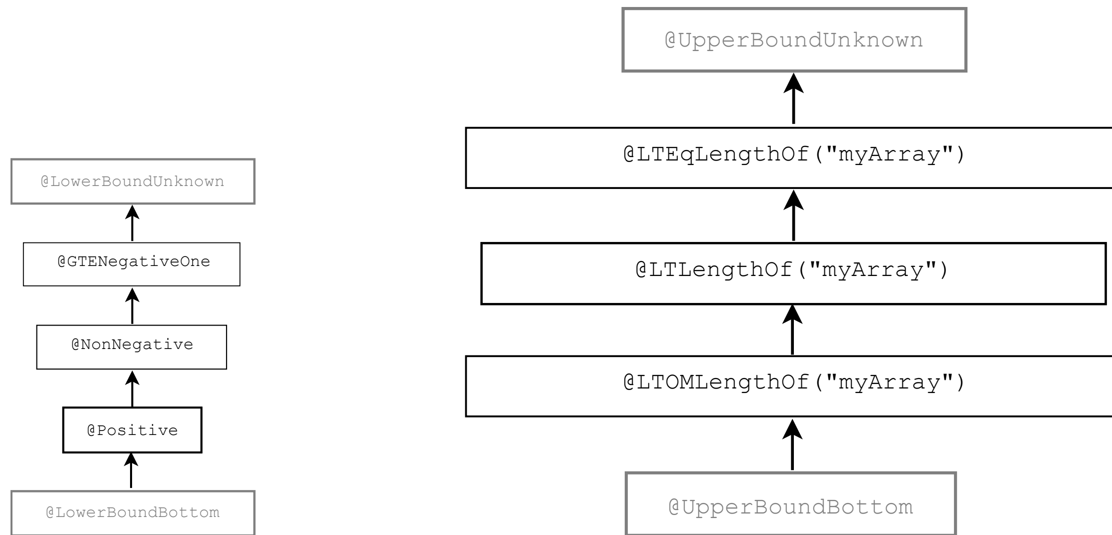


Figure 8.1: The two type hierarchies for integer types used by the Index Checker. On the left is a type system for lower bounds. On the right is a type system for upper bounds. Qualifiers written in gray should never be written in source code; they are used internally by the type system. `"myArray"` in the Upper Bound qualifiers is an example of an array's name.

`@LowerBoundUnknown` There is no information about the value. It may not be used as an index for a sequence, because it might be too low.

`@LowerBoundBottom` The value cannot take on any integral types. The bottom type, which should not need to be written by the programmer.

8.3 Upper bounds

The Index Checker issues an error when a sequence index might be too high. To do this, it maintains information about which expressions are safe indices for which sequences. The length of a sequence is `arr.length` for arrays and `str.length()` for strings. It uses a type system (Figure 8.1) with the following qualifiers:

It issues an error when a sequence `arr` is indexed by an integer that is not of type `@LTLengthOf("arr")` or `@LTOMLengthOf("arr")`.

`@LTLengthOf(String[] names, String[] offset)` An expression with this type has value less than the length of each sequence listed in `names`. The expression may be used as an index into any of those sequences, if it is non-negative. For example, an expression of type `@LTLengthOf("a") int` might be used as an index to `a`. The type `@LTLengthOf({"a", "b"})` is a subtype of both `@LTLengthOf("a")` and `@LTLengthOf("b")`. ("`LT`" stands for "Less Than".)
`@LTLengthOf` takes an optional `offset` element, meaning that the annotated expression plus the offset is less than the length of the given sequence. For example, suppose expression `e` has type `@LTLengthOf(value = {"a", "b"}, offset = {"-1", "x"})`. Then `e - 1` is less than `a.length`, and `e + x` is less than `b.length`. This helps to make the checker more precise. Programmers rarely need to write the `offset` element.

`@LTEqLengthOf(String[] names)` An expression with this type has value less than or equal to the length of each sequence listed in `names`. It may not be used as an index for these sequences, because it might be too high. `@LTEqLengthOf({"a", "b"})` is a subtype of both `@LTEqLengthOf("a")` and `@LTEqLengthOf("b")`. ("`LTEq`" stands for "Less Than or Equal to".)

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

@LTOMLengthOf(String[] names) An expression with this type has value at least 2 less than the length of each sequence listed in `names`. It may always used as an index for a sequence listed in `names`, if it is non-negative. This type exists to allow the checker to infer the safety of loops of the form:

```
for (int i = 0; i < array.length - 1; ++i) {
  arr[i] = arr[i+1];
}
```

This annotation should rarely (if ever) be written by the programmer; usually `@LTLengthOf(String[] names)` should be written instead. `@LTOMLengthOf({"a", "b"})` is a subtype of both `@LTOMLengthOf("a")` and `@LTOMLengthOf("b")`. ("LTOM" stands for "Less Than One Minus", because another way of saying "at least 2 less than `a.length`" is "less than `a.length-1`".)

@UpperBoundUnknown There is no information about the upper bound on the value of an expression with this type. It may not be used as an index for a sequence, because it might be too high. This type is the top type, and should never need to be written by the programmer.

@UpperBoundBottom This is the bottom type for the upper bound type system. It should never need to be written by the programmer.

The following method annotations can be used to establish a postcondition of a method which ensures that a certain expression is a valid index for a sequence:

@EnsuresLTLengthOf(String[] value, String[] targetValue, String[] offset) When the method with this annotation returns, the expression (or all the expressions) given in the `value` element is less than the length of the given sequences with the given offsets. More preciesly, the expression has the `@LTLengthOf` qualifier with the `value` and `offset` arguments taken from the `targetValue` and `offset` elements of this annotation.

@EnsuresLTLengthOfIf(String[] expression, boolean result, String[] targetValue, String[] o
If the method with this annotation returns the given boolean value, then the given expression (or all the given expressions) is less than the length of the given sequences with the given offsets.

8.4 Sequence minimum lengths

The Index Checker estimates, for each sequence expression, how long its value might be at run time by computing a minimum length that the sequence is guaranteed to have. This enables the Index Checker to verify indices that are compile-time constants. For example, this code:

```
String getThirdElement(String[] arr) {
  return arr[2];
}
```

is legal if `arr` has at least three elements, which can be indicated in this way:

```
String getThirdElement(String @MinLen(3) [] arr) {
  return arr[2];
}
```

When the index is not a compile-time constant, as in `arr[i]`, then the Index Checker depends not on a `@MinLen` annotation but on `i` being annotated as `@LTLengthOf("arr")`.

The MinLen type qualifier is implemented in practice by the Constant Value Checker, using `@ArrayLenRange` annotations (see Chapter 19, page 119). This means that errors related to the minimum lengths of arrays must be suppressed using the "value" argument to `@SuppressWarnings`. `@ArrayLenRange` and `@ArrayLen` annotations can also be used to establish the minimum length of a sequence, if a more precise estimate of length is known. For example, if `arr` is known to have exactly three elements:

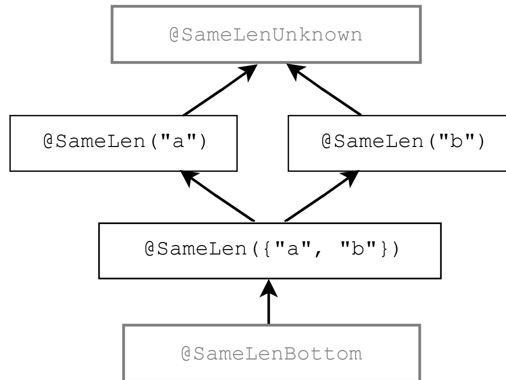


Figure 8.2: The type hierarchy for arrays of equal length ("a" and "b" are assumed to be in-scope sequences). Qualifiers written in gray should never be written in source code; they are used internally by the type system.

```
String getThirdElement(String @ArrayLen(3) [] arr) {
  return arr[2];
}
```

The following type qualifiers (from the Chapter 19, page 119) can establish the minimum length of a sequence:

@MinLen(int value) The value of an expression of this type is a sequence with at least `value` elements. The default annotation is `@MinLen(0)`, and it may be applied to non-sequences. `@MinLen(`x`)` is a subtype of `@MinLen(`$x-1$`)`. An `@MinLen` annotation is treated internally as an `@ArrayLenRange` with only its `from` field filled.

@ArrayLen(int[] value) The value of an expression of this type is a sequence whose length is exactly one of the integers listed in its argument. The argument can contain at most ten integers; larger collections of integers are converted to `@ArrayLenRange` annotations. The minimum length of a sequence with this annotation is the smallest element of the argument.

@ArrayLenRange(int from, int to) The value of an expression of this type is a sequence whose length is bounded by its arguments, inclusive. The minimum length of a sequence with this annotation is its `from` argument.

The following method annotation can be used to establish a postcondition of a method which ensures that a certain sequence has a minimum length:

@EnsuresMinLenIf(String[] expression, boolean result, int targetValue) If the method with this annotation returns the given boolean value, then the given expression (or all the given expressions) is a sequence with at least `targetValue` elements.

8.5 Sequences of the same length

The Index Checker determines whether two or more sequences have the same length. This enables it to verify that all the indexing operations are safe in code like the following:

```
boolean lessThan(double[] arr1, double @SameLen("#1") [] arr2) {
  for (int i = 0; i < arr1.length; i++) {
    if (arr1[i] < arr2[i]) {
      return true;
    } else if (arr1[i] > arr2[i]) {
```

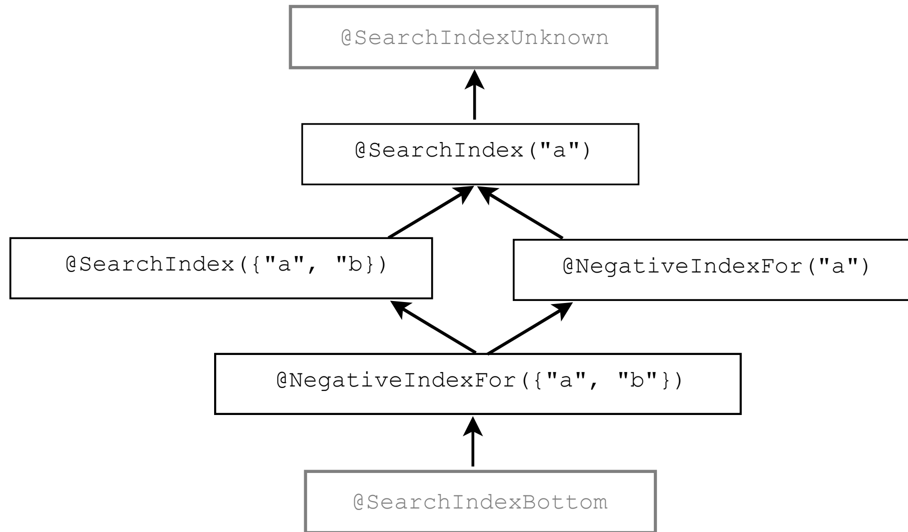


Figure 8.3: The type hierarchy for the Index Checker's internal type system that captures information about the results of calls to `Arrays.binarySearch`.

```
        return false;
      }
    }
    return false;
  }
```

When needed, you can specify which sequences have the same length using the following type qualifiers (Figure 8.2):

@SameLen(String[] names) An expression with this type represents a sequence that has the same length as the other sequences named in `names`. In general, `@SameLen` types that have non-intersecting sets of names are *not* subtypes of each other. However, if at least one sequence is named by both types, the types are actually the same, because all the named sequences must have the same length.

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

@SameLenUnknown No information is known about which other sequences have the same length as this one. This is the top type, and programmers should never need to write it.

@SameLenBottom This is the bottom type, and programmers should rarely need to write it. `null` has this type.

8.6 Binary search indices

The JDK's `Arrays.binarySearch` method returns either where the value was found, or a negative value indicating where the value could be inserted. The Search Index Checker represents this concept.

The Search Index Checker's type hierarchy (Figure 8.3) has four type qualifiers:

@SearchIndexFor(String[] names) An expression with this type represents an integer that could have been produced by calling `Arrays.binarySearch`: for each array `a` specifed in the annotation, the annotated integer is between `-a.length-1` and `a.length-1`, inclusive

@NegativeIndexFor(String[] names) An expression with this type represents a "negative index" that is between `a.length-1` and `-1`, inclusive; that is, a value that is both a `@SearchIndex` and is negative. Applying the bitwise complement operator (~) to an expression of this type produces an expression of type `@IndexOrHigh`.

@SearchIndexBottom This is the bottom type, and programmers should rarely need to write it.

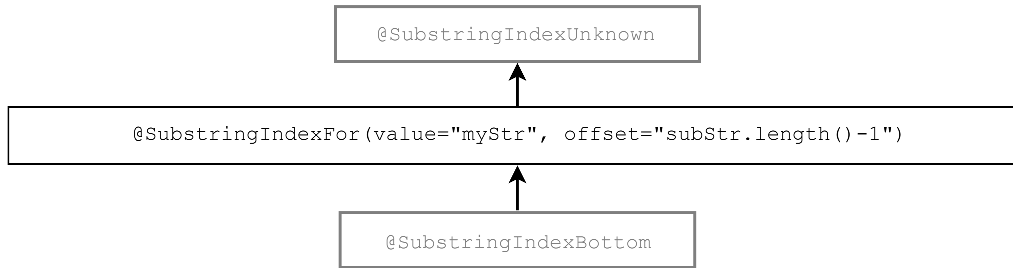


Figure 8.4: The type hierarchy for the Substring Index Checker, which captures information about the results of calls to `String.indexOf` and `String.lastIndexOf`.

`@SearchIndexUnknown` No information is known about whether this integer is a search index. This is the top type, and programmers should rarely need to write it.

8.7 Substring indices

The methods `String.indexOf` and `String.lastIndexOf` return an index of a given substring within a given string, or -1 if no such substring exists. The index `i` returned from `receiver.indexOf(substring)` satisfies the following property, which is stated here in three equivalent ways:

```
i == -1 || ( i >= 0       && i <= receiver.length() - substring.length()                 )
i == -1 || ( @NonNegative && @LTLengthOf(value="receiver", offset="substring.length()-1") )
@SubstringIndexFor(value="receiver", offset="substring.length()-1")
```

The return type of methods `String.indexOf` and `String.lastIndexOf` has the annotation `@SubstringIndexFor(value="this", offset="#1.length()-1"))`. This allows writing code such as the following with no warnings from the Index Checker:

```
public static String removeSubstring(String original, String removed) {
  int i = original.indexOf(removed);
  if (i != -1) {
    return original.substring(0, i) + original.substring(i + removed.length());
  }
  return original;
}
```

The `@SubstringIndexFor` annotation is implemented in a Substring Index Checker that runs together with the Index Checker and has its own type hierarchy (Figure 8.4) with three type qualifiers:

`@SubstringIndexFor(String[] value, String[] offset)` An expression with this type represents an integer that could have been produced by calling `String.indexOf`: the annotated integer is either -1, or it is non-negative and is less than or equal to `receiver.length - offset` (where the sequence `receiver` and the offset `offset` are corresponding elements of the annotation's arguments).

`@SubstringIndexBottom` This is the bottom type, and programmers should rarely need to write it.

`@SubstringIndexUnknown` No information is known about whether this integer is a substring index. This is the top type, and programmers should rarely need to write it.

8.7.1 The need for the `@SubstringIndexFor` annotation

No other annotation supported by the Index Checker precisely represents the possible return values of methods `String.indexOf` and `String.lastIndexOf`. The reason is the methods' special cases for empty strings and for failed

matches.

Consider the result `i` of `receiver.indexOf(substring)`:

- `i` is `@GTENegativeOne`, because `i >= -1`.
- `i` is `@LTEqLengthOf("receiver")`, because `i <= receiver.length()`.
- `i` is not `@IndexOrLow("receiver")`, because for `receiver = "", substring = "", i = 0`, the property `i >= -1 && i < receiver.length()` does not hold.
- `i` is not `@IndexOrHigh("receiver")`, because for `receiver = "", substring = "b", i = -1`, the property `i >= 0 && i <= receiver.length()` does not hold.
- `i` is not `@LTLengthOf(value = "receiver", offset = "substring.length()-1")`, because for `receiver = "", substring = "abc", i = -1`, the property `i + substring.length() - 1 < receiver.length()` does not hold.

The last annotation in the list above, `@LTLengthOf(value = "receiver", offset = "substring.length()-1")`, is the correct and precise upper bound for all values of `i` except -1. The offset expresses the fact that we can add `substring.length()` to this index and still get a valid index for `receiver`. That is useful for type-checking code that adds the length of the substring to the found index, in order to obtain the rest of the string. However, the upper bound applies only after the index is explicitly checked not to be -1:

```
int i = receiver.indexOf(substring);
// i is @GTENegativeOne and @LTEqLengthOf("receiver")
// i is not @LTLengthOf(value = "receiver", offset = "substring.length()-1")
if (i != -1) {
  // i is @NonNegative and @LTLengthOf(value = "receiver", offset = "substring.length()-1")
  int j = i + substring.length();
  // j is @IndexOrHigh("receiver")
  return receiver.substring(j); // this call is safe
}
```

The property of the result of `indexOf` cannot be expressed by any combination of lower-bound (Section 8.2) and upper-bound (Section 8.3) annotations, because the upper-bound annotations apply independently of the lower-bound annotations, but in this case, the upper bound `i <= receiver.length() - substring.length()` holds only if `i >= 0`. Therefore, to express this property and make the example type-check without false positives, a new annotation such as `@SubstringIndexFor(value = "receiver", offset = "substring.length()-1")` is necessary.

8.8 Inequalities

The Index Checker estimates which expression's values are less than other expressions' values.

`@LessThan(String[] values)` An expression with this type has a value that is less than the value of each expression listed in `values`. The expressions in values must be composed of final or effectively final variables and constants.

`@LessThanUnknown` There is no information about the value of an expression this type relative to other expressions. This is the top type, and should not be written by the programer.

`@LessThanBottom` This is the bottom type for the less than type system. It should never need to be written by the programmer.

8.9 Annotating fixed-size data structures

The Index Checker has built-in support for Strings and arrays. You can add support for additional fixed-size data structures by writing annotations. This allows the Index Checker to typecheck the data structure's implementation and to typecheck uses of the class.

This section gives an example: a fixed-length collection.

```
/** ArrayWrapper is a fixed-size generic collection. */
public class ArrayWrapper<T> {
    private final Object @SameLen("this") [] delegate;

    @SuppressWarnings("index") // constructor creates object of size @SameLen(this) by definition
    ArrayWrapper(@NonNegative int size) {
        delegate = new Object[size];
    }

    public @LengthOf("this") int size() {
        return delegate.length;
    }

    public void set(@IndexFor("this") int index, T obj) {
        delegate[index] = obj;
    }

    @SuppressWarnings("unchecked") // required for normal Java compilation due to unchecked cast
    public T get(@IndexFor("this") int index) {
        return (T) delegate[index];
    }
}
```

The Index Checker treats methods annotated with `@LengthOf("this")` as the length of a sequence like `arr.length` for arrays and `str.length()` for strings.

With these annotations, client code like the following typechecks with no warnings:

```
public static void clearIndex1(ArrayWrapper<? extends Object> a, @IndexFor("#1") int i) {
    a.set(i, null);
}

public static void clearIndex2(ArrayWrapper<? extends Object> a, int i) {
    if (0 <= i && i < a.size()) {
        a.set(i, null);
    }
}
```

Chapter 9

Fake Enum Checker for fake enumerations

The Fake Enum Checker, or Fenum Checker, enables you to define a type alias or typedef, in which two different sets of values have the same representation (the same Java type) but are not allowed to be used interchangeably. It is also possible to create a typedef using the Subtyping Checker (Chapter 22, page 132), and that approach is sometimes more appropriate.

One common use for the Fake Enum Checker is the *fake enumeration pattern* (Section 9.6). For example, consider this code adapted from Android's `IntDef` documentation:

```
@NavigationMode int NAVIGATION_MODE_STANDARD = 0;
@NavigationMode int NAVIGATION_MODE_LIST = 1;
@NavigationMode int NAVIGATION_MODE_TABS = 2;

@NavigationMode int getNavigationMode();

void setNavigationMode(@NavigationMode int mode);
```

The Fake Enum Checker can issue a compile-time warning if the programmer ever tries to call `setNavigationMode` with an `int` that is not a `@NavigationMode int`.

The Fake Enum Checker gives the same safety guarantees as a true enumeration type or typedef, but retaining backward-compatibility with interfaces that use existing Java types. You can apply fenum annotations to any Java type, including all primitive types and also reference types. Thus, you could use it (for example) to represent floating-point values between 0 and 1, or `String`s with some particular characteristic. (Note that the Fake Enum Checker does not let you create a shorter alias for a long type name, as a real typedef would if Java supported it.)

As explained in Section 9.1, you can either define your own fenum annotations, such as `@NavigationMode` above, or you can use the existing `@Fenum` with a string argument. Figure 9.1 shows part of the type hierarchy for the Fenum type system.

9.1 Fake enum annotations

The Fake Enum Checker supports two ways to introduce a new fake enum (fenum):

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

   ```
   package myPackage.qual;

   import java.lang.annotation.Documented;
   import java.lang.annotation.ElementType;
   import java.lang.annotation.Retention;
   import java.lang.annotation.RetentionPolicy;
   ```

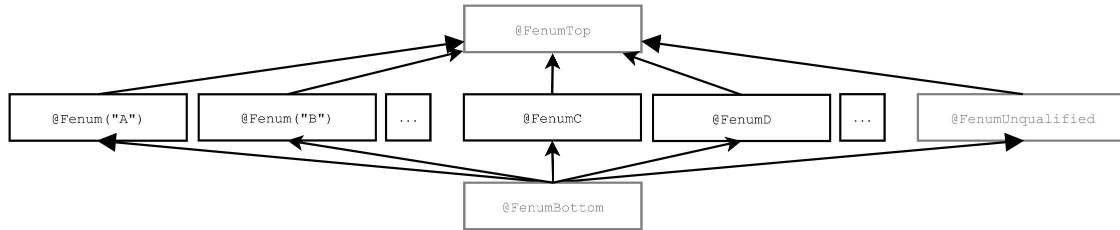


Figure 9.1: Partial type hierarchy for the Fenum type system. There are two forms of fake enumeration annotations — above, illustrated by `@Fenum("A")` and `@FenumC`. See Section 9.1 for descriptions of how to introduce both types of fenums. The type qualifiers in gray (`@FenumTop`, `@FenumUnqualified`, and `@FenumBottom`) should never be written in source code; they are used internally by the type system. `@FenumUnqualified` is the default qualifier for unannotated types, except for upper bounds which default to `@FenumTop`.

```
import java.lang.annotation.Target;
import org.checkerframework.checker.fenum.qual.FenumTop;
import org.checkerframework.framework.qual.SubtypeOf;

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})
@SubtypeOf(FenumTop.class)
public @interface MyFenum {}
```

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

Note that all custom annotations must have the `@Target({ElementType.TYPE_USE})` meta-annotation. See section 30.4.1.

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

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

9.2 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 from all types with a different fenum annotation or `String` argument.

The checker 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 25) can prevent failure to initialize a reference variable.)

9.3 Running the Fenum Checker

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

- If you define your own annotation(s), provide the name(s) of the annotation(s) through the `-Aquals` option, using a comma-no-space-separated notation:

```
javac -classpath /full/path/to/myProject/bin:/full/path/to/myLibrary/bin \
      -processor org.checkerframework.checker.fenum.FenumChecker \
      -Aquals=myPackage.qual.MyFenum MyFile.java ...
```

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 classpath) before you run the Fenum Checker with `javac`. It is not sufficient to supply their source files on the command line.

You can also provide the fully-qualified paths to a set of directories that contain the annotations through the `-AqualDirs` option, using a colon-no-space-separated notation. For example:

```
javac -classpath /full/path/to/myProject/bin:/full/path/to/myLibrary/bin \
      -processor org.checkerframework.checker.fenum.FenumChecker \
      -AqualDirs=/full/path/to/myProject/bin:/full/path/to/myLibrary/bin MyFile.java ...
```

Note that in these two examples, the compiled class file of the `myPackage.qual.MyFenum` annotation must exist in either the `myProject/bin` directory or the `myLibrary/bin` directory. The following placement of the class file will work with the above commands:

```
.../myProject/bin/myPackage/qual/MyFenum.class
```

The two options can be used at the same time to provide groups of annotations from directories, and individually named annotations.

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

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

For an example of running the Fake Enum Checker on Android code, see `https://github.com/karlicoss/checker-fenum-android-demo`.

9.4 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;
}
```

9.5 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
  }
}
```

Also, see the example project in the `docs/examples/fenum-extension` directory.

9.6 The fake enumeration pattern

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, and Android also uses `int` constants rather than Java enums. 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.

9.7 References

- Case studies of the Fake Enum Checker:
  "Building and using pluggable type-checkers" [DDE+11] (ICSE 2011, `http://homes.cs.washington.edu/~mernst/pubs/pluggable-checkers-icse2011.pdf#page=3`)
- 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:
  `https://www.javaworld.com/article/2077487/core-java/java-tip-122--beware-of-java-typesafe-enumeration html`

Chapter 10

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.

10.1 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 tainted (untrusted) or untainted (trusted) values. `@Tainted` is a supertype of `@Untainted`. It is the default qualifier.
- `@PolyTainted` is a qualifier that is polymorphic over tainting (see Section 24.2).

10.2 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 29 describes how to annotate external libraries):

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

10.3 `@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 has two consequences. The first consequence is that the annotations can serve many different purposes, such as:

- 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.

The second consequence is that the Tainting Checker is not useful unless you annotate the appropriate sources, sinks, and untainting/sanitization routines.

If you want more specialized semantics, or you want to annotate multiple types of tainting (for example, HTML and SQL) 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. You will change the copy to rename the annotations, and you will annotate libraries and/or your code to identify sources, sinks, and validation/sanitization routines. See Chapter 30 for more details.

Chapter 11

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.

11.1 Regex annotations

These qualifiers make up the Regex type system:

`@Regex` indicates that the run-time value is a valid regular expression `String`. If the optional parameter is supplied to the qualifier, then the number of capturing groups in the regular expression is at least that many. If not provided, the parameter defaults to 0. For example, if an expression's type is `@Regex(1) String`, then its run-time value could be `"colo(u?)r"` or `"(brown|beige)"` but not `"colou?r"` nor a non-regex string such as `"1) first point"`.

`@PolyRegex` indicates qualifier polymorphism (see Section 24.2).

The subtyping hierarchy of the Regex Checker's qualifiers is shown in Figure 11.1.

11.2 Annotating your code with `@Regex`

11.2.1 Implicit qualifiers

The Regex Checker adds implicit qualifiers, reducing the number of annotations that must appear in your code (see Section 25.3). The checker implicitly adds the `Regex` qualifier with the parameter set to the correct number of capturing

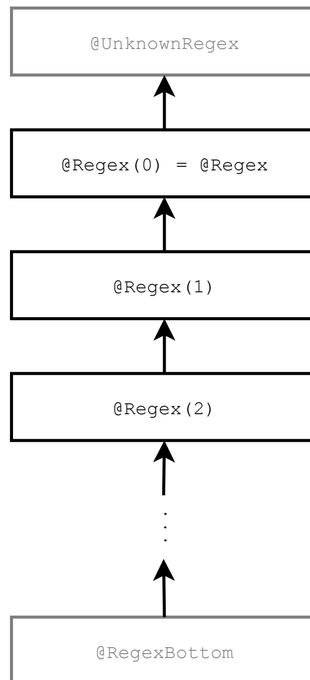


Figure 11.1: The subtyping relationship of the Regex Checker's qualifiers. Because the parameter to a `@Regex` qualifier is at least the number of capturing groups in a regular expression, a `@Regex` qualifier with more capturing groups is a subtype of a `@Regex` qualifier with fewer capturing groups. 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 11.2: An example of the Regex Checker's support for concatenation of non-regular-expression Strings to produce valid regular expression Strings.

---

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.

11.2.2 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.

11.2.3 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 11.2.

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

Figure 11.3: Example use of `RegexUtil` methods.

11.2.4 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 11.3 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. That is, the `checker-qual.jar` file must be on the classpath at run time. You can avoid this by copying the `RegexUtil` class into your own code.

11.2.5 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 11.3 should support most cases so the `asRegex` method should be used rarely.

Chapter 12

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 12.3 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.

12.1 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.

12.2 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 12.2.1. 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.

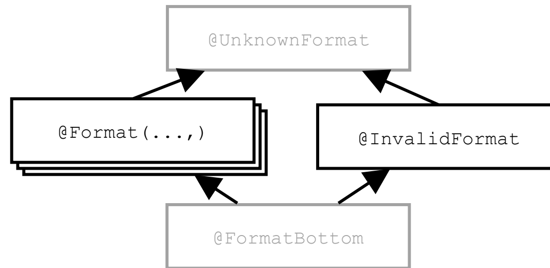


Figure 12.1: The Format String Checker type qualifier hierarchy. The figure does not show the subtyping rules among different `@Format(...)` qualifiers; see Section 12.2.2.

```
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 12.1 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. `@UnknownFormat` is the default that is applied to strings that are not literals and on which the user has not written a `@Format` annotation.

12.2.1 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 12.3.2.) 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 12.2 summarizes the subset relationship among all conversion categories.

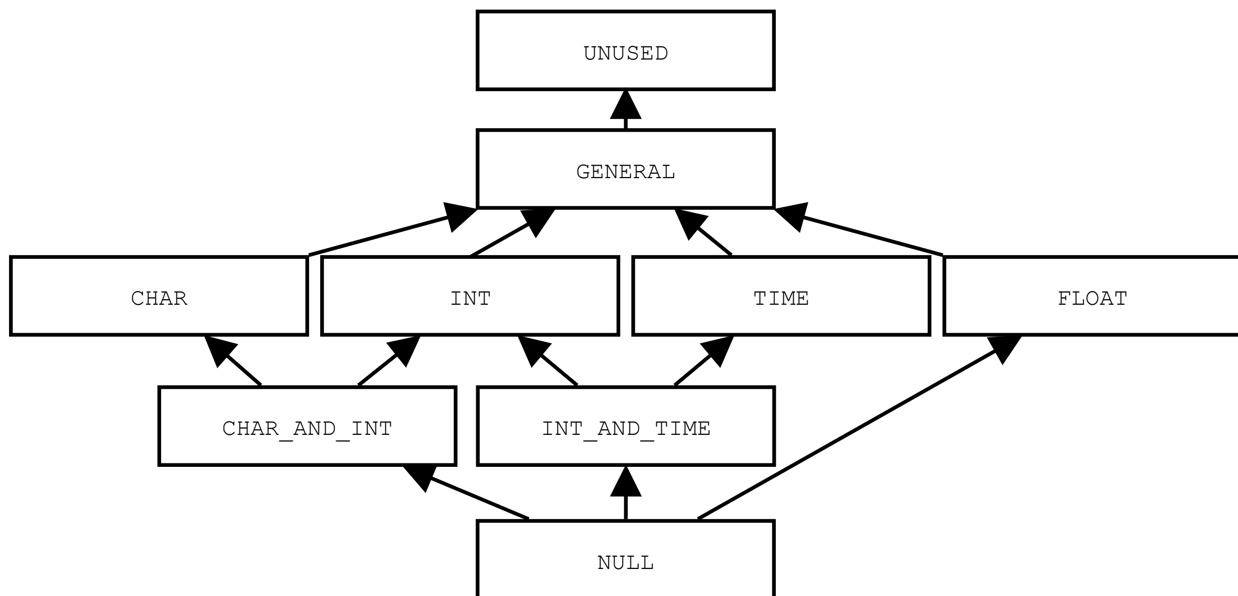


Figure 12.2: The subset relationship among conversion categories.

12.2.2 Subtyping rules for **@Format**

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. Although this is legal a warning is issued because most occurrences of this are due to programmer error.

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);
```

12.3 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 an 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)
```

12.3.1 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.

12.3.2 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)
```

12.4 Implicit qualifiers

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

12.5 @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);
    }
}
```

You should annotate such a method with the `@FormatMethod` annotation, which indicates that the `String` argument is a format string for the remaining arguments.

12.6 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 13

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 13.6.

The Internationalization Checker or I18n Checker (Chapter 14.2, page 102) 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.

13.1 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 13.1 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 13.2.

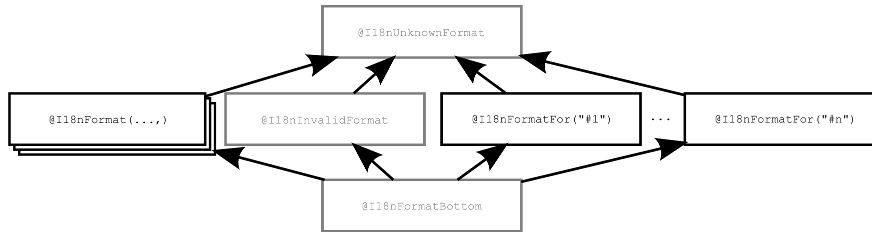


Figure 13.1: The Internationalization Format String Checker type qualifier hierarchy. The figure does not show the subtyping rules among different `@I18nFormat(...)` qualifiers; see Section 13.2. All `@I18nFormatFor` annotations are unrelated by subtyping. The qualifiers in gray are used internally by 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.

13.2 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 13.2 summarizes the subset relationship among all conversion categories.

13.3 Subtyping rules for `@I18nFormat`

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. Although this is legal a warning is issued because most occurrences of this are due to programmer error.

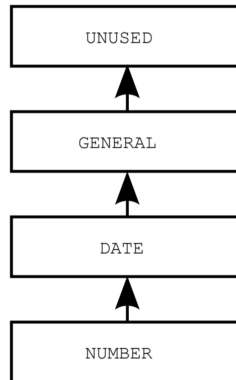


Figure 13.2: The subset relationship among i18n conversion categories.

The following example shows the subtyping rules in action:

```
@I18nFormat({NUMBER, DATE}) String f;

f = "{0, number, #.#} {1, date}"; // OK
f = "{0, number} {1}";            // OK, GENERAL is weaker (less restrictive) than DATE
f = "{0} {1, date}";              // OK, GENERAL is weaker (less restrictive) than NUMBER
f = "{0, number}";                // warning: last argument is ignored
f = "{0}";                        // warning: last argument is ignored
f = "{0, number} {1, number}";    // error: NUMBER is stronger (more restrictive) than DATE
f = "{0} {1} {2}";                // error: too many arguments
```

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`.

13.4 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}";
```

13.5 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 14, page 101.

13.6 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`

13.7 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 13.8.

13.8 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) {...}

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

// 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}";
s1 = "{0, number}";        // error: incompatible types in assignment
```

Chapter 14

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 14.1, page 101) 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 14.2, page 102) verifies that code is properly internationalized. A Compiler Message Key Checker (Section 14.3, page 102) 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.

14.1 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 `ResourceBundle`s, 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 ':'.

14.2 Internationalization Checker (I18n 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 13). 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.

14.2.1 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.

14.2.2 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.

14.3 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 15

Signature String 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 15.1), 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.

15.1 Signature annotations

Java defines six formats for the string representation of a type. There is an annotation for each of these representations. Figure 15.1 shows how they are related; examples appear in a table below.

`@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 conceptual name 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. It uses internal form (binary names, but with `/` instead of `.`; see JVMS §4.2) for class names. See examples below.

`@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 "`/`". See examples below.

`@InternalForm` The *internal form* (JVMS §4.2), such as `package/Outer$Inner`, is how a class name is actually represented in its own `.class` file. It is also known as the "syntax of binary names that appear in class file structures". It is the same as the binary name, but with periods (`.`) replaced by slashes (`/`). Programmers more often use the binary name, leaving the internal form as a JVM implementation detail.

`@ClassGetSimpleName` The type representation returned by the `Class.getSimpleName()` method. This format is not required by any method in the JDK, so you will rarely write it in source code. The string can be empty. This is not the same as the "simple name" defined in (JLS §6.2), which is the same as `@Identifier`.

Other type qualifiers are the intersection of two or more qualifiers listed above; for example, a `@SourceNameForNonInner` is a string that is a valid fully qualified name *and* a valid binary name. A programmer should never or rarely use these qualifiers, and you can ignore them as implementation details of the Signature Checker, though you might occasionally

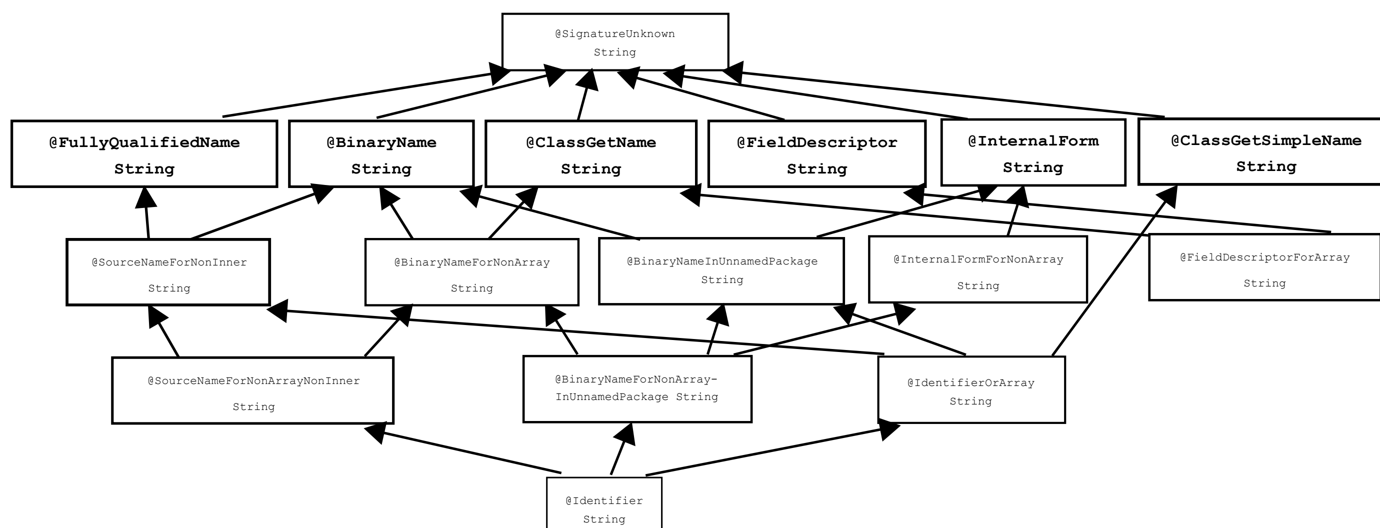


Figure 15.1: Partial type hierarchy for the Signature type system, showing string representations of a Java type. Programmers only need to write the boldfaced qualifiers, in the second row; qualifiers below those are included to improve the internal handling of String literals.

see them in an error message. These qualifiers exist to give literals sufficiently precise types that they can be used in any appropriate context.

Java also defines other string formats for a type: 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:

fully-qualified name	binary name	Class.getName	field descriptor	internal form	Class.getSimpleName
int	int	int	I	int	int
int[][]	int[][]	[[I	[[I	int[][]	int[][]
MyClass	MyClass	MyClass	LMyClass;	MyClass	MyClass
MyClass[]	MyClass[]	[LMyClass;	[LMyClass;	MyClass[]	MyClass[]
n/a for anonymous class	MyClass$22	MyClass$22	LMyClass$22;	MyClass$22	*(empty string)*
n/a for array of anon. class	MyClass$22[]	[LMyClass$22;	[LMyClass$22;	MyClass$22[]	[]
java.lang.Integer	java.lang.Integer	java.lang.Integer	Ljava/lang/Integer;	java/lang/Integer	Integer
java.lang.Integer[]	java.lang.Integer[]	[Ljava.lang.Integer;	[Ljava/lang/Integer;	java/lang/Integer[]	Integer[]
pkg.Outer.Inner	pkg.Outer$Inner	pkg.Outer$Inner	Lpkg/Outer$Inner;	pkg/Outer$Inner	Inner
pkg.Outer.Inner[]	pkg.Outer$Inner[]	[Lpkg.Outer$Inner;	[Lpkg/Outer$Inner;	pkg/Outer$Inner[]	Inner[]
n/a for anonymous class	pkg.Outer$22	pkg.Outer$22	Lpkg/Outer$22;	pkg/Outer$22	*(empty string)*
n/a for array of anon. class	pkg.Outer$22[]	[Lpkg.Outer$22;	[Lpkg/Outer$22;	pkg/Outer$22[]	[]

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;
```

15.2 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 16

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 16.1),
- class or package annotations controlling the default effect (Section 16.4), and
- *effect-polymorphism*: code that works for both the safe effect and the UI effect (Section 16.5).

16.1 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

`@SafeEffect` ≺ `@UIEffect`

16.2 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).

16.3 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 ...
```

16.4 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 16.5.2.)
- `@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.

16.5 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;`.

16.5.1 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.

16.5.2 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.

16.5.3 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.)

16.5.4 Subtyping with polymorphic effects

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

`@SafeEffect` ≺ `@PolyUIEffect` ≺ `@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:

`@AlwaysSafe` ≺ `@PolyUI` ≺ `@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.

Effect polymorphism and arguments

Sometimes it is useful to have `@PolyUI` parameters on a method. As a trivial example, this permits us to specify an identity method that works for both `@UI Runnable` and `@AlwaysSafe Runnable`:

```
public @PolyUI Runnable id(@PolyUI Runnable r) {
    return r;
}
```

This use of `@PolyUI` will be handled as is standard for polymorphic qualifiers in the Checker Framework (see Section 24.2).

`@PolyUIEffect` methods should generally not use `@PolyUI` arguments: it is permitted by the framework, but its interaction with inheritance is subtle, and may not behave as you would hope.

The shortest explanation is this: `@PolyUI` arguments may only be overridden by `@PolyUI` arguments, even though the implicitly `@PolyUI` *receiver* may be overridden with a `@AlwaysSafe` receiver.

As noted earlier (Section 24.1.6), Java's generics are invariant — `A<X>` is a subtype of `B<Y>` only if `X` is identical to `Y`. Class-level use of `@PolyUI` behaves slightly differently. Marking a type declaration `@PolyUIType` is conceptually equivalent to parameterizing the type by some `E extends Effect`. But in this view, `Runnable<SafeEffect>` (really `@AlwaysSafe Runnable`) would be considered a subtype of `Runnable<UIEffect>` (really `@UI Runnable`), as explained earlier in this section. Java's generics do not permit this, which is called *covariant* subtyping in the effect parameter. Permitting it for all generics leads to problems where a type system can miss errors. Java solves this by making all generics invariant, which rejects more programs than strictly necessary, but leads to an easy-to-explain limitation. For this checker, covariant subtyping of effect parameters is very important: being able to pass an `@AlwaysSafe Runnable` in place of a `@UI Runnable` is extremely useful. Since we need to allow some cases for flexibility, but need to reject other cases to avoid missing errors, the distinction is a bit more subtle for this checker.

Consider this small example (warning: the following is rejected by the GUI Effect Checker):

```
@PolyUIType
public interface Dispatcher {
    @PolyUIEffect
    void dispatch(@PolyUI Runnable toRun);
}
```

```
@SafeType
public class SafeDispatcher {
    @Override
    public void dispatch(@AlwaysSafe Runnable toRun) {
        runOnBackgroundThread(toRun);
    }
}
```

This may initially seem like harmless code to write, which simply specializes the implicit effect parameter from `Dispatcher` in the `SafeDispatcher` implementation. However, the way method effect polymorphism is implemented is by implicitly making the receiver of a `@PolyUIEffect` method — the object on which the method is invoked — `@PolyUI`. So if the definitions above were permitted, the following client code would be possible:

```
@AlwaysSafe SafeDispatcher s = ...;
@UI Runnable uitask = ...;
s.dispatch(uitask);
```

At the call to `dispatch`, the Checker Framework is free to consider `s` as its supertype, `@UI SafeDispatcher`. This permits the framework to choose the same qualifier for both the (implicit) receiver use of `@PolyUI` and the `toRun` argument to `Dispatcher.dispatch`, passing the checker. But this code would then pass a UI-thread-only task to a method which should only accept background thread tasks — exactly what the checker should prevent!

To resolve this, the GUI Effect Checker rejects the definitions above, specifically the `@AlwaysSafe` on `SafeDispatcher.dispatch`'s parameter, which would need to remain `@PolyUI`.

A subtlety of the code above is that the receiver for `SafeDispatcher.dispatch` is also overridden, switching from a `@PolyUI` receiver to a `@Safe` receiver. That change is permissible. But when that is done, the polymorphic arguments may no longer be interchangeable with the receiver.

16.6 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 17

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.

17.1 Units annotations

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

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:

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

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

1. For `@Acceleration`: Meter Per Second Square `@mPERs2`
2. For `@Angle`: Radians `@radians`, and the derived unit Degrees `@degrees`
3. For `@Area`: the derived units square millimeters `@mm2`, square meters `@m2`, and square kilometers `@km2`
4. For `@Current`: Ampere `@A`
5. For `@Length`: Meters `@m` and the derived units millimeters `@mm` and kilometers `@km`
6. For `@Luminance`: Candela `@cd`
7. For `@Mass`: kilograms `@kg` and the derived unit grams `@g`
8. For `@Speed`: meters per second `@mPERs` and kilometers per hour `@kmPERh`
9. For `@Substance`: Mole `@mol`
10. For `@Temperature`: Kelvin `@K` and the derived unit Celsius `@C`
11. 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.

The polymorphic annotation `@PolyUnit` enables you to write a method that takes an argument of any unit type and returns a result of that same type. For more about polymorphic qualifiers, see Section 24.2. For an example of its use, see the `@PolyUnit` Javadoc.

17.2 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)
@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.

Note that all custom annotations must have the `@Target(ElementType.TYPE_USE)` meta-annotation. See section 30.4.1.

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

17.3 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.)

17.4 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 fully-qualified class names of the annotations through the `-Aunits` option, using a comma-no-space-separated notation:

  ```
  javac -classpath /full/path/to/myProject/bin:/full/path/to/myLibrary/bin \
        -processor org.checkerframework.checker.units.UnitsChecker \
        -Aunits=myPackage.qual.MyUnit,myPackage.qual.MyOtherUnit MyFile.java ...
  ```

  The annotations listed in `-Aunits` 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 classpath) before you run the Units Checker with `javac`. It is not sufficient to supply their source files on the command line.

- You can also provide the fully-qualified paths to a set of directories that contain units qualifiers through the `-AunitsDirs` option, using a colon-no-space-separated notation. For example:

  ```
  javac -classpath /full/path/to/myProject/bin:/full/path/to/myLibrary/bin \
        -processor org.checkerframework.checker.units.UnitsChecker \
        -AunitsDirs=/full/path/to/myProject/bin:/full/path/to/myLibrary/bin MyFile.java ...
  ```

  Note that in these two examples, the compiled class file of the `myPackage.qual.MyUnit` and `myPackage.qual.MyOtherUnit` annotations must exist in either the `myProject/bin` directory or the `myLibrary/bin` directory. The following placement of the class files will work with the above commands:

  ```
  .../myProject/bin/myPackage/qual/MyUnit.class
  .../myProject/bin/myPackage/qual/MyOtherUnit.class
  ```

  The two options can be used at the same time to provide groups of annotations from directories, and individually named annotations.

Also, see the example project in the `docs/examples/units-extension` directory.

17.5 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`.

17.6 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 18

Signedness Checker

The Signedness Checker guarantees that signed and unsigned values are not mixed together in a computation. In addition, it prohibits meaningless operations, such as division on an unsigned value.

18.1 Annotations

The Signedness Checker uses type annotations to indicate the signedness that the programmer intends an expression to have.

These are the qualifiers in the signedness type system:

`@Unsigned` indicates that the programmer intends the value to be interpreted as unsigned. That is, if the most significant bit in the bitwise representation is set, then the bits should be interpreted as a large positive value.

`@Signed` indicates that the programmer intends the value to be interpreted as signed. That is, if the most significant bit in the bitwise representation is set, then the bits should be interpreted as a negative value. This is the default annotation.

`@Constant` indicates that a value is a compile-time constant and could be interpreted as unsigned or signed. This annotation is used internally, and should not be written by the programmer.

`@UnknownSignedness` indicates that a value's type is not relevant or known to this checker. This annotation is used internally, and should not be written by the programmer.

`@SignednessBottom` indicates that the value is `null`. This annotation is used internally, and should not be written by the programmer.

Signedness is primarily about how the bits of the representation are interpreted, not about the values that it can represent. An unsigned value is always positive, but just because a variable's value is positive does not mean that it should be marked as `@Unsigned`. If variable *v* will be compared to a signed value, or used in arithmetic operations with a signed value, then *v* should have signed type.

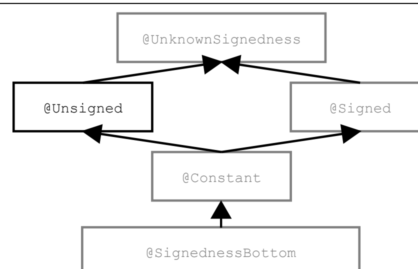


Figure 18.1: The type qualifier hierarchy of the signedness annotations. Qualifiers in gray are used internally by the type system but should never be written by a programmer.

18.1.1 Default qualifiers

The only type qualifier that the programmer should need to write is `@Unsigned`. When a programmer leaves an expression unannotated, the Signedness Checker treats it in one of the following ways:

- All `byte`, `short`, `int`, and `long` literals default to `@Constant`.
- All `byte`, `short`, `int`, and `long` variables default to `@Signed`.
- All other expressions default to `@UnknownSignedness`.

18.2 Prohibited operations

The Signedness Checker prohibits the following uses of operators:

- Division (`/`) or modulus (`%`) with an `@Unsigned` operand.
- Signed right shift (`>>`) with an `@Unsigned` left operand.
- Unsigned right shift (`>>>`) with a `@Signed` left operand.
- Greater/less than (or equal) comparators (`<`, `<=`, `>`, `>=`) with an `@Unsigned` operand.
- Any other binary operator with one `@Unsigned` operand and one `@Signed` operand, with the exception of left shift (`<<`).

Like every type-checker built with the Checker Framework, the Signedness Checker ensures that assignments and pseudo-assignments have consistent types. For example, it is not permitted to assign a `@Signed` expression to an `@Unsigned` variable or vice versa.

18.3 Rationale

The Signedness Checker prevents misuse of unsigned values in Java code. Most Java operations interpret operands as signed. If applied to unsigned values, those operations would produce unexpected, incorrect results.

Consider the following Java code:

```
int s1 = -1;
int s2 = -2;

@Unsigned int u1 = 0xFFFFFFFF;  // unsigned: 2^32 - 1, signed: -1
@Unsigned int u2 = 0xFFFFFFFE;  // unsigned: 2^32 - 2, signed: -2

int result;

result = s2 / s1;   // OK: result is 2, which is correct for -2 / -1
result = u2 / u1;   // ERROR: result is 2, which is incorrect for (2^32 - 1) / (2^32 - 2)

int s3 = -1;
int s4 = 5;

@Unsigned int u3 = 0xFFFFFFFF;    // unsigned: 2^32 - 1, signed: -1
@Unsigned int u4 = 5;

result = s3 % s4;    // OK: result is 4, which is correct for -2 % 5
result = u3 % u4;    // ERROR: result is 4, which is incorrect for (2^32 - 1) % 5
```

These examples illustrate why division and modulus with an unsigned operand are illegal. Other uses of operators are prohibited for similar reasons.

18.4 Utility routines for manipulating unsigned values

Class `SignednessUtil` provides static utility methods for working with unsigned values. Some of these re-implement functionality in JDK 8, making it available in earlier versions of Java. Others provide new functionality. All of them are properly annotated with `@Unsigned` where appropriate, so using them may reduce the number of annotations that you need to write.

Class `SignednessUtilExtra` contains more utility methods that reference packages not included in Android. This class is not included in `checker-qual.jar`, so you may want to copy the methods to your code.

Chapter 19

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.
- Alternately, you can run just the Constant Value Checker, by supplying the following command-line options to javac: `-processor org.checkerframework.common.value.ValueChecker -Astubs=statically-executable.astub`

19.1 Annotations

The Constant Value Checker uses type annotations to indicate the value of an expression (Section 19.1.1), and it uses method annotations to indicate methods that the Constant Value Checker can execute at compile time (Section 19.1.3).

19.1.1 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 main type annotations are `@BoolVal`, `@IntVal`, `@IntRange`, `@DoubleVal`, and `@StringVal`. Additional type annotations for arrays and strings are `@ArrayLen`, `@ArrayLenRange`, and `@MinLen`. A polymorphic qualifier (`@PolyValue`) is also supported (see Section 24.2).

Each `*Val` 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 `@IntRange` for integral types, `@ArrayLenRange` for array types, `@ArrayLen` or `@ArrayLenRange` for `String`, and `@UnknownVal` for all other types. The `@ArrayLen` annotation means that at run time, the expression evaluates to an array or a string whose length is one of the annotation's arguments.

In the case of too many strings in `@StringVal`, the values are forgotten and just the lengths are used in `@ArrayLen`. If this would result in too many lengths, only the minimum and maximum lengths are used in `@ArrayLenRange`, giving a range of possible lengths of the string.

The `@StringVal` annotation may be applied to a char array. Although byte arrays are often converted to/from strings, the `@StringVal` annotation may not be applied to them. This is because the conversion depends on the platform's character set.

`@IntRange` takes two arguments — a lower bound and an upper bound. Its meaning is that at run time, the expression evaluates to a value between the bounds (inclusive). For example, an expression of type `@IntRange(from=0, to=255)` evaluates to 0, 1, 2, ..., 254, or 255. An `@IntVal` and `@IntRange` annotation that represent the same set of values are semantically identical and interchangeable: they have exactly the same meaning, and using either one has the

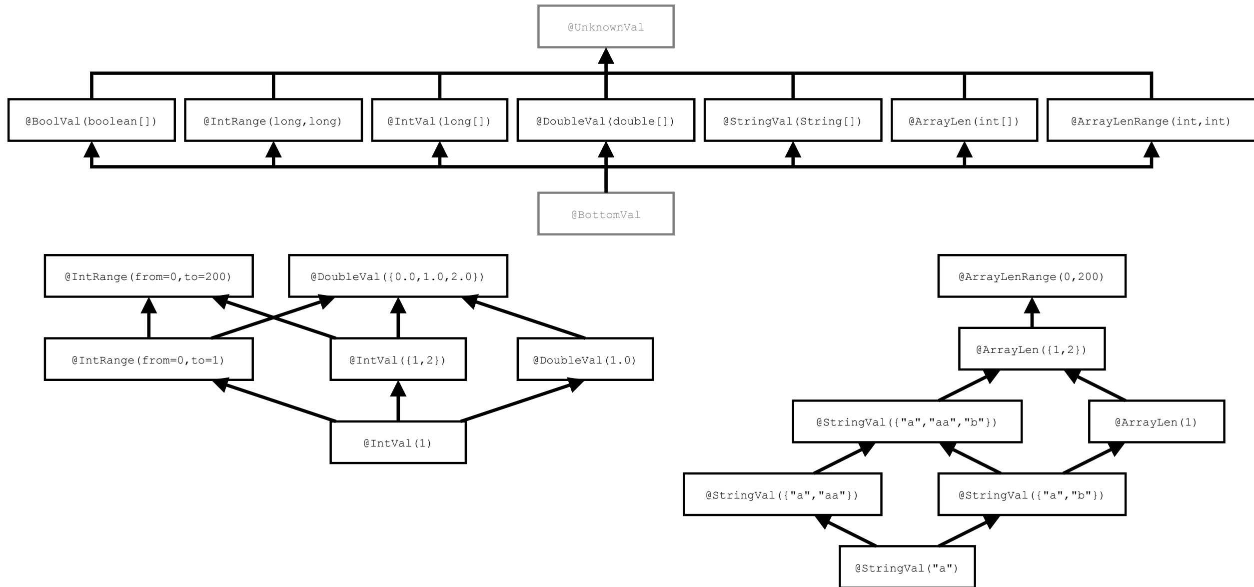


Figure 19.1: At the top, the type qualifier hierarchy of the Constant Value Checker annotations. Qualifiers in gray are used internally by the type system but should never be written by a programmer. At the bottom are examples of additional subtyping relationships that depend on the annotations' arguments.

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

same effect. `@ArrayLenRange` has the same relationship to `@ArrayLen` that `@IntRange` has to `@IntVal`. The `@MinLen` annotation is an alias for `@ArrayLenRange` (meaning that every `@MinLen` annotation is automatically converted to an `@ArrayLenRange` annotation) that only takes one argument, which is the lower bound of the range. The upper bound of the range is the maximum integer value.

Figure 19.1 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 19.2 illustrates how the Constant Value Checker infers type annotations (using flow-sensitive type qualifier refinement, Section 25.4).

If your code is already annotated with a different constant value or range annotation, the Checker Framework can type-check your code. It treats annotations from other tools as if you had written the corresponding annotation from the Constant Value Checker, as described in Figure 19.3. If the other annotation is a declaration annotation, it may be moved; see Section 27.1.1.

The Constant Value Checker trusts the `@Positive`, `@NonNegative`, and `@GTENegativeOne` annotations. If your code contains any of these annotations, then in order to guarantee soundness, you must run the Index Checker whenever you run the Constant Value Checker.

android.support.annotation.IntRange	⇒ org.checkerframework.checker.common.value.qual.IntRange

Figure 19.3: Correspondence between other constant value and range annotations and the Checker Framework's annotations.

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

19.1.2 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.

The Constant Value Checker statically executes operators that do not throw exceptions (e.g., `+`, `-`, `<<`, `!=`).

19.1.3 `@StaticallyExecutable` methods and the classpath

The Constant Value Checker statically executes methods annotated with `@StaticallyExecutable`, *if the method has already been compiled and is on the classpath*.

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. 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. For example, the command-line arguments to the Checker Framework might include:

```
-processor org.checkerframework.common.value.ValueChecker
-Astubs=statically-executable.astub
-classpath $CLASSPATH:MY_PROJECT/build/
```

19.2 Warnings

If the option `-AreportEvalWarns` options is used, 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].`

```
...
if (i > 5) {
  // i now has type:  @IntRange(from=5, to=Integer.MAX_VALUE)
  i = i + 1;
  // If i started out as Integer.MAX_VALUE, then i is now Integer.MIN_VALUE.
  // i's type is now @IntRange(from=Integer.MIN_VALUE, to=Integer.MAX_VALUE).
  // When ignoring overflow, i's type is now @IntRange(from=6, to=Integer.MAX_VALUE).
}
```

Figure 19.5: With the `-AignoreRangeOverflow` command-line option, the Constant Value Checker ignores overflow for range types, which gives smaller ranges to range types.

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 method that was annotated as `@StaticallyExecutable`.

- `[method.evaluation.exception] Failed to evaluate method public static int Test.foo(int) because it threw an exception:  java.lang.ArithmeticException:  / by zero.`
  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 are some other situations in which the Constant Value Checker produces a warning message:

- `[too.many.values.given] The maximum number of arguments permitted 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 replaced with `@IntRange`, `@ArrayLen`, `@ArrayLenRange`, or `@UnknownVal`.

19.3 Unsoundly ignoring overflow

The Constant Value Checker takes Java's overflow rules into account when computing the possible values of expressions. The `-AignoreRangeOverflow` command-line option makes it ignore the possibility of overflow for range annotations `@IntRange` and `@ArrayLenRange`. Figure 19.5 gives an example of behavior with and without the `-AignoreRangeOverflow` command-line option.

As with any unsound behavior in the Checker Framework, this option reduces the number of warnings and errors produced, and may reduce the number of `@IntRange` qualifiers that you need to write in the source code. However, it is possible that at run time, an expression might evaluate to a value that is not in its `@IntRange` qualifier. You should either accept that possibility, or verify the lack of overflow using some other tool or manual analysis.

Chapter 20

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.

20.1 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`):

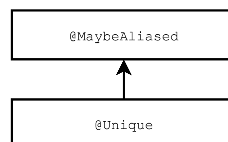


Figure 20.1: Type hierarchy for the Aliasing type system.

@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.

20.2 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 20.4.)

```
@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.
}
```

20.3 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>`.

20.4 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 20.2: 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 20.2 gives an example of the Aliasing Checker's type refinement rules.

Chapter 21

Reflection resolution

A call to `Method.invoke` might reflectively invoke any method. That method might place requirements on its formal parameters, and it might return any value. To reflect these facts, 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 19) 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` 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:

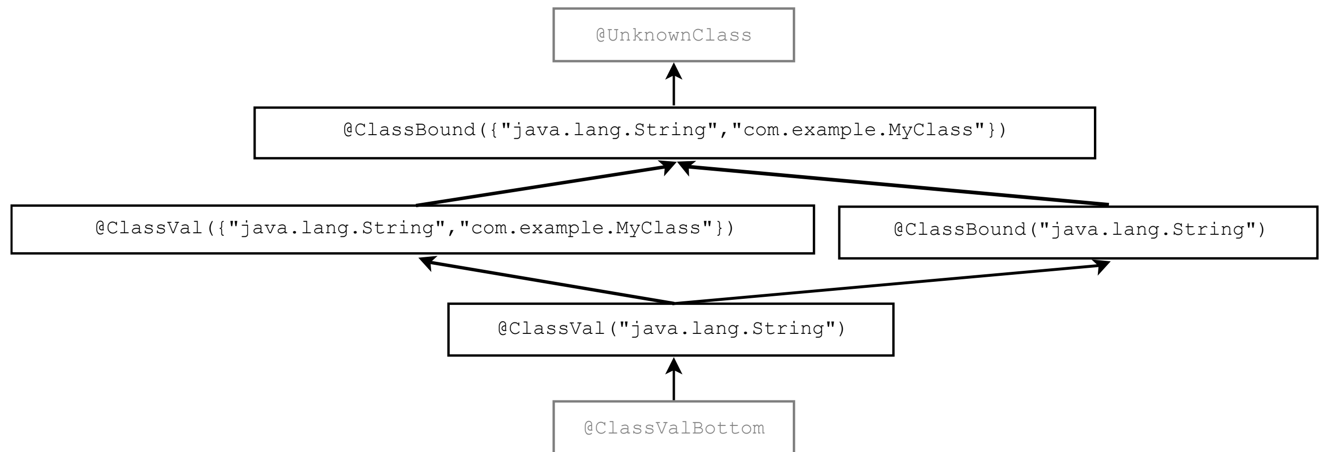


Figure 21.1: Partial type hierarchy for the ClassVal type system. The type qualifiers in gray (`@UnknownClass` and `@ClassValBottom`) should never be written in source code; they are used internally by the type system.

`@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})
```

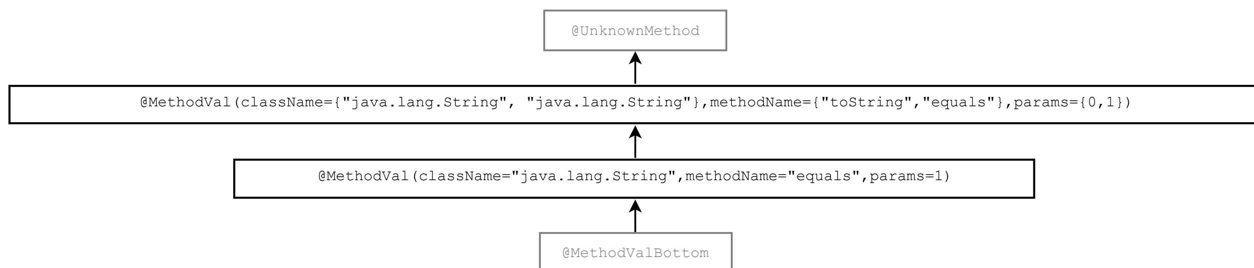


Figure 21.2: Partial type hierarchy for the MethodVal type system. The type qualifiers in gray (`@UnknownMethod` and `@MethodValBottom`) should never be written in source code; they are used internally by the type system.

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.

Subtyping rules

Figure 21.2 shows 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.

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

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

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

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

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

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.

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 30). 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 32.7.1 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 30.

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. One of the two following arguments must be used with the Subtyping Checker:

- Provide the fully-qualified class name(s) of the annotation(s) in the custom type system through the `-Aquals` option, using a comma-no-space-separated notation:

  ```
  javac -classpath /full/path/to/myProject/bin:/full/path/to/myLibrary/bin \
        -processor org.checkerframework.common.subtyping.SubtypingChecker \
        -Aquals=myPackage.qual.MyQual,myPackage.qual.OtherQual MyFile.java ...
  ```

  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 classpath) before you run the Subtyping Checker with `javac`. It is not sufficient to supply their source files on the command line.
- Provide the fully-qualified paths to a set of directories that contain the annotations in the custom type system through the `-AqualDirs` option, using a colon-no-space-separated notation. For example:

  ```
  javac -classpath /full/path/to/myProject/bin:/full/path/to/myLibrary/bin \
        -processor org.checkerframework.common.subtyping.SubtypingChecker \
        -AqualDirs=/full/path/to/myProject/bin:/full/path/to/myLibrary/bin MyFile.java
  ```

Note that in these two examples, the compiled class file of the `myPackage.qual.MyQual` and `myPackage.qual.OtherQual` annotations must exist in either the `myProject/bin` directory or the `myLibrary/bin` directory. The following placement of the class files will work with the above commands:

```
.../myProject/bin/myPackage/qual/MyQual.class
.../myLibrary/bin/myPackage/qual/OtherQual.class
```

The two options can be used at the same time to provide groups of annotations from directories, and individually named annotations.

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 myPackage.qual;

import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

/**
 * Denotes that the representation of an object is encrypted.
 */
@SubtypeOf(PossiblyUnencrypted.class)
@ImplicitFor(literal={LiteralKind.NULL})
@DefaultFor({TypeUseLocation.LOWER_BOUND})
@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})
public @interface Encrypted {}
```

```
package myPackage.qual;

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

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

Note that all custom annotations must have the `@Target(ElementType.TYPE_USE)` meta-annotation. See Section 30.4.1.

Don't forget to compile these classes:

```
$ javac myPackage/qual/Encrypted.java myPackage/qual/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 myPackage.qual.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

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

4. You can also provide the fully-qualified paths to a set of directories that contain the qualifiers using the `-AqualDirs` option, and add the directories to the boot classpath, for example:

```
  javac -classpath /full/path/to/myProject/bin:/full/path/to/myLibrary/bin \
        -processor org.checkerframework.common.subtyping.SubtypingChecker \
        -AqualDirs=/full/path/to/myProject/bin:/full/path/to/myLibrary/bin YourProgram.java
```

Note that in these two examples, the compiled class file of the `myPackage.qual.Encrypted` and `myPackage.qual.PossiblyUnencrypted` annotations must exist in either the `myProject/bin` directory or the `myLibrary/bin` directory. The following placement of the class files will work with the above commands:

```
  .../myProject/bin/myPackage/qual/Encrypted.class
  .../myProject/bin/myPackage/qual/PossiblyUnencrypted.class
```

Also, see the example project in the `docs/examples/subtyping-extension` directory.

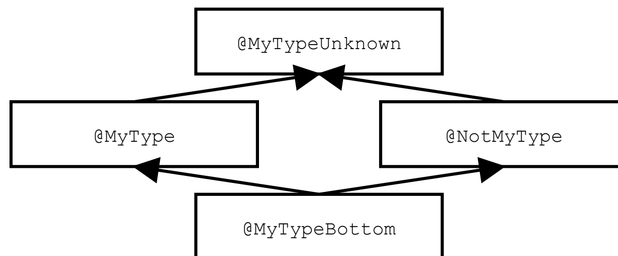


Figure 22.1: Type system for a type alias or typedef type system. The type system designer may choose to omit some of these types, but this is the general case. The type system designer's choice of defaults affects the interpretation of unannotated code, which affects the guarantees given for unannotated code.

22.3 Type aliases and typedefs

A type alias or typedef is a type that shares the same representation as another type but is conceptually distinct from it. For example, some strings in your program may be street addresses; others may be passwords; and so on. You wish to indicate, for each string, which one it is, and to avoid mixing up the different types of strings. Likewise, you could distinguish integers that are offsets from those that are absolute values.

Creating a new type makes your code easier to understand by conveying the intended use of each variable. It also prevents errors that come from using the wrong type or from mixing incompatible types in an operation.

If you want to create a type alias or typedef, you have multiple options: a regular Java subtype, the Units Checker (Chapter 17, page 112), the Fake Enum Checker (Chapter 9, page 79), or the Subtyping Checker.

A Java subtype is easy to create and does not require a tool such as the Checker Framework; for instance, you would declare `class Address extends String`. There are a number of limitations to this "pseudo-typedef", however [Goe06]. Primitive types and final types (including `String`) cannot be extended. Equality and identity tests can return incorrect results when a wrapper object is used. Existing return types in code would need to be changed, which is easy with an annotation but disruptive to change the Java type. Therefore, it is best to avoid the pseudo-typedef antipattern.

The Units Checker (Chapter 17, page 112) is useful for the particular case of units of measurement, such as kilometers verses miles.

The Fake Enum Checker (Chapter 9, page 79) builds in a set of assumptions. If those fit your use case, then it's easiest to use the Fake Enum Checker (though you can achieve them using the Subtyping Checker). The Fake Enum Checker forbids mixing of fenums of different types, or fenums and unannotated types. For instance, binary operations other than string concatenations are forbidden, such as `NORTH+1`, `NORTH+MONDAY`, and `NORTH==MONDAY`. However, `NORTH+SOUTH` is permitted.

By default, the Subtyping Checker does not forbid any operations.

If you choose to use the Subtyping Checker, then you have an additional design choice to make about the type system. In the general case, your type system will look something like Figure 22.1.

References whose type is `@MyType` are known to store only values from your new type. There is no such guarantee for `@MyTypeUnknown` and `@NotMyType`, but those types mean different things. An expression of type `@NotMyType` is guaranteed never to evaluate to a value of your new type. An expression of type `@MyTypeUnknown` may evaluate to any value — including values of your new type and values not of your new type. (`@MyTypeBottom` is the type of `null` and is also used for dead code and erroneous situations; it can be ignored for this discussion.)

A key choice for the type system designer is which type is the default. That is, if a programmer does not write `@MyType` on a given type use, should that type use be interpreted as `@MyTypeUnknown` or as `@NotMyType`?

- If unannotated types are interpreted as `@NotMyType`, then the type system enforces very strong separation between your new type and all other types. Values of your type will never mix with values of other types. If you don't see `@MyType` written explicitly on a type, you will know that it does not contain values of your type.
- If unannotated types are interpreted as `@MyTypeUnknown`, then a generic, unannotated type may contain a value of your new type. In this case, `@NotMyType` does not need to exist, and `@MyTypeBottom` may or may not exist in

your type system.

A downside of the stronger guarantee that comes from using `@NotMyType` as the default is the need to write additional annotations. For example, if `@NotMyType` is the default, this code does not typecheck:

```
void method(Object o) { ... }
<U> void use(List<U> list) {
  method(list.get(0));
}
```

Because (implicit) upper bounds are interpreted as the top type (see Section 24.1.2), this is interpreted as

```
void method(@NotMyType Object o) { ... }
<@U extends @MyTypeUnknown Object> void use(List<U> list) {
  // type error: list.get(0) has type @MyTypeUnknown, method expects @NotMyType
  method(list.get(0));
}
```

To make the code type-check, it is necessary to write an explicit annotation, either to restrict `use`'s argument or to expand `method`'s parameter type.

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 chronological 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 25.7) 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 https://www.lexspoon.org/expannots/.

Unlike the Units Checker that is distributed with the Checker Framework (see Section 17), 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 [WPM+09], a checker for thread locality, is available at http://www.it.uu.se/research/upmarc/loci/.

23.4 Safety-Critical Java checker

A checker for Safety-Critical Java (SCJ, JSR 302) [TPV10] is available at http://sss.cs.purdue.edu/projects/oscj/checker/checker.html. Developer resources are available at the project page https://code.google.com/archive/p/scj-jsr302/.

23.5 Generic Universe Types checker

A checker for Generic Universe Types [DEM11], a lightweight ownership type system, is available from https://ece.uwaterloo.ca/~wdietl/ownership/.

23.6 EnerJ checker

A checker for EnerJ [SDF+11], 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 https://checkerframework.org/sparta/. The paper "Collaborative verification of information flow for a high-assurance app store" appeared in CCS 2014.

23.9 Immutability checkers: IGJ, OIGJ, and Javari

Javari [TE05], IGJ [ZPA+07], and OIGJ [ZPL+10] are type systems that enforce immutability constraints. Type-checkers for all three type systems were distributed with the Checker Framework through release 1.9.13 (dated 1 April 2016). If you wish to use them, install Checker Framework version 1.9.13.

They were removed from the main distribution on June 1, 2016 because the implementations were not being maintained as the Checker Framework evolved. The type systems are valuable, and some people found the type-checkers useful. However, we wanted to focus on distributing checkers that are currently being maintained.

23.10 Read Checker for CERT FIO08-J

CERT rule FIO08-J describes a rule for the correct handling of characters/bytes read from a stream.

The Read Checker enforces this rule. It is available from `https://github.com/opprop/ReadChecker`.

23.11 SQL checker that supports multiple dialects

jOOQ is a database API that lets you build typesafe SQL queries. jOOQ version 3.0.9 and later ships with a SQL checker that provides even more safety: it ensures that you don't use SQL features that are not supported by your database implementation. You can learn about the SQL checker at `https://blog.jooq.org/2016/05/09/jsr-308-and-the-checker-framework-add-even-more-typesafety-to-jooq-3-9/`.

23.12 Glacier: Class immutability

Glacier [CNA+17] enforces transitive class immutability in Java. According to its webpage:

- Transitive: if a class is immutable, then every field must be immutable. This means that all reachable state from an immutable object's fields is immutable.
- Class: the immutability of an object depends only on its class's immutability declaration.
- Immutability: state in an object is not changable through any reference to the object.

Chapter 24

Generics and polymorphism

This chapter describes support for Java generics (also known as "parametric polymorphism") and polymorphism over type qualifiers.

Section 24.2 describes support for polymorphism over type qualifiers.

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 11.1, page 86) 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`.

Defaulting never affects a use of a type variable, even if the type variable use has no explicit annotation. Defaulting helps to choose a single type qualifier for a concrete Java class or interface. By contrast, a type variable use represents a set of possible types.

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 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 loophole 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 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.

24.1.8 The Bottom type

Many type systems have a `*Bottom` type that is used only for the `null` value, dead code, and some erroneous situations. A programmer should rarely write the bottom type.

One use is on a lower bound, to indicate that any type qualifier is permitted. A lower-bounded wildcard indicates that a consumer method can accept a collection containing any Java type above some Java type, and you can add the bottom type qualifier as well:

```
public static void addNumbers(List<? super @SignednessBottom Integer> list) { ... }
```

24.2 Qualifier polymorphism

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 in 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 the generics feature of Java or a template in C++. 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:

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;
import org.checkerframework.framework.qual.PolymorphicQualifier;

@PolymorphicQualifier
```

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

See Section 24.2.3 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. You use them in the same cases, such as:

- A method operates on collections with different types of elements.
- Two different arguments have the same type, without constraining them to be one specific type.
- A method returns a value of the same type as its argument.

If a method is written using Java generics, it usually does not need qualifier polymorphism. If you can use Java's generics, then that is often better. On the other hand, 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 Java compiler still treats the base Java types non-generically.

In some cases, you don't need qualifier polymorphism because subtyping already provides the needed functionality. `String` is a supertype of `@Interned String`, so a method `toUpperCase` that is declared to take a `String` parameter can also be called on an `@Interned String` argument.

24.2.3 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 you can use `@PolyAll`. This is better than trying to write 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;
}
```

24.2.4 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. (Section 24.2.5 describes some exceptions to this rule: times when it makes sense to write a single polymorphic qualifier in a signature.)

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 main 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 particular example, it would be cleaner to rewrite your code to use Java generics, if you can do so:

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

It is unusual, but permitted, to write just one polymorphic qualifier, on a return type. This is just like it is unusual, but permitted, to write just one occurrence of a generic type parameter, on a return type. An example of such a method is `Collections.emptyList()`.

24.2.5 Using a single polymorphic qualifier in a method signature

As explained in Section 24.2.4, you will usually use a polymorphic qualifier multiple times in a signature. This section describes situations when it makes sense to write just one polymorphic qualifier in a method signature. Some of these situations can be avoided by writing a generic method, but in legacy code it may not be possible for you to change a method to be generic.

Using a single polymorphic qualifier on an element type

It can make sense to use a polymorphic qualifier just once, 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) { ... }
```

This example uses arrays, but analogous examples exist that use collections.

Using a single polymorphic qualifier to indicate all arguments are legal

A single `@PolyAll` annotation can indicate that any possible value is permitted to be passed. For example:

```
boolean eq(@PolyAll Object other) {
  return this == other;
}
```

The `@PolyAll` annotation applies to all type systems. It would be infeasible to write the top qualifier for every possible type system and to update this method's annotation whenever a new type system is defined.

By contrast, a declaration of `eq` without `@PolyAll`:

```
boolean eq(Object other) {
  return this == other;
}
```

would reject some calls, in type systems where the default type qualifier applied to `Object` is not the top type.

A related use of a single polymorphic qualifier is to override a generic type. For example, the annotation on `Comparable.compareTo()` is:

```
public interface Comparable<T extends @NonNull Object> {
  @Pure int compareTo(@PolyAll @NonNull T a1);
}
```

which indicates that, for every type system other than the nullness type system, every value is permitted as an argument, regardless of how the `Comparable` type was instantiated. For example, this call is legal:

```
Comparable<@MyBottom String> cble;
@MyTop String s;
...
cble.compareTo(s);
```

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, facts about read-only usage are not built into any of the type-checkers. Therefore, when using 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 11, page 85), 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 25), `enum` values, string literals, and method receivers are always non-null.
   - In the Interning type system (see Chapter 6, page 55), 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 30.7), 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 Java types to be treated as if they had a default annotation. (Defaulting never applies to uses of type varables, even if they do not have an explicit type 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. (Some of these currently do not work in stub files.)

- Use the innermost user-written `@DefaultQualifier`, as explained in this section.
- Use the default specified by the type system designer (Section 30.4.4); 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 `Class` name of an annotation. 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(Tainted.class)
  public boolean isJavaFile(File myfile) {  // myFile has type "@Tainted File"
    ...
  }
}
```

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

```
@DefaultQualifiers({
  @DefaultQualifier(NonNull.class),
  @DefaultQualifier(Tainted.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`.

The type system applies that default qualifier to most but not all types. In particular, unless otherwise stated, every type system uses 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 (some) 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:

- Local variables are defaulted to top because type refinement (Section 25.4) is applied to local variables. If a local variable starts as the top type, then the Checker Framework refines it to the best (most specific) possible type based on assignments to it. As a result, a programmer rarely writes an explicit annotation on any of those locations.
  Variables defaulted to top include local variables, resource variables in the try-with-resources construct, variables in for statements, and catch arguments (known as exception parameters in the Java Language Specification). Exception parameters need to have the top type because exceptions of arbitrary qualified types can be thrown and the Checker Framework does not provide runtime checks.
- Cast and instanceof types are not really defaulted to top. Rather, they are given the same type as their argument, which is the most specific possible type. That would also have been the effect if they were given the top type and then flow-sensitively refined to the type of their 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
@DefaultFor({ TypeUseLocation.EXCEPTION_PARAMETER })
public @interface NonNull { }

public @interface Nullable { }
```

As mentioned above, the exception parameters are always non-null, so `@DefaultFor({ TypeUseLocation.EXCEPTION_PARAMETER })` on `@NonNull` overrides the CLIMB-to-top rule.

A type system designer can specify defaults that differ from 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.

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, and do computations over them, to understand the meaning of an unannotated type in a given file.

Computation is necessary because different annotations might be inherited from a supertype and an interface, or from two interfaces. For return types, the inherited type should be the least upper bound of all annotations on overridden implementations in supertypes. For method parameters, the inherited type should be the greatest lower bound of all annotations on overridden implementations in supertypes. In each case, an error would be thrown if no such annotations existed.

In the future, this feature 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;
```

The Checker Framework copies these annotations because wildcards must be within the bounds of their corresponding type parameter. By contrast, if the bounds of a wildcard were defaulted differently from the bounds of its corresponding type parameter, then there would be many false positive `type.argument.type.incompatible` warnings

Here is another example of two equivalent wildcard declarations:

```
class MyList<@Regex(5) T extends @Regex(1) Object> {}

MyList<?> listOfRegexes;
MyList<@Regex(5) ? extends @Regex(1) Object> listOfRegexes;
```

Note, this copying of annotations for a wildcard's bounds applies only to unbounded wildcards. The two wildcards in the following example are equivalent.

```
class MyList<@NonNull T extends @Nullable Object> {}

MyList<? extends Object> listOfNonNulls;
MyList<@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). Also note that the `MyList` class declaration could have been more succinctly written as: `class MyList<T extends @Nullable Object>` where the lower bound is implicitly the bottom annotation: `@NonNull`.

25.3.5 Default qualifiers for `.class` files (conservative library defaults)

(*Note:* Currently, the conservative library defaults presented in this section are off by default and can be turned on by supplying the `-AuseDefaultsForUncheckedcode=bytecode` command-line option. In a future release, they will be turned on by default and it will be possible to turn them off by supplying a `-AuseDefaultsForUncheckedCode=-bytecode` 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 29.4 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. Unless otherwise noted, the choice is:

- For method parameters and lower bounds, use the bottom qualifier (see Section 30.4.7).
- For method return values, fields, and upper bounds, use the top qualifier (see Section 30.4.7).

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 29) or alternately suppress the warnings (see Chapter 26).

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.

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

The checkers soundly treat local variables and expressions within a method body as having a subtype of their declared or defaulted (Section 25.3.1) type. This functionality reduces your burden of annotating types in your program and eliminates some false positive warnings, but it never introduces unsoundness nor causes an error to be missed. This

functionality works within a method, but you still need to annotate method signatures (parameter and return type) and field types.

By default all checkers automatically incorporate type refinement. Most of the time, users don't have to think about, and may not even notice, type refinement. (And most readers can skip reading this section of the manual, except possibly the examples in Section 25.4.1.) 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 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 33.2) that includes a small piece of Java code that reproduces the problem.

25.4.1 Type refinement examples

Suppose you write

```
@Nullable String myVar;
...
if (myVar != null) {
  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. However, the Nullness Checker does not issue a warning for the call `myVar.hashCode()` in the code above. Within the body of the `if` test, the type of `myVar` is `@NonNull String`, even though `myVar` is declared as `@Nullable String`.

Here is another example:

```
@Nullable String myVar;
...                    // myVar has type @Nullable String
myVar = "hello";
...                    // myVar has type @NonNull String
myVar.hashCode();
...
myVar = myMap.get(someKey);
...                    // myVar has type @Nullable String
```

The Nullness Checker does not issue a warning for the call `myVar.hashCode()` above because after the assignment, the type-checker treats `myVar` as having type `@NonNull String`, which is more precise than the programmer-written type.

Flow-sensitive type refinement applies to every checker, including new checkers that you write. Here is an example for the Regex Checker (Chapter 11, page 85):

```
void m2(@Unannotated String s) {
  s = RegexUtil.asRegex(s, 2);  // asRegex throws error if arg is not a regex
                                // with the given number of capturing groups
  ...   // s now has type "@Regex(2) String"
}
```

As a further example, 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
}
```

This example shows the general rules for when 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).

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 with declared type `@NonNull` type is never inferred to be treated as possibly-null).

25.4.2 Types that are not refined

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[]`. For details, see Section 24.1.6 and Section 25.1.

25.4.3 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 25)
- Map Key Checker to track which values are keys in a map (see Chapter 4, page 50)
- Optional Checker for errors in using the `Optional` type (see Chapter 5, page 53)
- Lock Checker for concurrency and lock errors (see Chapter 7, page 59)
- Index Checker for array accesses (see Chapter 8, page 70)
- Regex Checker to prevent use of syntactically invalid regular expressions (see Chapter 11, page 85)
- Format String Checker to ensure that format strings have the right number and type of `%` directives (see Chapter 12, page 88)
- Internationalization Format String Checker to ensure that i18n format strings have the right number and type of `{}` directives (see Chapter 13, page 95)

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 6, page 55)
- Property File Checker to ensure that valid keys are used for property files and resource bundles (see Chapter 14, page 101)
- Internationalization Checker to ensure that code is properly internationalized (see Chapter 14.2, page 102)
- Signature String Checker to ensure that the string representation of a type is properly used, for example in `Class.forName` (see Chapter 15, page 104).
- Constant Value Checker to determine whether an expression's value can be known at compile time (see Chapter 19, page 119)

Type systems that cannot support a run-time test are:

- Initialization Checker to ensure all fields are set in the constructor (see Chapter 3.9, page 35)
- Fake Enum Checker to allow type-safe fake enum patterns and type aliases or typedefs (see Chapter 9, page 79)
- Tainting Checker for trust and security errors (see Chapter 10, page 83)
- GUI Effect Checker to ensure that non-GUI threads do not access the UI, which would crash the application (see Chapter 16, page 107)
- Units Checker to ensure operations are performed on correct units of measurement (see Chapter 17, page 112)
- Signedness Checker to ensure unsigned and signed values are not mixed (see Chapter 18, page 116)
- Aliasing Checker to identify whether expressions have aliases (see Chapter 20, page 123)
- Subtyping Checker for customized checking without writing any code (see Chapter 22, page 132)

25.4.4 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";
  ```

  Such an inferred type may leak to the public interface of the class. If you wish 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. Consider the following example, where `updatedAt` is a nullable field:

  ```
  class DBObject {
    @Nullable Date updatedAt;

    void m() {
      // updatedAt is @Nullable, so warning about .getTime()
      ... updatedAt.getTime() ... // warning about possible NullPointerException

      if (updatedAt == null) {
        updatedAt = new Date();
      }

      // updatedAt is now @NonNull, so .getTime() call is OK
      ... updatedAt.getTime() ...
    }
  }
  ```

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

A method call may invalidate inferences about field types; see Section 25.4.5.

25.4.5 Side effects, determinism, purity, and flow-sensitive analysis

Side effect analysis is important for helping a checker reason about the values of expressions.

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, you can 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 externally-visible side effects.

`@Deterministic` indicates that if the method is called multiple times with identical arguments, then it returns the identical result according to == (not just according to `equals()`).

`@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() { ... }

void m() {
  ...
  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 `myField` 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) { ... }

void m() {
```

```
  ...
  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 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.Object;
attempting to use an incompatible purity declaration
    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.

The `@TerminatesExecution` annotation indicates that a given method never returns. This can enable the flow-sensitive type refinement to be more precise.

25.4.6 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`
- `@EnsuresLockHeld`
- `@EnsuresLockHeldIf`
- `@GuardedBy`
- `@Holding`

The set of permitted expressions is a subset of all Java expressions:

- the receiver object, `this`. You can write `this` to annotate any variable or declaration where you could write `this` in code. Notably, it cannot be used in annotations on declarations of static fields or methods. For a field, `this` is the field's receiver, i.e. its container. For a local variable, it is the method's receiver.
- 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).
- `<self>`, i.e. the value of the annotated reference (non-primitive) variable. Currently only defined for the `@GuardedBy` type system. For example, `@GuardedBy("<self>") Object o` indicates that the value referenced by `o` is guarded by the intrinsic (monitor) lock of the value referenced by `o`.
- a formal parameter, represented as `#` 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 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.)
- a local variable. Write the variable name. For example: `myLocalVar`. The variable must be in scope; for example, a method annotation on method `m` cannot mention a local variable that is declared inside `m`.
- 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`. You may optionally omit a leading "`this.`", just as in Java. Thus, `this.next` and `next` are equivalent.

- an array access. For example: `this.myArray[i]`, `vals[#1]`.
- literals: string, integer, long, null, class literals.
- 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")`.
  One unusual feature of the Checker Framework's Java expressions is that a method call is allowed to have side effects. Other tools forbid methods with side effects (and doing so is necessary if a specification is going to be checked at run time via assertions). The Checker Framework enables you to state more facts. For example, consider the annotation on `java.io.BufferedReader.ready()`:

  ```
  @EnsuresNonNullIf(expression="readLine()", result=true)
  @Pure public boolean ready() throws IOException { ... }
  ```

  This states that if `readLine()` is called immediately after `ready()` returns true, then `readLine()` returns a non-null value. Currently, the Checker Framework cannot prove all contracts about method calls, so you may need to suppress some warnings.

Limitations: The following Java expressions may not currently be written:

- Some literals: floats, doubles, and chars.
- String concatenation expressions.
- Mathematical operators (plus, minus, division, ...).
- Comparisons (equality, less than, etc.).

Additionally, it is not possible to write quantification over all array components (e.g. to express that all array elements are non-null). There is no such Java expression, but it would be useful when writing specifications.

25.6 Field invariants

Sometimes a field declared in a superclass has a more precise type in a subclass. To express this fact, write `@FieldInvariant` on the subclass. It specifies the field's type in the class on which this annotation is written. The field must be declared in a superclass and must be final.

For example,

```
class Person {
  final @Nullable String nickname;
  public Person(@Nullable String nickname) {
    this.nickname = nickname;
  }
}

// A rapper always has a nickname.
@FieldInvariant(qualifier = NonNull.class, field = "nickname")
class Rapper extends Person {
  public Rapper(String nickname) {
    super(nickname);
  }
  void method() {
    ... nickname.length() ...   // legal, nickname is non-null in this class.
  }
}
```

A field invariant annotation can refer to more than one field. For example, `@FieldInvariant(qualifier = NonNull.class, field = {fieldA, fieldB})` means that `fieldA` and `fieldB` are both non-null in the class upon which the annotation is written. A field invariant annotation can also apply different qualifiers to different fields. For example,

`@FieldInvariant(qualifier = {NonNull.class, Untainted.class}, field = {fieldA, fieldB})` means that `fieldA` is non-null and `fieldB` is untainted.

This annotation is inherited: if a superclass is annotated with `@FieldInvariant`, its subclasses have the same annotation. If a subclass has its own `@FieldInvariant`, then it must include the fields in the superclass annotation and those fields' annotations must be a subtype (or equal) to the annotations for those fields in the the superclass `@FieldInvariant`.

Currently, the `@FieldInvariant` annotation is trusted rather than checked. In other words, the `@FieldInvariant` annotation introduces a loophole in the type system, which requires verification by other means such as manual examination.

25.7 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 25.7.1), 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.

25.7.1 `@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 fix the underlying problem, by changing the code or its annotations. For each warning, follow the methodology in Section 2.4.6 to correct the underlying problem.

This section describes what to do if the methodology of Section 2.4.6 indicates that you need to suppress the warning. You won't change your code, but you will prevent 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 warning message in a single variable initializer, 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 `-AuseDefaultsForUncheckedCode=source` command-line option (Section 29.4),
- the `-Alint` command-line option enables/disables optional checks (Section 26.6), or
- not running the annotation processor (Section 26.7).

Some type checkers can suppress warnings via

- checker-specific mechanisms (Section 26.8).

The rest of this chapter explains these mechanisms in turn.

26.1 `@SuppressWarnings` annotation

You can suppress specific errors and warnings by use of the `@SuppressWarnings` annotation, for example `@SuppressWarnings("interning")` 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* or *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. (`assert` plus `@AssumeAssertion` can be used between statements and can affect arbitrary expressions; see Section 26.2.)

Always write a `@SuppressWarnings` annotation on the smallest possible scope. To reduce the scope of a `@SuppressWarnings` annotation, it is sometimes desirable to refactor the code. You might extract an expression into a local variable, so that warnings can be suppressed just for that local variable's initializer expression. Likewise, you might extract some code into a separate method, so that warnings can be suppressed just for its body.

As an example, consider suppressing a warning at a cast that you know is safe. Here is an example that uses the Tainting Checker (Section 10); 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

You may be able to suppress a warning by writing `@AssumeAssertion` for some subpart of it. See Section 26.2.

Otherwise, if a particular expression causes a false positive warning, don't suppress warnings for a larger expression or an entire method body. It is better to extract the expression into a local variable and place a `@SuppressWarnings` annotation on the variable declaration. 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 programmer knows 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. Also make it clear what error is being suppressed; this is particularly important when the `@SuppressWarnings` is on a method declaration and the suppressed warning might be anywhere in the method body.

This documentation will help you and others to understand the reason for the `@SuppressWarnings` annotation. It will also help you audit your code to verify all the warning suppressions. (The code is correct only if the checker issues no warnings *and* each `@SuppressWarnings` is correct.)

Here are some terse examples from plume-lib:

```
@SuppressWarnings("purity") // side effect to local state of type BitSet
@SuppressWarnings("cast") // cast is redundant (except when checking nullness)
@SuppressWarnings("interning") // FbType.FREE is interned but is not annotated
@SuppressWarnings("interning") // equality testing optimization
@SuppressWarnings("nullness") // used portion of array is non-null
@SuppressWarnings("nullness") // oi.factory is a static method, so null first argument is OK
```

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:

```
while (c != Object.class) {
  ...
  c = c.getSuperclass();
  assert c != null
    : "@AssumeAssertion(nullness): c was not Object, so its superclass is not null";
 }
```

the Nullness Checker assumes that `c` is non-null from the `assert` statement forward (including on the next iteration through the loop).

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). Sometimes, writing an assertion and `@AssumeAssertion` is a less disruptive code change than refactoring to create a location where `@SuppressWarnings` can be written.

The `-AassumeAssertionsAreEnabled` and `-AassumeAssertionsAreDisabled` command-line options (Section 25.4.6) 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 possible 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 `NullnessUtil.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 `NullnessUtil.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 that belong to the `java.lang` or `java.util` packages. (Depending on your shell or other tool, you might need to change or remove the quoting.)

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 6.3). 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 Don't run the processor

You can 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.

You can direct your build system to avoid compiling certain parts of your code. For example, the `-Dmaven.test.skip=true` command-line argument tells Maven not to compile (or run) the tests.

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 9.4) and the Units Checker (Section 17.5).

Chapter 27

Handling legacy code

Section 2.4.2 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 2.2.3.

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 29 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 Chapter 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 29 discusses adding annotations to signatures when you do not have source code available. Chapter 26 discusses suppressing warnings.

27.1.1 Declaration annotations

When a declaration annotation is an alias for a type annotation, then the Checker Framework may move the annotation before replacing it by the canonical version. (If the declaration annotation is in an org.checkerframework package, it is not moved.)

For example,

```
import android.support.annotation.NonNull;
...
@NonNull Object [] returnsArray();
```

is treated as if the programmer had written

```
import org.checkerframework.checker.nullness.qual.NonNull;
...
Object @NonNull [] returnsArray();
```

because Android's `@NonNull` annotation is a declaration annotation, which is understood to apply to the top-level return type of the annotated method.

When possible, you should use type annotations rather than declaration annotations.

Chapter 28

Type inference

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

1. Type inference during type-checking: The type-checker fills in an appropriate type where the programmer didn’t write one, but does not change the source code. See Section 28.1.
2. Type inference to annotate a program: As a separate step before type-checking, a type inference tool inserts type qualifiers into the source code. See Section 28.2.

Each variety has its own advantages, discussed below. Advantages of *all* varieties of type 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.

28.1 Local type inference during type-checking

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 does not change the source code. 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 warning.

Local type inference is built into the Checker Framework. Every checker automatically uses it. As a result, a programmer typically does not have to write any qualifiers inside the body of a method. However, it primarily works within a method, not across method boundaries. The source code must already contain annotations for method signatures (arguments and return values) and fields.

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.
- If the code changes, then there is no old annotation that might need to be updated.
- Within-method type inference occurs automatically. The programmer doesn’t have to do anything to take advantage of it.

For more details about local type inference during type-checking, also known as “flow-sensitive local type refinement”, see Section 25.4.

28.2 Type inference to annotate a program

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 make the program type-check. (If no such set exists, for example because the program is not

type-correct, then the inference tool does its best but makes no guarantees.) 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.

Advantages of this variety of type inference include:

- The inference may be more precise by taking account of the entire program rather than just reasoning one method at a time.
- The program source code contains documentation in the form of type qualifiers, which can aid programmer understanding and may make type-checking warnings more comprehensible.

28.2.1 Type inference tools

This section lists tools that take a program and output a set of annotations for it. It first lists tools that work only for a single type system (but may do a more accurate job for that type system) then lists general tools that work for any type system.

Type inference for specific type systems

Section 3.3.7 lists several tools that infer annotations for the Nullness Checker.

If you run the Checker Framework with the `-AsuggestPureMethods` command-line option, it will suggest methods that can be marked as `@SideEffectFree`, `@Deterministic`, or `@Pure`; see Section 25.4.5.

Type inference for any type system

By supplying the `-Ainfer` command-line option, any type-checker can infer annotations. See Section 28.3.

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`.

28.3 Whole-program inference

Whole-program inference is an interprocedural inference that infers types for fields, method parameters, and method return types that do not have a user-written annotation (for the given type system). The inferred types are inserted into your program. The inferred type is the most specific type that is compatible with all the uses in the program. For example, the inferred type for a field is the least upper bound of the types of all the expressions that are assigned into the field.

To use whole-program inference, make sure that `insert-annotations-to-source`, from the Annotation File Utilities project, is on your path (for example in the `$PATH` environment variable). Then, run the script `checker-framework/checker/bin/infer-and-annotate.sh`. Its command-line arguments are:

1. Optional: Command-line arguments to `insert-annotations-to-source`.
2. Processor's name.
3. Target program's classpath. This argument is required; pass "" if it is empty.
4. Optional: Extra processor arguments which will be passed to the checker, if any. You may supply any number of such arguments, or none. Each such argument must start with a hyphen.
5. Optional: Paths to `.jaif` files used as input in the inference process.
6. Paths to `.java` files in the program.

For example, to add annotations to the `plume-lib` project:

```
git clone https://github.com/mernst/plume-lib.git
cd plume-lib
make jar
$CHECKERFRAMEWORK/checker/bin/infer-and-annotate.sh \
    "LockChecker,NullnessChecker" java/plume.jar:java/lib/junit-4.12.jar \
    -AprintErrorStack \
    `find java/src/plume/ -name "*.java"`
# View the results
git diff
```

You may need to wait a few minutes for the command to complete. You can ignore warnings that the command outputs while trying different annotations in your code.

It is recommended that you run `infer-and-annotate.sh` on a copy of your code, so that you can see what changes it made and so that it does not change your only copy. One way to do this is to work in a clone of your repository that has no uncommitted changes.

Whole-program inference differs from type refinement (Section 25.4) in three ways. First, type refinement only works within a method body. Second, type refinement always refines the current type, regardless of whether the value already has an annotation in the source code. Third, whole-program inference can infer a subtype or a supertype of the default type, by contrast with type refinement which always refines the current type to a subtype.

28.3.1 Whole-program inference ignores some code

Whole-program inference ignores code within the scope of a `@SuppressWarnings` annotation with an appropriate key (Section 26.1). In particular, uses within the scope do not contribute to the inferred type, and declarations within the scope are not changed. You should remove `@SuppressWarnings` annotations from the class declaration of any class you wish to infer types for.

As noted below, whole-program inference requires invocations of your code, or assignments to your methods, to generalize from. If a field is set via reflection (such as via injection), then whole-program inference would produce an inaccurate result. There are two ways to make whole-program inference ignore such a field. (1) You probably have an annotation such as `@Inject` or `@Option` that indicates such fields. Meta-annotate the declaration of the `Inject` or `Option` annotation with `@IgnoreInWholeProgramInference`. (2) Annotate the field to be ignored with `@IgnoreInWholeProgramInference`.

Whole-program inference, for a type-checker other than the Nullness Checker, ignores (pseudo-)assignments where the right-hand-side is the `null` literal.

28.3.2 Manually checking whole-program inference results

As with any type inference tool, it is a good idea to manually examine the results.

- Whole-program inference can produce undesired results when your code has non-representative or erroneous calls to a particular method or assignments to a particular field, as explained below. This is especially noticeable when the arguments or assignments are literals.
- If an annotation is inferred for a *use* of type variables; it might be more appropriate for you to move those annotations to the corresponding upper bounds of the type variable *declaration*.

Poor whole-program inference results due to non-representative uses

Whole-program inference determines a method parameter's type annotation based on what arguments are passed to the method, but not on how the parameter is used within the method body.

- If the program contains erroneous calls, the inferred annotations may reflect those errors.
  Suppose you intend method `m2` to be called with non-null arguments, but your program contains an error and one of the calls to `m2` passes `null` as the argument. Then the tool will infer that `m2`'s parameter has `@Nullable` type. You should correct the bug and re-run inference.

- If the program uses (say) a method in a limited way, then the inferred annotations will be legal for the program as currently written but may not be as general as possible and may not accommodate future program changes.
  Here are some examples:
  - Suppose that your program currently calls method `m1` only with non-null arguments. The tool will infer that `m1`'s parameter has `@NonNull` type. If you had intended the method to be able to take `null` as an argument and you later add such a call, the type-checker will issue a warning because the automatically-inserted `@NonNull` annotation is inconsistent with the new call.
  - If your program (or test suite) passes only `null` as an argument, the inferred type will be the bottom type, such as `@GuardedByBottom`.
  - It is common for whole-program inference to infer `@Interned` and `@Regex` annotations on String variables for which the analyzed code only uses a constant string.

  In each case, you can correct the inferred results manually, or you can add tests that pass additional values then re-run inference.

28.3.3 How whole-program inference works

This section explains how the `infer-and-annotate.sh` script works. If you merely want to run the script and you are not encountering trouble, you can skip this section.

If you supply to javac the command-line option `-Ainfer`, then the checker outputs `.jaif` files with refined types for fields and method signatures. The output `.jaif` files are located in the folder `build/whole-program-inference`, relative to where you executed the javac command.

You can use the Annotation File Utilities (`https://checkerframework.org/annotation-file-utilities/`) to insert these refined types in your program. Then, the next time that you run type-checking, there are likely to be fewer type-checking warnings.

Note that a three-step process is required:

1. Run the checker with the `-Ainfer` command-line option to produce a `.jaif` file. Some type-checking errors may result.
2. Insert the `.jaif` file's annotations in the program.
3. Run the checker again. Fewer type-checking errors may result.

A good approach is to repeatedly run the above process until there are no more changes to the inference results (that is, the `.jaif` file is unchanged between two runs). That is exactly what the `infer-and-annotate.sh` script does.

The `infer-and-annotate.sh` script insulates you from the clumsy multi-step process. The multi-step process is required because type-checking is modular: it processes each class only once, independently. Modularity enables you to run type-checking on only part of your program, and it makes type-checking fast. However, it has some disadvantages:

- The first run of the type-checker cannot take advantage of whole-program inference results because whole-program inference is only complete at the end of type-checking, and modular type-checking does not revisit any already-processed classes.
- The reason that multiple executions are required is that revisiting an already-processed class may result in a better estimate.

Chapter 29

Annotating libraries

When your code uses a library that is not currently being compiled, the Checker Framework looks up the library's annotations in its class files. Section 2.2.1 tells you how to use a version of a library that contains type annotations. 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. (See Section 25.3.5 for details, an example, and how to override this conservative behavior.) 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, or to part of your codebase, by use of the `-AskipUses` or `-AonlyUses` command-line option; see Sections 26.4–26.5.)

You can write annotations for a library, and make them known to a checker, in two ways.

1. Write annotations in a copy of the library's source code (for instance, in a fork of the library's GitHub project). In addition to writing annotations, adjust the build system to run pluggable-type-checking when compiling (see Chapter 31, page 205).
   Then, when doing pluggable type-checking, put the annotated library's `.jar` file on the classpath, as explained in Section 2.2.1.
   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.
   For instructions, see Section 29.2.
2. Write annotations in a "stub file", if you do not have access to the source code.
   Then, when doing pluggable type-checking, supply the "stub file" textually to the Checker Framework.
   This 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 be updated when a new version of the library is released, unless the API has changed (such as defining a new method).
   For instructions, see Section 29.5.

If you annotate a new library, please inform the Checker Framework developers so that your annotated library can be distributed with the Checker Framework. Sharing your annotations is useful even if the library is only partially annotated. However, as noted in Sections 2.4.2 and 29.1.4, you should annotate an entire class at a time. You may find type inference tools (Chapter 28.2, page 172) helpful when getting started, but you should always examine their results.

29.1 Tips for annotating a library

Section 2.4 gives general tips for writing annotations. This section gives tips that are specific to annotating a third-party library.

29.1.1 Don't change the code

When you annotate a library, you should only add annotations and, when necessary, documentation of those annotations in a Java comment (`//` or `/*...*/`).

Do not change the library's code, which will change its behavior and make the annotated version inconsistent with the unannotated version. (Sometimes it is acceptable to make a refactoring, such as extracting an expression into a new local variable in order to annotate its type or suppress a warning. Perform refactoring only when you cannot use `@AssumeAssertion`.)

Do not change publicly-visible documentation, such as Javadoc comments. That also makes the annotated version inconsistent with the unannotated version.

Do not change formatting and whitespace.

All of these changes increase the difference between upstream (the original version) and your annotated version. This makes it harder for others to understand what you have done, and they make it harder to pull changes from upstream into the annotated library.

29.1.2 Library annotations should reflect the specification, not the implementation

Publicly-visible annotations (including those on public method formal parameters and return types) should be based on the documentation, typically the method's Javadoc. In other words, your annotations should re-state facts that are in the Javadoc documentation.

Do not add requirements or guarantees beyond what the library author has already committed to. If a project's Javadoc says nothing about nullness, then you should not assume that the specification forbids null, nor that it permits it. (If the project's Javadoc explicitly mentions null everywhere it is permitted, then you can assume it is forbidden elsewhere, where the author omitted those statements.)

If a fact is not mentioned in the documentation, then it is usually an implementation detail. Clients should not depend on implementation details, which are prone to change without notice. (In some cases, you can infer some facts from the implementation, such as that null is permitted for a method's parameter or return type if the implementation explicitly passes null to the method or the method implementation returns null.)

If there is a fact that you think should be in the library's documentation but was unintentionally omitted by its authors, then please submit a bug report asking them to update the documentation to reflect this fact. After they do, you can also express the fact as an annotation.

29.1.3 Report bugs upstream

While annotating the library, you may discover bugs or missing/wrong documentation. If you have a documentation improvement or a bug fix, then open a pullrequest against the upstream version of the library. This will benefit all users of the library. And, once the documentation is updated, you will be able to add annotations that are consistent with the documentation.

29.1.4 Fully annotate the library, or indicate which parts you did not

If you do not annotate all the files in the library, then use `@AnnotatedFor` to indicate what files you have annotated.

Whenever you annotate any part of a file, fully annotate the file! That is, write annotations for all the methods and fields, based on their documentation. If you don't annotate the whole file, then other people won't know whether a particular method is unannotated because you didn't get to it yet or because you decided that it didn't need any annotations.

29.1.5 Verify your annotations

Ideally, after you annotate a file, you should type-check the file to verify the correctness and completeness of your annotations.

An alternative is to only annotate method signatures. The alternative is quicker but more error-prone, and there is no difference from the point of view of clients, who can only see annotations on public methods and fields. When you compile the library, the type-checker will probably issue warnings, but if you supply `-Awarns`, you don't have to suppress those warnings and the compiler will still produce `.class` files.

29.2 Creating an annotated library

This section describes how to create an annotated library.

1. See the the `org.checkerframework.annotatedlib` group in the Central Repository to find out whether an annotated version of the library already exists.
   If it exists, fork its repository in `https://github.com/typetools/`, tweak its buildfile to run an additional checker, and add annotations for that checker.
   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 (such as a git commit id); that will enable others to see exactly what edits you have made.
   Adjust the library's build process, such as an Ant, Gradle, or Maven buildfile.
   (a) Every time the build system runs the compiler, it should:
      - pass the `-AuseDefaultsForUncheckedCode=source,bytecode` command-line option and
      - run every pluggable type-checker for which any annotations exist, using `-processor` *`TypeSystem1`*`,`*`TypeSystem2`*`,`*`TypeSystem3`*

   (b) When the build system creates a `.jar` file, the resulting `.jar` file includes the contents of `checker-framework/checker/dist/checker-qual.jar`.

   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 unannotated 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 add your annotations to a repository in `https://github.com/typetools/` and upload a compiled artifact to the Maven Central Repository.
   - 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.
     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.

29.3 Creating an annotated JDK

This section tells how to create an annotated JDK for a new type system. Section 33.2.1 tells how to improve annotations for the JDK for an existing type system.

When you create a new checker, you need to also supply annotations for parts of the JDK, either as stub files or as source code that will be compiled and its annotations inserted into the JDK. This section describes the latter approach. It assumes that you have already followed the instructions for building the Checker Framework from source (Section 33.3).

If you are adding annotations to an existing annotated JDK, then you only need to follow the last two steps.

1. Get Java 8 source code (must be version 8)
   (a) Download JDK8 from Oracle
   (b) Open the downloaded tar (`tar -xvzf`)
   (c) Unzip the contained `src.zip` (resulting in folders: `com/`, `java/`, `javax/`, `launcher/`, `org/`)
2. Set up the Checker Framework (replace *mychecker* with the name of your checker)
   (a) `mkdir $JSR308/checker-framework/checker/jdk/`*`mychecker`*`/`
   `cd $JSR308/checker-framework/checker/jdk/`*`mychecker`*`/`
   `echo "include ../Makefile.jdk" > Makefile`
   (b) Add *mychecker* to `$JSR308/checker-framework/checker/jdk/MakeFile` in the definition of `CHECKER_DIRS` and in the definition of `ANNOTATED_CLASSES` (don't forget to add a closing parenthesis at the end of the definition of `ANNOTATED_CLASSES`!).
3. For each file you want to annotate, copy the JDK version into the directory `$JSR308/checker-framework/checker/jdk/`*`mychec`*
   using the same directory structure as the JDK.
   Whenever you add a file, fully annotate it, as described in Section 29.1.
   If you are only annotating fields and method signatures (but not ensuring that method bodies type-check), then you don't need to suppress warnings, because the JDK is built using the `-Awarns` command-line option.
4. Build the annotated JDK: run `./gradlew buildJdk`
   You may receive a compilation error because your files use a part of the JDK that is not in the annotated JDK. If the missing part is relatively small, please add it. If the missing part is large, then you can make small changes to the JDK source code, such as commenting out a method body. (Use `/*` and `*/` on their own lines, so that the diffs are minimal.)

29.4 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. See Section 29.1.)

There are two concerns:

- Ignoring type-checking errors in unannotated parts of the library. Use the `-AskipDefs` or `-AonlyDefs` command-line arguments; see Section 26.5.
- Putting conservative annotations in unannotated parts of the 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. This section describes how to do this. You use `@AnnotatedFor` to indicate which classes you have annotated.

29.4.1 The -AuseDefaultsForUncheckedCode=source,bytecode command-line argument

When compiling a library that is not fully annotated, use command-line argument -AuseDefaultsForUncheckedCode=source,bytecode. This causes the checker to behave normally for classes with a relevant @AnnotatedFor annotation. For classes without @AnnotatedFor, the checker uses unchecked code 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 -AuseDefaultsForUncheckedCode=source,bytecode 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 2.2.5). Writing @AnnotatedFor on a class doesn't necessarily mean that you wrote any annotations, but that the user examined the source code and verified that all appropriate annotations are present.

Whenever you compile a class using the Checker Framework, including when using the -AuseDefaultsForUncheckedCode=source,bytecode 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.

29.5 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 29.3 explains how you should choose between creating stub classes or creating an annotated library. Section 29.5.3 describes how to create stub classes. Section 29.5.1 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.

29.5.1 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.

If both bytecode and a stub file provide information for the same element, the stub file information is used. In particular, an un-annotated type variable in a stub file is used instead of annotations on a type variable in bytecode. This feature allows stub files to change the effective annotations in all possible situations. Use the -AstubWarnIfOverwritesBytecode command-line option to get a warning whenever a stub file overwrites bytecode annotations.

A file being compiled takes precedence over a stub file. If file `A.java` is being compiled, then any stub for class `A` is ignored.

When a stub file is provided by the author of a checker, the stub file is used automatically, with no need for the user to supply a command-line option.

29.5.2 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 6) could be:

```
import org.checkerframework.checker.interning.qual.Interned;
package java.lang;
@Interned class Class<T> { }
class String {
  @Interned String intern();
}
```

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: Imports may appear at the beginning of the file or after any package declaration. The only required import statements are the ones to import type annotations. Import statements for types are optional.

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

29.5.3 Creating a stub file

Stub files are generally stored together with the checker implementation, in the same directory as the checker's `.java` source code.

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 the Checker Framework, because it parses the stub files each time it runs.

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 29.5.3), 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` must be on your classpath.)

   ```
   cd nullness-stub
   java -cp $JSR308/checker-framework/checker/dist/checker.jar org.checkerframework.framework.stub.Stub
   ```

   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 29.5.2.

29.5.4 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 `-AstubWarnIfOverwritesBytecode` to warn whenever some element of a stub file overwrites annotations contained in bytecode.

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
```

29.6 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 30

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. There is also a developer manual for people who wish to edit the Checker Framework source code or make pull requests.

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

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

@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.

Three choices for creating your own checker are:

- Customizing an existing checker. Checkers that are designed for extension include the Subtyping Checker (Chapter 22, page 132), the Fake Enumeration Checker (Chapter 9, page 79), the Units Checker (Chapter 17, page 112), and a typestate checker (Chapter 23.1, page 137).
- Follow the instructions in this chapter to create a checker from scratch. This enables creation of checkers that are more powerful than customizing an existing checker.
- Copy and then modify a different existing checker — whether one distributed with the Checker Framework or a third-party one. This can be fast, or you can get tangled up if you don't fully understand the subtleties of the existing checker that you are modifying. Usually following the instructions in this chapter is easier. (If you are going to copy a checker, one good choice to copy and modify is the Regex Checker (Chapter 11, page 85). A bad choice is the Nullness Checker (Chapter 3, page 25), which is more sophisticated than anything you want to start out building.)

You do not need all of the details in this chapter, 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. The Javadoc documentation of the framework and the checkers is in the distribution and is also available online at `https://checkerframework.org/api/`.

If you write a new checker and wish to advertise it to the world, let us know so we can mention it in Chapter 23, page 137 or even include it in the Checker Framework distribution.

30.1 How checkers build on the Checker Framework

This table shows the relationship among tools that the Checker Framework builds on or that are built on the Checker Framework. 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.

<table>
<tr><td>Subtyping
Checker</td><td>Nullness
Checker</td><td>Index
Checker</td><td>Tainting
Checker</td><td>...</td><td>Your
Checker</td><td></td><td></td></tr>
<tr><td colspan="6">Base Checker
(enforces subtyping rules)</td><td>Type
inference</td><td>Other
tools</td></tr>
<tr><td colspan="6">Checker Framework
(enables creation of pluggable type-checkers)</td><td colspan="2">Annotation File Utilities
(.java ↔ .class files)</td></tr>
<tr><td colspan="8">Type Annotations syntax and classfile format
(no built-in semantics)</td></tr>
</table>

The Base Checker (more precisely, the `BaseTypeChecker`) enforces the standard subtyping rules. 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.

30.2 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. To simplify checker implementations, by default the Checker Framework automatically discovers the parts of a checker by looking for specific files. Thus, checker implementations follow a very formulaic structure. To illustrate, a checker for MyProp must be laid out as follows:

```
myPackage/
  | qual/                               type qualifiers
  | MyPropChecker.java                  [optional] interface to the compiler
  | MyPropVisitor.java                  [optional] type rules
  | MyPropAnnotatedTypeFactory.java     [optional] type introduction and dataflow rules
```

Note that `MyPropChecker.java` is required unless you are building on the Subtyping Checker.

Sections 30.4–30.5 describe the individual components of a type system as written using the Checker Framework:

30.4 **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`).

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

30.6 **Type rules.** You specify the type system semantics (type rules), violation of which yields a type error. A type system has two types of rules.

- Subtyping rules related to the type hierarchy, such as that every assignment satisfies a subtyping relationship. Your checker automatically inherits these subtyping rules from the Base Checker (Chapter 22), so there is nothing for you to do.
- Additional rules that are specific to your particular checker. For example, in the Nullness type system, only references whose type is `@NonNull` may be dereferenced. You write these additional rules yourself.

30.7 **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`.

30.8 **Dataflow rules.** These optional rules enhance flow-sensitive type qualifier inference (also known as local variable type inference).

30.3 Compiling and using a custom checker

You can place your checker's source files wherever you like.

- Forking the Checker Framework repository and putting your files in parallel to existing checker implementations is a particularly convenient choice. You must be able to compile the Checker Framework (Section 33.3).
- If you put your checker elsewhere, then when you compile your checker, the classpath must include `$CHECKERFRAMEWORK/checker/dist/checker.jar`.

Once your custom checker is written, using it is very similar to using a built-in checker (Section 2.2): simply pass the fully-qualified name of your `BaseTypeChecker` subclass to the `-processor` command-line option:

```
javac -processor mypackage.MyPropChecker SourceFile.java
```

Note that your custom checker's `.class` files must be on the Java classpath. Invoking a custom checker that builds on the Subtyping Checker is slightly different (Section 22.1).

30.3.1 Tips for creating a checker

To make your job easier, we recommend that you build your type-checker incrementally, testing at each phase rather than trying to build the whole thing at once.

Here is a good way to proceed.

1. Before you start coding, first write the user manual. The manual explains the type system, what it guarantees, how to use it, etc., from the point of view of a user. Writing the manual will help you flesh out your goals and the concepts, which are easier to understand and change in text than in an implementation. Section 30.12 gives a suggested structure for the manual chapter, which will help you avoid omitting any parts. Get feedback from someone else at this point to ensure that your manual is comprehensible.
   Once you have designed and documented the parts of your type system, you may also want to "play computer", manually type-checking some code according to the rules you defined. During manual checking, ask yourself what reasoning you applied, what information you needed, and whether your written-down rules were sufficient.
2. Implement the type qualifiers and hierarchy (Section 30.4).
   Write simple test cases that consist of only assignments, to test your type hierarchy. For instance, if your type hierarchy consists of a supertype `@UnknownSign` and a subtype `@NonNegative`, then you could write a test case such as:
   ```
   void foo(@UnknownSign int us, @NonNegative int nn) {
     @UnknownSign int a = us;
     @UnknownSign int b = nn;
     // :: error: assignment.type.incompatible
     @NonNegative int c = us;  // expected error on this line
     @NonNegative int d = nn;
   }
   ```
   Type-check your test files using the Subtyping Checker (Chapter 22, page 132).
3. Write the checker class itself (Section 30.5).
   Ensure that you can still type-check your test files and that the results are the same. You will not use the Subtyping Checker any more; you will call the checker directly, as in
   ```
   javac -processor mypackage.MyChecker File1.java File2.java ...
   ```
4. If your checker source code is in a clone of the Checker Framework repository, integrate your checker with the Checker Framework's Ant targets for testing (Section 30.10). This will make it much more convenient to run tests, and to ensure that they are passing, as your work proceeds.
5. Annotate parts of the JDK, if relevant (Section 30.9).
   Write test cases for at least some of the annotated JDK methods to ensure that the annotations are being properly read by your checker.

6. Implement type rules, if any (Section 30.6). (Some type systems need JDK annotations but don't have any additional type rules.)
   Before implementing type rules (or any other code in your type-checker), you are recommended to read the javadoc for the utility routines in the `org.checkerframework.javacutil` package, especially `AnnotationBuilder`, `AnnotationUtils`, `ElementUtils`, `TreeUtils`, `TypeAnnotationUtils`, and `TypesUtils`. Familiarity with them will help you to know how to access needed information and avoid reimplementing existing functionality.
   Write simple test cases to test the type rules, and ensure that the type-checker behaves as expected on those test files. For example, if your type system forbids indexing an array by a possibly-negative value, then you would write a test case such as:

```
void foo(String[] myarray, @UnknownSign int us, @NonNegative int nn) {
  myarray[us];  // expected error on this line
  myarray[nn];
}
```

7. Implement type introduction rules, if any (Section 30.7).
   Test your type introduction rules. For example, if your type system sets the qualifier for manifest literal integers and for array lengths, you would write a test case like the following:

```
void foo(String[] myarray) {
  @NonNegative nn1 = -1;  // expected error on this line
  @NonNegative nn2 = 0;
  @NonNegative nn3 = 1;
  @NonNegative nn4 = myarray.length;
}
```

8. Optionally, implement dataflow refinement rules (Section 30.8).
   Test them if you wrote any. For instance, if after an arithmetic comparison, your type system infers which expressions are now known to be non-negative, you could write a test case such as:

```
void foo(@UnknownSign int us, @NonNegative int nn) {
  @NonNegative nn2;
  nn2 = us;  // expected error on this line
  if (us > j) {
    nn2 = us;
  }
  if (us >= j) {
    nn2 = us;
  }
  if (j < us) {
    nn2 = us;
  }
  if (j <= us) {
    nn2 = us;
  }
  nn = us;  // expected error on this line
}
```

30.4 Annotations: Type qualifiers and hierarchy

A type system designer specifies the qualifiers in the type system (Section 30.4.1) 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 30.4.2), or procedurally through subclassing `QualifierHierarchy` or `TypeHierarchy` (Section 30.4.3).

30.4.1 Defining the type qualifiers

Type qualifiers are defined as Java annotations. In Java, an annotation is defined using the Java `@interface` keyword. Here is how to define a two-qualifier hierarchy:

```
package mypackage.qual;
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import org.checkerframework.framework.qual.DefaultQualifierInHierarchy;
import org.checkerframework.framework.qual.SubtypeOf;
/**
 * The run-time value of the integer is unknown.
 *
 * @checker_framework.manual #nonnegative-checker Non-Negative Checker
 */
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})
@SubtypeOf({})
@DefaultQualifierInHierarchy
public @interface UnknownSign {}
```

```
package mypackage.qual;
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import org.checkerframework.framework.qual.ImplicitFor;
import org.checkerframework.framework.qual.LiteralKind;
import org.checkerframework.framework.qual.SubtypeOf;
/**
 * Indicates that the value is greater than or equal to zero.
 *
 * @checker_framework.manual #nonnegative-checker Non-Negative Checker
 */
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})
@SubtypeOf({UnknownSign.class})
@ImplicitFor(literals = LiteralKind.NULL)
public @interface NonNegative {}
```

The `@SubtypeOf` meta-annotation indicates the parent in the type hierarchy.

The `@Target` meta-annotation indicates where the annotation may be written. All type qualifiers that users can write in source code should have the value `ElementType.TYPE_USE` and optionally with the additional value of `ElementType.TYPE_PARAMETER`, but no other `ElementType` values.

The annotations should be placed within a directory called `qual`, and `qual` should be placed in the same directory as your checker's source file.

For example, the Nullness Checker's source file is located at `.../nullness/NullnessChecker.java`. The `@NonNull` qualifier is defined in file `.../nullness/qual/NonNull.java`.

The Checker Framework automatically treats any annotation that is declared in the `qual` package as a type qualifier. (See Section 30.5.1 for more details.)

Your type system should include a top qualifier and a bottom qualifier (Section 30.4.7). In most cases, the bottom qualifier should be meta-annotated with `@ImplicitFor(literals=LiteralKind.NULL)`.

You should also define a polymorphic qualifier `@Poly`*`MyTypeSystem`* (Section 24.2).

The Javadoc of every type qualifier should include an example use of the qualifier.

30.4.2 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:

  ```
  @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.

  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:

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

  If the top qualifier of the hierarchy is the generic unqualified type (this is not recommended!), then its children will use `@SubtypeOf(Unqualified.class)`, but no `@SubtypeOf({})` annotation on the top qualifier `Unqualified` 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:

  ```
  @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 30.4.3) by overriding the `isSubtype` method in the checker class (Section 30.5). However, the declarative mechanism is sufficient for most type systems.

30.4.3 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). 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., `@NonNull Date` is a subtype of `@Nullable 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".)

30.4.4 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 overriding the `GenericAnnotatedTypeFactory.addCheckedCodeDefaults` method. You may do this even if you have declaratively defined the qualifier hierarchy.

If the default qualifier in the type hierarchy requires a value, there are ways for the type system designer to specify a default value both declaratively and procedurally, as well. To do so declaratively, append the string `default` *`value`* where *value* is the actual value you want to be the default, after the declaration of the value in the qualifier file. For instance, `int value() default 0;` would make `value` default to zero. Alternatively, the procedural method described above can be used.

30.4.5 Relevant Java types

A checker can use the `@RelevantJavaTypes` annotation on the checker class to specify which Java types are relevant to the checker. All irrelevant types without explicit annotations are defaulted to the top annotation.

30.4.6 Do not re-use type qualifiers

Every annotation should belong to only one type system. No annotation should be used by multiple type systems. This is true even of annotations that are internal to the type system and are not intended to be written by the programmer.

Suppose that you have two type systems that both use the same type qualifier `@A`. In a client program, a use of type `T` may require type qualifier `@A` for one type system but a different qualifier for the other type system. There is no annotation that a programmer can write to make the program type-check under both type systems.

This also applies to type qualifiers that a programmer does not write, because the compiler outputs `.class` files that contain an explicit type qualifier on every type — a defaulted or inferred type qualifier if the programmer didn't write a type qualifier explicitly.

30.4.7 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.) A way to do this is to meta-annotate your bottom type with `@ImplicitFor(literals=LiteralKind.NULL)`. This legal code will not type-check unless `null` has the bottom type:

```
<T> T f() {
    return null;
}
```

Some type systems have a special bottom type that is used *only* for the `null` value, and for dead code and other erroneous situations. In this case, users should only write the bottom qualifier on explicit bounds. In this case, the definition of the bottom qualifier should be meta-annotated with:

```
@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})
@TargetLocations({TypeUseLocation.EXPLICIT_LOWER_BOUND, TypeUseLocation.EXPLICIT_UPPER_BOUND})
```

Furthermore, by convention the name of such a qualifier ends with "`Bottom`".

The hierarchy shown in Figure 3.3 lacks a bottom qualifier, but the actual implementation does contain a (non-user-visible) bottom qualifier.

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.

30.4.8 Annotations whose argument is a Java expression (dependent type annotations)

Sometimes, an annotation needs to refer to a Java expression. Section 25.5 gives examples of such annotations and also explains what Java expressions can and cannot be referred to.

This section explains how to implement a dependent type annotation.

A "dependent type annotation" must have one attribute, `value`, that is an array of strings. The Checker Framework verifies that the annotation's arguments are valid expressions according to the rules of Section 25.5. If the expression is not valid, an error is issued and the string in the annotation is changed to indicate that the expression is not valid.

The Checker Framework standardizes the expression strings. For example, a field `f` can be referred to as either "`f`" or "`this.f`". If the programmer writes "`f`", the Checker Framework treats it as if the programmer had written "`this.f`". An advantage of this canonicalization is that comparisons, such as `isSubtype`, can be implemented as string comparisons.

The Checker Framework viewpoint-adapts type annotations on method, constructor, and field declarations at uses for those methods. For example, given the following class

```
class MyClass {
   Object field = ...;
   @Anno("this.field") Object field2 = ...;
}
```

and assuming the variable `myClass` is of type `MyClass`, then the type of `myClass.field` is viewpoint-adapted to `@Anno("myClass.field")`.

To use this built-in functionality, add a `@JavaExpression` annotation to any annotation element that should be interpreted as a Java expression. The type of the element must be an array of Strings. If your checker requires special handling

of Java expressions, your checker implementation should override `GenericAnnotatedTypeFactory.createDependentTypesHelper` to return a subclass of `DependentTypesHelper`.

Given a specific expression in the program (of type Tree or Node), a checker may need to obtain its canonical string representation. This enables the checker to create an dependent type annotation that refers to it, or to compare to the string expression of an existing expression annotation. To obtain the string, first create a `FlowExpressions.Receiver` object by calling `internalReprOf(AnnotationProvider, ExpressionTree)` or `internalReprOf(AnnotationProvider, Node)`. Then, call `toString()` on the `FlowExpressions.Receiver` object.

30.5 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:

```
package my.package;
import org.checkerframework.common.basetype.BaseTypeChecker;
@SupportedLintOptions({"dotequals"})
public final class InterningChecker extends BaseTypeChecker { }

package my.package;
import org.checkerframework.common.basetype.BaseTypeChecker;
@SupportedLintOptions({"flow", "cast", "cast:redundant"})
public class NullnessChecker extends BaseTypeChecker { }
```

(The `@SupportedLintOptions` annotation is optional, and many checker classes do not have one.)

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 166).

30.5.1 Indicating supported annotations

A checker must indicate the annotations that it supports (make up its type hierarchy), including whether it supports the polymorphic qualifier `@PolyAll`.

By default, a checker supports `PolyAll`, and all annotations located in a subdirectory called `qual` that's located in the same directory as the checker. Note that only annotations defined with the `@Target(ElementType.TYPE_USE)` meta-annotation (and optionally with the additional value of `ElementType.TYPE_PARAMETER`, but no other `ElementType` values) are automatically considered as supported annotations.

To indicate support for annotations that are located outside of the `qual` subdirectory, annotations that have other `ElementType` values, or to indicate whether a checker supports the polymorphic qualifier `@PolyAll`, checker writers can override the `createSupportedTypeQualifiers` method (see its Javadoc for details).

An aggregate checker (which extends `AggregateChecker`) does not need to specify its type qualifiers, but each of its component checkers should do so.

30.5.2 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 either 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 MyFile.java
   ```

   the user can write

   ```
   javac -processor MyUnitCheckers MyFile.java
   ```

   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 14.2, page 102), 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. Obtain the `AnnotatedTypeFactory` by calling `getTypeFactoryOfSubchecker`.

30.5.3 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({"myflag"})
```

to your checker subclass. The value of the flag can be queried using

```
checker.getLintOption("myflag", 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 MyChecker -Alint=myflag
```

Named string values

For more complicated options, one can use the standard `@SupportedOptions` annotation on the checker, as in:

```
@SupportedOptions({"myoption"})
```

The value of the option can be queried using

```
checker.getOption("myoption")
```

To pass an option on the command line, call javac as follows:

```
javac -processor MyChecker -Amyoption=p1,p2
```

The value is returned as a single string and you have to perform the required parsing of the option.

30.6 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. Put them in a file *MyChecker*`Visitor.java` that extends `BaseTypeVisitor`.

BaseTypeVisitor 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 (by calling `SourceChecker.report`) whenever the type system is violated.

Most type-checkers override only a few methods in `BaseTypeVisitor`. A checker's visitor overrides one method in the base visitor for each special rule in the type qualifier system. The last line of the overridden version is "`return super.visit`*TreeType*`(node, p);`". If the method didn't raise any error, the superclass implementation can perform standard checks.

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

30.6.1 AST traversal

The Checker Framework needs to do its own traversal of the AST even though it operates as an ordinary annotation processor [Dar06]. Java provides a visitor for Java code that is intended to be used by annotation processors, 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 or IntelliJ IDEA, but runs as an external tool (see Section 31.3).

30.6.2 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 sometimes necessary but is usually the wrong approach. It is better to write annotations, in a stub file or annotated JDK (Chapter 29), and leave the work to the standard type-checking rules.

30.7 Type factory: Implicit annotations (type introduction rules)

For some types and expressions, a qualifier should be treated as present even if a programmer did not explicitly write it. For example, every literal (other than `null`) has a `@NonNull` type. Section 25.3 explains the meaning of implicit qualifiers, such as that they cannot be overridden.

The implicit annotations may be specified declaratively (Section 30.7.1) and/or procedurally (Section 30.7.2). It is easiest to specify them declaratively, when the declarative method is sufficiently expressive.

30.7.1 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 four fields in `ImplicitFor`.

For example, consider these (partial) definitions of the `@Nullable` and `@NonNull` type qualifiers:

```
@SubtypeOf({})
@ImplicitFor(literals = { LiteralKind.NULL },
             typeNames = { java.lang.Void.class })
@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})
public @interface Nullable { }

@SubtypeOf( { Nullable.class } )
@ImplicitFor(literals = { LiteralKind.STRING },
             types = { TypeKind.PACKAGE, TypeKind.INT, TypeKind.BOOLEAN,
                       TypeKind.CHAR, TypeKind.DOUBLE, TypeKind.FLOAT,
                       TypeKind.LONG, TypeKind.SHORT, TypeKind.BYTE })
@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})
public @interface NonNull {}
```

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 30.13 can help you get started.

[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.

30.7.2 Procedurally specifying implicit annotations

If the `@ImplicitFor` annotation is not sufficiently expressive, then you can write code to set implicit annotations. To do so, create a subclass of `AnnotatedTypeFactory` and override its `addComputedTypeAnnotations` methods.

`AnnotatedTypeFactory`, when given a program expression, returns the expression's type. This should include not only the qualifiers that the programmer explicitly wrote in the source code, but also default and implicit annotations, and flow-sensitive local type inference (see Section 25.3 for explanations of these concepts).

To add implicit annotations, you should override `addComputedTypeAnnotations(Tree,AnnotatedTypeMirror)` (or `addComputedTypeAnnotations(Tree,AnnotatedTypeMirror,boolean)` if extending `GenericAnnotatedTypeFactory`) and `addComputedTypeAnnotations(Element,AnnotatedTypeMirror)`. The methods operate on `AnnotatedTypeMirror`, which is the Checker Framework's representation of an annotated type.

30.8 Dataflow: enhancing flow-sensitive type qualifier inference

By default, every checker performs flow-sensitive type refinement, also known as local type inference, as described in Section 25.4.

This section of the manual explains how to enhance the Checker Framework's built-in flow-sensitive type refinement. Most commonly, you will inform the Checker Framework about a run-time test that gives information about the type qualifiers in your type system. Section 25.4.3 (page 156) gives examples of type systems with and without run-time tests.

The steps to customizing type refinement are:

1. §30.8.1 Determine which expressions will be refined.
2. §30.8.2 Create required class and configure its use.
3. §30.8.3 Override methods that handle `Node`s of interest.
4. §30.8.4 Implement the refinement.

The Regex Checker's dataflow customization for the `RegexUtil.asRegex` run-time check is used as an example throughout the steps.

If needed, you can find more details about the implementation of flow-sensitive type refinement, and the control flow graph (CFG) data structure that it uses, in the Dataflow Manual.

30.8.1 Determine expressions to refine the types of

A run-time check or run-time operation involves multiple expressions (arguments, results). Determine which expression the customization will refine. This is usually specific to the type system and run-time test. There is no code to write in this step; you are merely determining the design of your type refinement.

For the program operation `op(a,b)`, you can refine the types in either or both of the following ways:

1. Change the result type of the entire expression `op(a,b)`.
   As an example (and as the running example of implementing a dataflow refinement), the `RegexUtil.asRegex` method is declared as:
   ```
   @Regex(0) String asRegex(String s, int groups) { ... }
   ```
   This annotation is sound and conservative: it says that an expression such as `RegexUtil.asRegex(myString, myInt)` has type `@Regex(0) String`. However, this annotation is imprecise. When the `group` argument is known at compile time, a better estimate can be given. For example, `RegexUtil.asRegex(myString, 2)` has type `@Regex(2) String`.
2. Change the type of some other expression, such as `a` or `b`.
   As an example, consider an equality test in the Nullness type system:
   ```
   @Nullable String s;
     if (s != null) {
       ...
   ```

```
} else {
  ...
}
```

The type of `s != null` is always `boolean`. However, in the true branch, the type of `s` can be refined to `@NonNull String`.

If you are refining the types of arguments or the result of a method call, then you may be able to implement your flow-sensitive refinement rules by just writing `@EnsuresQualifier` and/or `@EnsuresQualifierIf` annotations. When this is possible, it is the best approach.

Sections 30.8.2–30.8.4 explain how to create a transfer class when the `@EnsuresQualifier` and `@EnsuresQualifierIf` annotations are insufficient.

30.8.2 Create required class

In the same directory as *MyChecker*`Checker.java`, create a class named *MyChecker*`Transfer` that extends `CFTransfer`.

Leave the class body empty for now. Your class will add functionality by overriding methods of CFTransfer, which performs the default Checker Framework type refinement.

As an example, the Regex Checker's extended `CFTransfer` is `RegexTransfer`.

(If you disregard the instructions above and choose a different name or a different directory for your *MyChecker*`Transfer` class, you will also need to override the `createFlowTransferFunction` method in your type factory to return to return a new instance of the class.)

(As a reminder, use of `@EnsuresQualifier` and `@EnsuresQualifierIf` may obviate the need for a transfer class.)

30.8.3 Override methods that handle Nodes of interest

Decide what source code syntax is relevant to the the run-time checks or run-time operations you are trying to support. The CFG (control flow graph) represents source code as `Node`, a node in the abstract syntax tree of the program being checked (see "Program representation" below).

In your extended `CFTransfer` override the visitor method that handles the `Node`s relevant to your run-time check or run-time operation. Leave the body of the overriding method empty for now.

For example, the Regex Checker refines the type of a run-time test method call. A method call is represented by a `MethodInvocationNode`. Therefore, `RegexTransfer` overrides the `visitMethodInvocation` method:

```
public TransferResult<CFValue, CFStore> visitMethodInvocation(
  MethodInvocationNode n, TransferInput<CFValue, CFStore> in)  { ... }
```

Program representation

The `Node` subclasses can be found in the `org.checkerframework.dataflow.cfg.node` package. Some examples are `EqualToNode`, `LeftShiftNode`, `VariableDeclarationNode`.

A `Node` is basically equivalent to a javac compiler `Tree`.

See Section 30.13 for more information about `Tree`s. As an example, the statement `String a = "";`, is represented as this abstract syntax tree:

```
VariableTree:
  name: "a"
  type:
    IdentifierTree
      name: String
  initializer:
    LiteralTree
      value: ""
```

30.8.4 Implement the refinement

Each visitor method in `CFAbstractTransfer` returns a `TransferResult`. A `TransferResult` represents the refined information that is known after an operation. It has two components: the result type for the `Node` being evaluated, and a map from expressions in scope to estimates of their types (a `Store`). Each of these components is relevant to one of the two cases in Section 30.8.1:

1. 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`.)
   For example, When `RegexTransfer` evaluates a `RegexUtils.asRegex` invocation, it updates the `TransferResult`'s result type. This changes the type of the `RegexUtils.asRegex` invocation when its `Tree` is looked up by the `AnnotatedTypeFactory`. See below for details.
2. Updating the `Store` treats an expression as 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`.

The rest of 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`.

1. **Determine if the visited `Node` is of interest**
   A visitor method is invoked for all instances of a given `Node` kind in the program. The visitor must inspect the `Node` 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) {
    String receiverName = ((ClassNameNode) receiver).getElement().toString();

    // Is this a call to method RegexUtil.isRegex(s, groups)?
    if (receiverName.equals("RegexUtil")
        && ElementUtils.matchesElement(method,
               null, "isRegex", String.class, int.class)) {
          ...
```

2. **Determine the refined type**
   Sometimes the refined type is dependent on the parts of the operation, 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. For simplicity the below 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 Constant Value Checker (Chapter 19, page 119).

```
AnnotationMirror regexAnnotation;
Node count = n.getArgument(1);
if (count instanceof IntegerLiteralNode) {
  // argument is a literal integer
  IntegerLiteralNode iln = (IntegerLiteralNode) count;
```

```
    Integer groupCount = iln.getValue();
    regexAnnotation = factory.createRegexAnnotation(groupCount);
  } else {
    // argument is not a literal integer; fall back to @Regex(), which is the same as @Regex(0)
    regexAnnotation = AnnotationBuilder.fromClass(factory.getElementUtils(), Regex.class);
  }
```

3. **Return a `TransferResult` with the refined types**
   Recall that the type of an expression is refined by modifying the `TransferResult` returned by a visitor method. 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());
```

   As a result of this code, 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.

30.8.5 Disabling flow-sensitive inference

In the uncommon case that you wish to disable the Checker Framework's built-in flow inference in your checker (this is different than choosing not to extend it as described in Section 30.8), put the following two lines at the beginning of the constructor for your subtype of `BaseAnnotatedTypeFactory`:

```
// disable flow inference
super(checker, false);
```

30.9 Annotated JDK

You will need to supply annotations for relevant parts of the JDK; otherwise, your type-checker may produce spurious warnings for code that uses the JDK. As described in Section 29, there are two general ways to supply an annotated library: in Java files that will be compiled to `.class` files, or as stub files (partial Java source files).

It's easier to start out with stub files. If you need to annotate many classes (say, more than 20 or so), then you should create an annotated JDK.

To supply an annotated JDK that will be compiled, see Section 29.3.

To supply an annotated JDK as a stub file, create a file `jdk.astub` in the checker's main source directory. It will be automatically used by the checker. You can also supply `.astub` files in that directory for other libraries. You should list those other libraries in a `@StubFiles` annotation on the checker's main class, so that they will also be automatically used.

30.10 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`.

If your checker's source code is within a fork of the Checker Framework repository, then you can copy the testing infrastructure used by some existing type system. Here are some of the tasks you will perform:

- Make sure `all-tests` tests the new checker.

30.11 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.

30.11.1 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.)
- `-AprintVerboseGenerics`: print more information about type parameters and wildcards when they appear in warning messages. This produces a subset of the information than `-AprintAllQualifiers` does.
- `-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.

30.11.2 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.

30.11.3 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 30.6).
- `-AshowInferenceSteps`: print debugging information about intermediate steps in method type argument inference (as performed by `DefaultTypeArgumentInference`).

30.11.4 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. Note that this argument cannot be included in a file containing command-line arguments passed to the compiler using the @argfile syntax.
  Example usage: `-AoutputArgsToFile=$HOME/scriptfile`

30.11.5 Visualizing the dataflow graph

- `-Aflowdotdir=`*`somedir`*: Specify directory for `.dot` files visualizing the CFG. Shorthand for `-Acfgviz=org.checkerframework.dataflow.cfg.DOTCFGVisualizer,outdir=`*`somedir`*. The directory must already exist.
- `-Averbosecfg`: Enable additional output in the CFG visualization. Equivalent to passing `verbose` to `cfgviz`, e.g. as in `-Acfgviz=MyVisualizer,verbose`
- `-Acfgviz=`*`VizClassName`*`[,`*`opts`*`,...]`: Mechanism to visualize the control flow graph (CFG) of all the methods and code fragments analyzed by the dataflow analysis (Section 30.8). The graph also contains information about flow-sensitively refined types of various expressions at many program points.

The argument is a comma-separated sequence of values or key-value pairs. The first argument is the fully-qualified name of the `org.checkerframework.dataflow.cfg.CFGVisualizer` implementation that should be used. The remaining values or key-value pairs are passed to `CFGVisualizer.init`.

You can visualize `.dot` graph files with the Graphviz program. For example, to convert a `.dot` file to PDF:

```
dot -Tpdf -o myfile.pdf myfile.dot
```

30.11.6 Miscellaneous debugging options

- `-AresourceStats`: Whether to output resource statistics at JVM shutdown.

30.11.7 Examples

The following example demonstrates how these options are used:

```
$ javac -processor org.checkerframework.checker.interning.InterningChecker \
    docs/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
docs/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
```

30.11.8 Using an external debugger

You can use any standard debugger to observe the execution of your checker. 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/checker/dist/checker.jar" \
    -J-Xdebug -J-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005 \
    -processor org.checkerframework.checker.nullness.NullnessChecker \
    src/sandbox/FileToCheck.java
```

30.12 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 diagram will appear in directory `docs/manual/figures/`; see its `README` file for tips.

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 the appropriate part of Section 25.4.3 And add the file to `docs/manual/Makefile`. Keep the 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:`". Figure labels should start with "fig-*checkername*" and not with "fig:". These conventions are for the benefit of the Hevea program that produces the HTML version of the manual. Use `\begin{figure}` for all figures, including those whose content is a table, in order to have a single consistent numbering for all figures.

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. The documentation for any annotation should include an example use of the annotation. Also ensure that the Javadoc links back to the manual, using the `@checker_framework.manual` custom Javadoc tag.

30.13 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.

30.13.1 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`.

30.13.2 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. 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 30.6.1.

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://mail.openjdk.java.net/pipermail/compiler-dev/` (subscribe at `http://mail.openjdk.java.net/mailman/listinfo/compiler-dev`).

30.14 Integrating a checker with the Checker Framework

To integrate a new checker with the Checker Framework release, perform the following:

- Make sure `check-compilermsgs` and `check-purity` run without warnings or errors.

Chapter 31

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:

- Android Gradle Plugin (Section 31.1)
- Ant (Section 31.2)
- Eclipse (Section 31.3)
- Gradle (Section 31.4)
- IntelliJ IDEA (Section 31.5)
- javac (Section 31.6)
- Maven (Section 31.7)
- NetBeans (Section 31.8)
- tIDE (Section 31.9)

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 31.6. 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.

All examples in this chapter are in the public domain, with no copyright nor licensing restrictions.

31.1 Android Studio 3.0 and the Android Gradle Plugin 3.0

Android Studio 3.0 and Android Gradle Plugin 3.0.0 support type annotations. (See `https://developer.android.com/studio/write/java8-support.html` for more details.) This section explains how to configure your Andriod project to use the Checker Framework. All the changes should be made to the module's `build.gradle` file — not the app's `build.gradle` file.

1. In your module's `build.gradle` file, set the source and target compatiblity to `JavaVersion.VERSION_1_8`

```
android {
    ...
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
}
```

2. Add a build variant for running checkers.

```
android {
    ...
      buildTypes {
      ...
        checkTypes {
            javaCompileOptions.annotationProcessorOptions.
                    classNames.add("org.checkerframework.checker.nullness.NullnessChecker")
            // You can pass options like so:
            // javaCompileOptions.annotationProcessorOptions.arguments.put("warns", "")
        }
    }
}
```

3. Add a dependency configuration for the annotated JDK:

```
configurations {
    checkerFrameworkAnnotatedJDK {
        description = 'a copy of JDK classes with Checker Framework type qualifers inserted'
    }
}
```

4. Declare the Checker Framework dependencies:

```
dependencies {
    ... existing dependencies...
    ext.checkerFrameworkVersion = '2.5.1'
    implementation "org.checkerframework:checker-qual:${checkerFrameworkVersion}"
    annotationProcessor "org.checkerframework:checker:${checkerFrameworkVersion}"
    checkerFrameworkAnnotatedJDK "org.checkerframework:jdk8:${checkerFrameworkVersion}"
}
```

5. Direct all tasks of type `JavaCompile` used by the CheckTypes build variant to use the annotated JDK.

```
gradle.projectsEvaluated {
    tasks.withType(JavaCompile).all { compile ->
        if (compile.name.contains("CheckTypes")) {
            compile.options.compilerArgs += [
                    "-Xbootclasspath/p:${configurations.checkerFrameworkAnnotatedJDK.asPath}"
            ]
        }
    }
}
```

6. To run the checkers, build using the checkTypes variant.

```
gradlew checkTypes
```

31.2 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.

31.2.1 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.

31.3 Eclipse

You need to run the Checker Framework via Ant or via another build tool, rather than by supplying the `-processor` command-line option to the ejc compiler, which is also known as eclipsec. 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.)

31.3.1 Using an Ant task

Add an Ant target as described in Section 31.2. 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.

31.3.2 Troubleshooting Eclipse

Eclipse issues an "Unhandled Token in @SuppressWarnings" warning if you write a `@SuppressWarnings` annotation containing a string that does not know about. Unfortunately, Eclipse hard-codes this list.

To eliminate the warnings: disable all "Unhandled Token in @SuppressWarnings" warnings in Eclipse. Look under the menu headings Java → Compiler → Errors/Warnings → Annotations → Unhandled Token in '@SuppressWarnings', and set it to ignore.

31.4 Gradle

This section describes how to run a checker on a project that uses the Gradle build system.

1. Copy the file `checkerframework.gradle` to your project.
2. Customize the file, such as setting which type-checkers are run.
3. In your main buildfile, add the lines

```
// Checker Framework pluggable type-checking
apply from: "checkerframework.gradle"
```

Note: Gradle 3.4 has a bug where it does not run annotation processors, even if the `-processor` flag is passed. To work around this, install a later version of gradle. Or, run `gradle wrapper -gradle-version 4.5` (or any version 3.5 or later) and then always use `./gradlew` where you would have used `gradle`.

31.5 IntelliJ IDEA

To run a checker within IntelliJ:

1. Set the language level for your project to 8. To do so, go to the "Project Structure" menu via (File>Project Structure) or (Ctrl-Alt-Shift-S), and set the "Project language level" field in the "Project" sub-menu to 8.
   Note: under MacOS, the menu is "Preferences" rather than "Project Structure".
2. Add the Checker Framework libraries. In this same "Project Structure" menu, navigate to the "Libraries" sub-menu. Click on the green "+" that appears in this menu, select "Java" in the resulting pop-down menu, select `$CHECKERFRAMEWORK/checker/dist/checker.jar` in the resulting menu, and click "OK".
3. Add the annotated JDK library using the following method. Go to the "Settings" menu via (File>Settings) or (Ctrl-Alt-S) and navigate to the "Java Compiler" sub-menu via (Build, Execution, Deployment>Compiler>Java Compiler). Add the following to the field "Additional command line parameters":
   `-Xbootclasspath/p:"$CHECKERFRAMEWORK/checker/dist/jdk8.jar"`
4. Create an annotation profile and specify which checkers to run. Directly under the "Java Compiler" sub-menu is the "Annotation Processors" sub-menu. Navigate here and click on the green "+" to create a new Annotation Processor profile. Give this profile a unique name of your choosing that does not contain special characters or spaces. Select the modules you would like to use checkers with from under the "Default" profile and move them to your new profile via the button directly right of the "-" button.
   Now select your new profile and check the "Enable annotation processing" option. Select the "Processor path" radio button and enter `$CHECKERFRAMEWORK/checker/dist/checker.jar` into the field to the right of it. To add a checker to be run during compilation, click on the green "+" in the "Processor FQ Name" section and write that checker's fully-qualified name in the resulting text field, and press the "ENTER" key. (The fully-qualified name

is usually an instantiation of `org.checkerframework.checker.`*`CHECKER`*`.`*`CHECKER`*`Checker`, replacing *CHECKER* by the checker you want to run; an example would be `org.checkerframework.checker.nullness.NullnessChecke`

Now, when you compile your code, the checker will be run.

It is necessary to manually inform the IDE via a plugin if an annotation system adds any dependencies beyond those that normally exist in Java. For information about the extension points, see `https://youtrack.jetbrains.com/issue/IDEA-159286`.

31.6 Javac compiler

To perform pluggable type-checking, run the `javac` compiler with the Checker Framework on the classpath. There are three ways to achieve this. 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-2.5.1/checker/bin` to your path, *before* any other directory that contains a `javac` executable.
  If you are using the bash shell, a way to do this is to add the following to your `~/.profile` (or alternately `~/.bash_profile` or `~/.bashrc`) file:

  ```
  export CHECKERFRAMEWORK=${HOME}/checker-framework-2.5.1
  export PATH=${CHECKERFRAMEWORK}/checker/bin:${PATH}
  ```

  then log out and back in to ensure that the environment variable setting takes effect.
  Now, whenever you run `javac`, you will use the "Checker Framework compiler". It is exactly the same as the OpenJDK compiler, with one small differences: it includes the Checker Framework jar file on its classpath.
- 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=${HOME}/checker-framework-2.5.1
  alias javacheck='$CHECKERFRAMEWORK/checker/bin/javac'
  ```

- 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=${HOME}/checker-framework-2.5.1
  alias javacheck='java -jar "$CHECKERFRAMEWORK/checker/dist/checker.jar"'

  # Windows
  set CHECKERFRAMEWORK = C:\Program Files\checker-framework-2.5.1\
  doskey javacheck=java -jar "%CHECKERFRAMEWORK%\checker\dist\checker.jar" $*
  ```

  (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`.)

31.7 Maven

If you use the Maven tool, then you can enable Checker Framework checkers by following the instructions below.

See the directory `docs/examples/MavenExample/` for examples of the use of Maven build files that run a checker. These examples can be used to verify that Maven is correctly downloading the Checker Framework from Maven Central Repository and executing it.

Please note that the `-AoutputArgsToFile` command-line option (see Section 30.11.4) and shorthands for built-in checkers (see Section 2.2.5) are not available when following these instructions. Both these features are available only when a checker is launched via `checker.jar` such as when `$CHECKERFRAMEWORK/checker/bin/javac` is run. The instructions in this section bypass `checker.jar` and cause the compiler to run a checker as an annotation processor directly.

1. Declare a dependency on the Checker Framework artifacts, either from Maven Central or from a local directory. Find the existing `<dependencies>` section and add the following new `<dependency>` items:

   (a) To obtain artifacts from Maven Central:

   ```
   <dependencies>
     ... existing <dependency> items ...

     <!-- Annotations from the Checker Framework: nullness, interning, locking, ... -->
     <dependency>
       <groupId>org.checkerframework</groupId>
       <artifactId>checker-qual</artifactId>
       <version>2.5.1</version>
     </dependency>
     <dependency>
       <groupId>org.checkerframework</groupId>
       <artifactId>jdk8</artifactId>
       <version>2.5.1</version>
     </dependency>
   </dependencies>
   ```

   Periodically update to the most recent version, to obtain the latest bug fixes and new features:

   ```
   mvn versions:use-latest-versions -Dincludes="org.checkerframework:*"
   ```

   (b) To use a local version of the Checker Framework (for example, one that you have built from source):

   ```
   <dependencies>
     ... existing <dependency> items ...

     <!-- Annotations from the Checker Framework: nullness, interning, locking, ... -->
     <dependency>
       <groupId>org.checkerframework</groupId>
       <artifactId>checker-qual</artifactId>
       <version>2.5.1</version>
       <scope>system</scope>
       <systemPath>${env.CHECKERFRAMEWORK}/checker/dist/checker-qual.jar</systemPath>
     </dependency>
     <dependency>
       <groupId>org.checkerframework</groupId>
       <artifactId>checker</artifactId>
       <version>2.5.1</version>
       <scope>system</scope>
       <systemPath>${env.CHECKERFRAMEWORK}/checker/dist/checker.jar</systemPath>
     </dependency>
     <!-- The annotated JDK to use. -->
     <dependency>
       <groupId>org.checkerframework</groupId>
   ```

```
        <artifactId>jdk8</artifactId>
        <version>2.5.1</version>
        <scope>system</scope>
        <systemPath>${env.CHECKERFRAMEWORK}/checker/dist/jdk8.jar</systemPath>
      </dependency>
    </dependencies>
```

Periodically update to the most recent version, to obtain the latest bug fixes and new features:

```
mvn versions:use-latest-versions -Dincludes="org.checkerframework:*"
```

2. Use Maven properties to hold the locations of the annotated JDK. It was declared as Maven dependencies above. To set the value of this property automatically, you will use the Maven Dependency plugin.
   First, create the property in the `properties` section of the POM:

```
<properties>
  <!-- These properties will be set by the Maven Dependency plugin -->
  <annotatedJdk>${org.checkerframework:jdk8:jar}</annotatedJdk>
</properties>
```

   Change the reference to the `maven-dependency-plugin` within the `<plugins>` section, or add it if it is not present.

```
<plugin>
  <!-- This plugin will set properties values using dependency information -->
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-dependency-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>properties</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

3. Direct the Maven compiler plugin to use the desired checkers. Change the reference to the `maven-compiler-plugin` within the `<plugins>` section, or add it if it is not present.
   For example, to use the `org.checkerframework.checker.nullness.NullnessChecker`:

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.6.1</version>
  <configuration>
    <source>1.8</source>
    <target>1.8</target>
    <compilerArguments>
      <Xmaxerrs>10000</Xmaxerrs>
      <Xmaxwarns>10000</Xmaxwarns>
    </compilerArguments>
    <annotationProcessorPaths>
      <path>
        <groupId>org.checkerframework</groupId>
        <artifactId>checker</artifactId>
        <version>2.5.1</version>
      </path>
    </annotationProcessorPaths>
    <annotationProcessors>
      <!-- Add all the checkers you want to enable here -->
      <annotationProcessor>org.checkerframework.checker.nullness.NullnessChecker</annotationProcessor>
    </annotationProcessors>
    <compilerArgs>
      <arg>-AprintErrorStack</arg>
      <!-- location of the annotated JDK, which comes from a Maven dependency -->
```

```
        <arg>-Xbootclasspath/p:${annotatedJdk}</arg>
        <!-- Uncomment the following line to turn type-checking warnings into errors. -->
        <!-- <arg>-Awarns</arg> -->
      </compilerArgs>
    </configuration>
  </plugin>
```

Now, building with Maven will run the checkers during compilation.

Notice that using this approach, no external setup is necessary, so your Maven build should be reproducible on any server.

If you want to allow Maven to compile your code without running the checkers, you may want to move the declarations above to within a Maven profile, so that the checkers run only if the profile was enabled.

31.8 NetBeans

There are two approaches to running a checker in NetBeans: via modifying the project properties, or via a custom ant target.

Note: The 'compile and save' action in NetBeans 8.1 automatically runs the Checker Framework whereas this functionality has not yet been incorporated into NetBeans 8.2. Additionally, JDK annotations are currently unavailable on Netbeans 8.1 and 8.2.

31.8.1 Adding a checker via the Project Properties window

1. Add the Checker Framework libraries to your project's library. First, right click on the project in the Projects panel, and select "Properties" in the drop-down menu. Then, in the "Project Properties" window, navigate to the "Libraries" tab.
2. Add `checker-qual.jar` to the compile-time libraries. To do so, select the "Compile" tab, click the "Add JAR/Folder" button on the right and browse to add `$CHECKERFRAMEWORK/checker/dist/checker-qual.jar`.
3. Add `checker.jar` to the processor-path libraries. To do so, select the "Processor" tab, click the "Add JAR/Folder" button on the right and browse to add `$CHECKERFRAMEWORK/checker/dist/checker.jar`.
4. Enable annotation processor underlining in the editor. Go to "Build>Compiling" and check the box "Enable Annotation Processing," and under that, "Enable Annotation Processing in Editor."
5. Add the checker to run, by clicking "Add" next to the box labeled "Annotation Processors" and enter the fully qualified name of the checker (for example, `org.checkerframework.checker.nullness.NullnessChecker`) and click "OK" to add.

The selected checker should be run on the project either on a save (if Compile on Save is enabled), or when the project is built, and annotation processor output will appear in the editor.

31.8.2 Adding a checker via an ant target

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. Override the `-init-macrodef-javac-with-processors` target to use `jsr308.javac` instead of `javac` and to run the checker. In this example, a nullness checker is run:

```
<target depends="-init-ap-cmdline-properties" if="ap.supported.internal"
      name="-init-macrodef-javac-with-processors">
  <echo message = "${checkerframework}"/>
  <macrodef name="javac" uri="http://www.netbeans.org/ns/j2se-project/3">
      <attribute default="${src.dir}" name="srcdir"/>
      <attribute default="${build.classes.dir}" name="destdir"/>
      <attribute default="${javac.classpath}" name="classpath"/>
      <attribute default="${javac.processorpath}" name="processorpath"/>
      <attribute default="${build.generated.sources.dir}/ap-source-output" name="apgeneratedsrcdir"/>
      <attribute default="${includes}" name="includes"/>
      <attribute default="${excludes}" name="excludes"/>
      <attribute default="${javac.debug}" name="debug"/>
      <attribute default="${empty.dir}" name="sourcepath"/>
      <attribute default="${empty.dir}" name="gensrcdir"/>
      <element name="customize" optional="true"/>
      <sequential>
          <property location="${build.dir}/empty" name="empty.dir"/>
          <mkdir dir="${empty.dir}"/>
          <mkdir dir="@{apgeneratedsrcdir}"/>
          <jsr308.javac debug="@{debug}" deprecation="${javac.deprecation}"
                  destdir="@{destdir}" encoding="${source.encoding}"
                  excludes="@{excludes}" fork="${javac.fork}"
                  includeantruntime="false" includes="@{includes}"
                  source="${javac.source}" sourcepath="@{sourcepath}"
                  srcdir="@{srcdir}" target="${javac.target}"
                  tempdir="${java.io.tmpdir}">
              <src>
                  <dirset dir="@{gensrcdir}" erroronmissingdir="false">
                      <include name="*"/>
                  </dirset>
              </src>
              <classpath>
                  <path path="@{classpath}"/>
              </classpath>
              <compilerarg line="${endorsed.classpath.cmd.line.arg}"/>
              <compilerarg line="${javac.profile.cmd.line.arg}"/>
              <compilerarg line="${javac.compilerargs}"/>
              <compilerarg value="-processorpath"/>
              <compilerarg path="@{processorpath}:${empty.dir}"/>
              <compilerarg line="${ap.processors.internal}"/>
              <compilerarg line="${annotation.processing.processor.options}"/>
              <compilerarg value="-s"/>
              <compilerarg path="@{apgeneratedsrcdir}"/>
              <compilerarg line="${ap.proc.none.internal}"/>
              <compilerarg line="-processor org.checkerframework.checker.nullness.NullnessChecker"/>
              <compilerarg line="-Xmaxerrs 10000"/>
              <compilerarg line="-Xmaxwarns 10000"/>
              <customize/>
          </jsr308.javac>
      </sequential>
  </macrodef>
</target>
<target name="-post-jar">
</target>
```

When Build and Clean Project is used, the output of the checker will now appear in the build console. However, annotation processor output will not appear in the editor.

31.9 tIDE

tIDE, an open-source Java IDE, supports the Checker Framework. You can download it from `https://sourceforge.net/projects/tide/`.

31.10 Type inference tools

There are two different tasks that are commonly called "type inference". You are probably interested in type inference to annotate a program's source code; see Section 28.2. This section explains the two different varieties of 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 28.2): 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 28.2). 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.

Chapter 32

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:

32.1: Motivation for pluggable type-checking
32.1.1: I don't make type errors, so would pluggable type-checking help me?
32.1.2: When should I use type qualifiers, and when should I use subclasses?

32.2: Getting started
32.2.1: How do I get started annotating an existing program?
32.2.2: Which checker should I start with?
32.2.3: How can I join the checker-framework-dev mailing list?

32.3: Usability of pluggable type-checking
32.3.1: Are type annotations easy to read and write?
32.3.2: Will my code become cluttered with type annotations?
32.3.3: Will using the Checker Framework slow down my program? Will it slow down the compiler?
32.3.4: How do I shorten the command line when invoking a checker?
32.3.5: Method pre-condition contracts, including formal parameter annotations, make no sense for public methods.

32.4: How to handle warnings
32.4.1: What should I do if a checker issues a warning about my code?
32.4.2: What does a certain Checker Framework warning message mean?
32.4.3: Can a pluggable type-checker guarantee that my code is correct?
32.4.4: What guarantee does the Checker Framework give for concurrent code?
32.4.5: How do I make compilation succeed even if a checker issues errors?
32.4.6: Why does the checker always say there are 100 errors or warnings?
32.4.7: Why does the Checker Framework report an error regarding a type I have not written in my program?
32.4.8: How can I do run-time monitoring of properties that were not statically checked?

32.5: False positive warnings
32.5.1: What is a "false positive" warning?
32.5.2: How can I improve the Checker Framework to eliminate a false positive warning?
32.5.3: Why doesn't the Checker Framework infer types for fields and method return types?

32.5.4: Why doesn't the Checker Framework track relationships between variables?
32.5.5: Why isn't the Checker Framework path-sensitive?

32.6: Syntax of type annotations
32.6.1: What is a "receiver"?
32.6.2: What is the meaning of an annotation after a type, such as `@NonNull Object @Nullable`?
32.6.3: What is the meaning of array annotations such as `@NonNull Object @Nullable []`?
32.6.4: What is the meaning of varargs annotations such as `@English String @NonEmpty ...`?
32.6.5: What is the meaning of a type qualifier at a class declaration?
32.6.6: Why shouldn't a qualifier apply to both types and declarations?
32.6.7: How do I annotate a fully-qualified type name?
32.6.8: What is the difference between type annotations and declaration annotations?

32.7: Semantics of type annotations
32.7.1: How can I handle typestate, or phases of my program with different data properties?
32.7.2: Why are explicit and implicit bounds defaulted differently?
32.7.3: Why are type annotations declared with `@Retention(RetentionPolicy.RUNTIME)`?

32.8: Creating a new checker
32.8.1: How do I create a new checker?
32.8.2: What properties can and cannot be handled by type-checking?
32.8.3: Why is there no declarative syntax for writing type rules?

32.9: Tool questions
32.9.1: How does pluggable type-checking work?
32.9.2: What classpath is needed to use an annotated library?
32.9.4: Is there a type-checker for managing checked and unchecked exceptions?

32.10: Relationship to other tools
32.10.1: Why not just use a bug detector (like FindBugs)?
32.10.2: How does the Checker Framework compare with Eclipse's Null Analysis?
32.10.3: How does the Checker Framework compare with the JDK's `Optional` type?
32.10.4: How does pluggable type-checking compare with JML?
32.10.5: Is the Checker Framework an official part of Java?
32.10.6: What is the relationship between the Checker Framework and JSR 305?
32.10.7: What is the relationship between the Checker Framework and JSR 308?

32.1 Motivation for pluggable type-checking

32.1.1 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.

32.1.2 Should I use pluggable types (type qualifiers) or Java subtypes?

In brief, use subtypes when you can, and use type qualifiers when you cannot use 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 runs every time you compile your code; by contrast, it is possible to forget to run the pluggable type-checker. However, sometimes type qualifiers are a better choice; here are some reasons:

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 write library annotations (see Chapter 29).

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. 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.

For more details, see Section 32.1.2.

32.2 Getting started

32.2.1 How do I get started annotating an existing program?

See Section 2.4.2.

32.2.2 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 25), 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.

Some of the third-party checkers (see Chapter 23, page 137) have known bugs that limit their usability. (Report the ones that affect you, so the developers will prioritize fixing them.)

32.2.3 How can I join the checker-framework-dev mailing list?

The `checker-framework-dev@googlegroups.com` mailing list is for Checker Framework developers. Anyone is welcome to join `checker-framework-dev`, after they have had several pull requests accepted.

Anyone is welcome to send mail to the `checker-framework-dev@googlegroups.com` mailing list — for implementation details it is generally a better place for discussions than the general `checker-framework-discuss@googlegroups.com` mailing list, which is for user-focused discussions.

Anyone is welcome to join `checker-framework-discuss@googlegroups.com` and send mail to it.

32.3 Usability of pluggable type-checking

32.3.1 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.

32.3.2 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`)

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.

32.3.3 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.

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

The compile options to javac can be long to type; for example, `javac -processor org.checkerframework.checker.nullness.NullnessChecker ...`. See Section 2.2.4 for a way to avoid the need for the `-processor` command-line option.

32.3.5 Method pre-condition contracts, including formal parameter annotations, make no sense for public methods

Some people go further and say that pre-condition contracts make no sense for any method. This objection is sometimes stated as, "A method parameter should never be annotated as `@NonNull`. A client could pass any value at all, so the method implementation cannot depend on the value being non-null. Furthermore, if a client passes an illegal value, it is the method's responsibility to immediately tell the client about the illegal value."

Here is an example that invalidates this general argument. Consider a binary search routine. Its specification requires that clients pass in a sorted array.

```
/** Return index of the search key, if it is contained it the sorted array a; otherwise ... */
int binarySearch(Object @Sorted [] a, Object key)
```

The `binarySearch` routine is fast — it runs in O(log n) time where n is the length of the array. If the routine had to validate that its input array is sorted, then it would run in O(n) time, negating all benefit of binary search. In other words, `binarySearch` should *not* validate its input!

The nature of a contract is that if the *caller* violates its responsibilities by passing bad values, then the *callee* is absolved of its responsibilities. It is polite for the callee to try to provide a useful diagnostic to the misbehaving caller (for example, by raising a particular exception quickly), but it is *not* required. In such a situation, the callee has the flexibility to do anything that is convenient.

In some cases a routine has a *complete* specification: the contract permits the caller to pass any value, and the callee is required to throw particular exceptions for particular inputs. This approach is common for public methods, but it is not required and is not always the right thing. As explained in section 2.4.3, even when a method has a complete specification, the annotations should indicate normal behavior: behavior that will avoid exceptions.

32.4 How to handle warnings and errors

32.4.1 What should I do if a checker issues a warning about my code?

For a discussion of this issue, see Section 2.4.6.

32.4.2 What does a certain Checker Framework warning message mean?

Read the error message first; sometimes that is enough to clarify it.

Search through this manual for the text of the warning message or for words that appear in it.

If nothing else explains it, then ask on the mailing list. Be sure to say what you think it means or what specific part does not make sense to you, and what you have already done to try to understand it.

32.4.3 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.

32.4.4 What guarantee does the Checker Framework give for concurrent code?

The Lock Checker (see Chapter 7) 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.

32.4.5 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.

32.4.6 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.

32.4.7 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. Note that an invocation of an impure method may cause the loss of all information that was determined via flow-sensitive type refinement; see Section 25.4.5.

32.4.8 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 `NullnessUtil.castNonNull` method (see Section 3.4.1) and the Regex Checker's `RegexUtil` class (see Section 11.2.4). But, it would be better to have more general support that does not require the user to explicitly insert method calls.

32.5 False positive warnings

32.5.1 What is a "false positive" warning?

A "false positive" is when the tool reports a potential problem, but the code is actually correct and will never violate the given property at run time.

The Checker Framework aims to be sound; that is, if the Checker Framework does not report any possible errors, then your code is correct.

Every sound tool suffers false positive errors. Wherever the Checker Framework issues an error, you can think of it as saying, "I can't prove this code is safe," but the code might be safe for some complex, tricky reason that is beyond the capabilities of its analysis.

If you are sure that the warning is a false positive, you have several options. Perhaps you just need to write annotations, especially on method signatures but perhaps within method bodies as well. Sometimes you can rewrite the code in a clearer way that the Checker Framework can verify, and that might be easier for people to understand, too. If these don't work, then you can suppress the warning (Section 2.4.6). You also might want to report the false positive in the Checker Framework issue tracker (Section 33.2), if it appears in real-world, well-written code. Finally, you could improve the Checker Framework to make it more precise, so that it does not suffer that false positive (see Section 32.5.2).

32.5.2 How can I improve the Checker Framework to eliminate a false positive warning?

As noted in Section 32.5.1, *every* sound analysis tool suffers false positives.

For any given false positive warning, it is theoretically possible to improve the Checker Framework to eliminate it. (But, it's theoretically impossible to eliminate all false positives. That is, there will always exist some programs that don't go wrong at run time but for which the Checker Framework issues a warning.)

Some improvements affect the implementation of the type system; they do not add any new types. Such an improvement is invisible to users, except that the users suffer fewer false positive warnings. This type of improvement to the type checker's implementation is often worthwhile.

Other improvements change the type system or add a new type system. Defining new types is a powerful way to improve precision, but it is costly too. A simpler type system is easier for users to understand, less likely to contain bugs, and more efficient.

By design, each type system in the Checker Framework has limited expressiveness. Our goal is to implement enough functionality to handle common, well-written real-world code, not to cover every possible situation.

When reporting bugs, please focus on realistic scenarios. We are sure that you can make up artificial code that stymies the type-checker! Those bugs aren't a good use of your time to report nor the maintainers' time to evaluate and fix. When reporting a bug, it's very helpful to minimize it to give a tiny example that is easy to evaluate and fix, but please also indicate how it arises in real-world, well-written code.

32.5.3 Why doesn't the Checker Framework infer types for fields and method return types?

Consider the following code. A programmer can tell that all three invocations of `format` are safe — they never suffer an `IllegalFormatException` exception at run time:

```
class MyClass {

  final String field = "%d";

  String method() {
    return "%d";
  }

  void method m() {
```

```
    String local = "%d";
    String.format(local, 5);    // Format String Checker verifies that call is safe
    String.format(field, 5);    // Format String Checker warns that call might not be safe
    String.format(method(), 5); // Format String Checker warns that call might not be safe
  }
}
```

However, the Format String Checker can only verify the first call. It issues a false positive warning about the second and third calls.

The Checker Framework can verify all three calls, with no false positive warnings, if you annotate the type of `field` and the return type of `method` as `@Format(INT)`.

By default, the Checker Framework infers types for local variables (Section 25.4), but not for fields and method return types. (The Checker Framework includes a whole-program type inference tool that infers field and method return types; see Section 28.3.) There are several reasons for this design choice.

Separation of specification from implementation The designer of an API makes certain promises to clients; these are codified in the API's specification or contract. The implementation might return a more specific type today, but the designer does not want clients to depend on that. For example, a string might happen to be a regular expression because it contains no characters with special meaning in regexes, but that is not guaranteed to always be true. It's better for the programmer to explicitly write the intended specification.

Separate compilation To infer types for a non-final method, it is necessary to examine every overriding implementation, so that the method's return type annotation is compatible with all values that are returned by any overriding implementation. In general, examining all implementations is impossible, because clients may override the method. When possible, it is inconvenient to provide all that source code, and it would slow the type-checker down.

A related issue is that determining which values can be returned by a method *m* requires analyzing *m*'s body, which requires analyzing all the methods called by *m*, and so forth. This quickly devolves to analyzing the whole program. Determining all possible values assigned to a field is equally hard.

Type-checking is modular — it works on one procedure at a time, examining that procedure and the specifications (not implementations) of methods that it calls. Therefore, type-checking is fast and can work on partial programs. It is undesirable to change type-checking into a whole-program analysis.

Order of compilation When the compiler is called on class `Client` and class `Library`, the programmer has no control over which class is analyzed first. When the first class is compiled, it has access only to the signature of the other class. Therefore, a programmer would see inconsistent results depending on whether `Client` was compiled first and had access only to the declared types of `Library`, or the `Library` was compiled first and the compiler refined the types of its methods and fields before `Client` looked them up.

Consistent behavior with or without pluggable type-checking The `.class` files produced with or without pluggable type-checking should specify the same types for all public fields and methods. If pluggable type-checking changed the those types, then users would be confused. Depending on how a library was compiled, pluggable type-checking of a client program would give different results.

32.5.4 Why doesn't the Checker Framework track relationships between variables?

The Checker Framework estimates the possible run-time value of each variable, as expressed in a type system. In general, the Checker Framework does estimate relationships between two variables, except for specific relationships listed in Section 25.5.

For example, the Checker Framework does not track which variables are equal to one another. The Nullness Checker issues a warning, "dereference of possibly-null reference y", for expression `y.toString()`:

```
void nullnessExample1(@Nullable Object x) {
    Object y = x;
    if (x != null) {
```

```
    System.out.println(y.toString());
  }
}
```

Code that checks one variable and then uses a different variable is confusing and is often considered poor style.

The Nullness Checker is able to verify the correctness of a small variant of the program, thanks to flow-sensitive type refinement (Section 25.4):

```
void nullnessExample12(@Nullable Object x) {
  if (x != null) {
    Object y = x;
    System.out.println(y.toString());
  }
}
```

The difference is that in the first example, nothing was known about `x` at the time `y` was set to it, and so the Nullness Checker recorded no facts about `y`. In the second example, the Nullness Checker knew that `x` was non-null when `y` was assigned to it.

In the future, the Checker Framework could be enriched by tracking which variables are equal to one another, a technique called "copy propagation".

This would handle the above examples, but wouldn't handle other examples. For example, the following code is safe:

```
void requiresPositive(@Positive int arg) {}

void intExample1(int x) {
  int y = x*x;
  if (x > 0) {
    requiresPositive(y);
  }
}

void intExample2(int x) {
  int y = x*x;
  if (y > 0) {
    requiresPositive(x);
  }
}
```

However, the Index Checker (Chapter 8, page 70), which defines the `@Positive` type qualifier, issues warnings saying that it cannot prove that the arguments are `@Positive`.

A slight variant of `intExample1` can be verified:

```
void intExample1a(int x) {
  if (x > 0) {
    int y = x*x;
    requiresPositive(y);
  }
}
```

No variant of `intExample2` can be verified. It is not worthwhile to make the Checker Framework more complex and slow by tracking rich properties such as arbitrary arithmetic.

As another example of a false positive warning due to arbitrary arithmetic properties, consider the following code:

```
void falsePositive2(int arg) {
  Object o;
  if (arg * arg >= arg) { // always true!
    o = new Object();
  }
  o.toString();  // Nullness Checker issues a false positive warning
}
```

32.5.5 Why isn't the Checker Framework path-sensitive?

The Checker Framework is not path-sensitive. That is, it maintains one estimate for each variable, and it assumes at every branch (such as `if` statement) that every choice could be taken.

In the following code, there are two `if` statements.

```
void falsePositive1(boolean b) {
  Object o;
  if (b) {
    o = new Object();
  }
  if (b) {
    o.toString();  // Nullness Checker issues a false positive warning
  }
}
```

In general, if code has two `if` statements in succession, then there are 4 possible paths through the code: [true, true], [true, false], [false, true], and [false, false]. However, for this code only two of those paths are feasible: namely, [true, true] and [false, false].

The Checker Framework is not path-sensitive, so it issues a warning.

The lack of path-sensitivity can be viewed as special case of the fact that the Checker Framework maintains a single estimate for each variable value, rather than tracking relationships between multiple variables (Section 32.5.4).

Making the Checker Framework path-sensitive would make it more powerful, but also much more complex and much slower. We have not yet found this necessary.

32.6 Syntax of type annotations

There is also a separate FAQ for the type annotations syntax (`https://checkerframework.org/jsr308/jsr308-faq.html`).

32.6.1 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`.

32.6.2 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 32.6.3).

32.6.3 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 have inconsistent semantics for annotations on array and varargs elements. Their semantics is unfortunate and confusing; developers should convert their code to use type annotations instead. Section 27.1.1 explains how the Checker Framework handles declaration annotations in the meanwhile.

32.6.4 What is the meaning of varargs annotations such as `@English String @NonEmpty ...`?

Varargs annotations are treated similarly to array annotations. (A way to remember this is that when you write a varargs formal parameter such as `void method(String... x) {}`, the Java compiler generates a method that takes an array of strings; whenever your source code calls the method with multiple arguments, the Java compiler packages them up into an array before calling the method.)

Either of these annotations

```
void method(String @NonEmpty [] x) {}
void method(String @NonEmpty ... x) {}
```

applies to the array: the method takes a non-empty array of strings, or the varargs list must not be empty.

Either of these annotations

```
void method(@English String [] x) {}
void method(@English String ... x) {}
```

. applies to the element type. The annotation documents that the method takes an array of English strings.

32.6.5 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`.

32.6.6 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 7.6.1, it means two different things when applied to a field or a method. To reduce confusion and increase expressiveness, the Lock Checker (see Chapter 7) uses the `@Holding` annotation for one of these meanings, rather than overloading `@GuardedBy` with two distinct meanings.

A final example of a type-and-declaration annotation is some `@Nullable` or `@NonNull` annotations that are intended to work both with modern tools that process type annotations and with old tools that were written before Java had type annotations. Such type-and-declaration annotations were a temporary measure, intended to be used until the tool supported Java 8, and should not be necessary any longer.

32.6.7 How do I annotate a fully-qualified type name?

If you write a fully-qualified type name in your program, then the Java language requires you to write a type annotation on the simple name part, such as

```
entity.hibernate. @Nullable User x;
```

If you try to write the type annotation before the entire fully-qualified name, such as

```
@Nullable entity.hibernate.User x;  // illegal Java syntax
```

then you will get an error like one of the following:

```
error: scoping construct for static nested type cannot be annotated
error: scoping construct cannot be annotated with type-use annotation
```

32.6.8 What is the difference between type annotations and declaration annotations?

Java has two distinct varieties of annotation: type annotations and declaration annotations.

A **type annotation** can be written on any use of a **type**. It conceptually creates a new, more specific type. That is, it describes what values the type represents.

As an example, the `int` type contains these values: ..., -2, -1, 0, 1, 2, ...
The `@Positive int` type contains thes values: 1, 2, ...
Therefore, `@Positive int` is a subtype of `int`.

A **declaration annotation** can be written on any **declaration** (a class, method, or variable). It describes the thing being declared, but does not describe run-time values. Here are examples of declaration annotations:

```
@Deprecated    // programmers should not use this class
class MyClass { ... }

@Override      // this method overrides a method in a supertype
void myMethod() { ... }

@SuppressWarnings(...) // compiler should not warn about the initialization expr
int myField = INITIALIZATION-EXPRESSION;
```

Here are examples that use both a declaration annotation and a type annotation:

```
@Override
@Regex String getPattern() { ... }

@GuardedBy("myLock")
@NonNull String myField;
```

Note that the type annotation describes the value, and the declaration annotation says something about the method or use of the field.

As a matter of style, declaration annotations are written on their own line, and type annotations are written directly before the type, on the same line.

32.7 Semantics of type annotations

32.7.1 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 137), but they have not been as extensively tested.

32.7.2 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 11.1.

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`.

32.7.3 Why are type annotations declared with `@Retention(RetentionPolicy.RUNTIME)`?

Annotations such as `@NonNull` are declared with `@Retention(RetentionPolicy.RUNTIME)`. In other words, these type annotations are available to tools at run time. Such run-time tools could check the annotations (like an `assert` statement), type-check dynamically-loaded code, check casts and `instanceof` operations, resolve reflection more precisely, or other tasks that we have not yet thought of. Not many such tools exist today, but the annotation designers wanted to accommodate them in the future.

`RUNTIME` retention has negligible costs (no run-time dependency, minimal increase in heap size).

For the purpose of static checking at compile time, `CLASS` retention would be sufficient. Note that `SOURCE` retention would not be sufficient, because of separate compilation: when type-checking a class, the compiler needs to read the annotations on libraries that it uses, and separately-compiled libraries are available to the compiler only as class files.

32.8 Creating a new checker

32.8.1 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 30 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.

```
@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.

32.8.2 What properties can and cannot be handled by type-checking?

In theory, any property about a program can be expressed and checked within a type system. In practice, types are a good choice for some properties and a bad choice for others.

A type expresses the set of possible values for an expression. Therefore, types are a good choice for any property that is about variable values or provenance.

Types are a poor choice for expressing properties about timing, such as that action B will happen within 10 milliseconds of action A. Types are not good for verifying the results of calculations; for example, they could ensure that code always call an `encrypt` routine in the appropriate places, but not that the `encrypt` routine is correctly implemented. Types are not a good solution for preventing infinite loops, except perhaps in special cases.

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

A type system implementer can declaratively specify the type qualifier hierarchy (Section 30.4.2) and the type introduction rules (Section 30.7.1). However, the Checker Framework uses a procedural syntax for specifying type-checking rules (Section 30.6). 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.

32.9 Tool questions

32.9.1 How does pluggable type-checking work?

The Checker Framework enables you to define a new type system. It finds errors, or guarantees their absence, by performing type-checking that is similar to that already performed by the Java compiler.

Type-checking examines each statement of your program in turn, one at a time.

- Expressions are processed bottom-up. Given types for each sub-expression, the type-checker determines whether the types are legal for the expression's operator and determines the type of the expression.
- An assignment is legal if the type of the right-hand side is a subtype of the declared type of the left-hand side.
- At a method call, the arguments are legal if they can be assigned to the formal parameters (this is called a "pseudo-assignment" and it follows the normal rules for assignment). The type of the method call is the declared type of the return type, where the method is declared. If the method declaration is not annotated, then a default annotation is used.

- Suppose that method `Sub.m` overrides method `Super.m`. The return type of `Sub.m` must be equal to or a subtype of the return type of `Super.m` (this is called "covariance"). The type of formal parameter *i* of `Sub.m` must be equal to or a *super*type of the type of formal parameter *i* in `Super.m` (this is called "contravariance").

32.9.2 What classpath is needed to use an annotated library?

Suppose that you distribute a library, which contains Checker Framework annotations such as `@Nullable`. This enables clients of the library to use the Checker Framework to type-check their programs. To do so, they must have the Checker Framework annotations on their classpath, for instance by using the Checker Framework Compiler.

Clients who do not wish to perform pluggable type-checking do not need to have the Checker Framework annotations (`checker-qual.jar`) in their classpath, either when compiling or running their programs.

The JVM does not issue a link error if an annotation is not found when a class is loaded. (By contrast, the JVM does issue a link error if a superclass, or a parameter/return/field type, is not found.) Likewise, there is no problem compiling against a library even if the library's annotations are not on the classpath. These are properties of Java, and are not specific to the Checker Framework's annotations.

32.9.3 Why do `.class` files contain more annotations than the source code?

You may notice annotations such as `@Pure` and `@SideEffectFree` in compiled `.class` files even though the source code containes no annotations. These annotations are propagated from annotated libraries, such as the JDK, to your code.

When an overridden method has a side effect annotation, the overriding method must have one too. If you do not write one explicitly, the Checker Framework could issue a warning, or it could automatically add the missing annotation. It does the latter, for annotations declared with the `@InheritedAnnotation` meta-annotation.

In addition, defaulted and inferred type qualifiers are written into `.class` files; see Section 25.3.

32.9.4 Is there a type-checker for managing checked and unchecked exceptions?

It is possible to annotate exception types, and any type-checker built on the Checker Framework enforces that type annotations are consistent between `throw` statements and `catch` clauses that might catch them.

The Java compiler already enforces that all checked exceptions are caught or are declared to be passed through, so you would use annotations to express other properties about exceptions.

Checked exceptions are an example of a "type and effect" system, which is like a type system but also accounts for actions/behaviors such as side effects. The GUI Effect Checker (Chapter 16, page 107) is a type-and-effect system that is distributed with the Checker Framework.

32.10 Relationship to other tools

32.10.1 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 33.6. 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].

32.10.2 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 25) is more precise: it does a deeper semantic analysis, so it issues fewer false positives than Eclipse. Eclipse's nullness analysis is missing many features that the Checker Framework supports, such as handling of map keys, partially-initialized objects, method pre- and post-conditions, polymorphism, and a powerful dataflow analysis. These are essential for practical verification of real-world code without poor precision. Furthermore, Eclipse by default ignores unannotated code (even unannotated parameters within a method that contains other annotations). As a result, Eclipse is more useful for bug-finding than for verification, and that is what the Eclipse documentation recommends.

Eclipse assumes by default that all code is multi-threaded, which cripples its local type inference. (This default can be turned off, however.) 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 32.4.4).

The Checker Framework builds on javac, so it is easier to run in integration scripts or in environments where not all developers have installed 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 add it to your build scripts. 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.)

32.10.3 How does the Checker Framework compare with the JDK's `Optional` type?

JDK 8 introduced the `Optional` class, a container that is either empty or contains a non-null value.

`Optional` has numerous problems without countervailing benefits. `Optional` does not make your code more correct or robust. There is a real problem that `Optional` tries to solve, but you are better off using a regular possibly-null Java reference and the Nullness Checker (see Chapter 3, page 25), rather than using Optional.

The `Optional` class does *not* solve the problem of null pointer exceptions. Changing your code to use the `Optional` class has these effects:

- It transforms a `NullPointerException` into a `NoSuchElementException`, which still crashes your program.
- It creates new problems that were not a danger before.
- It clutters your code.
- It adds both space and time overheads.
- It does not address important sources of null pointer exceptions, such as partially-initialized objects and calls to `Map.get`.

See the article "Nothing is better than Java's `Optional` class" for more details and explanation of the benefits of `@Nullable` over `Optional`.

The `Optional` class provides utility routines to reduce clutter when using `Optional`. The Nullness Checker provides an `Opt` class that provides all the same methods, but written for regular possibly-null Java references.

32.10.4 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.

32.10.5 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.

32.10.6 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; as a result, JSR 305 was of limited utility as a specification. JSR 305 has been abandoned; there has been no activity by its expert group since 2009.

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.

In the future, the Java Community Process might revitalize JSR 305 or create a replacement JSR to 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 3.8.

32.10.7 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.

Chapter 33

Troubleshooting, getting help, and contributing

The manual might already answer your question, so first please look for your answer in the manual, including this chapter and the FAQ (Chapter 32). 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 https://groups.google.com/forum/#!forum/checker-framework-discuss. To report bugs, please see Section 33.2. 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://rawgit.com/typetools/checker-framework/master/docs/developer/gsoc-ideas.html.

33.1 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.

33.1.1 Unable to compile the Checker Framework

If you get the following error while compiling the Checker Framework itself:

```
checker-framework/stubparser/dist/stubparser.jar(org/checkerframework/stubparser/ast/CompilationUnit.class):
warning: [classfile] Signature attribute introduced in version 49.0 class files is ignored in version 46.0 class files
```

and you have used Eclipse to compile the Checker Framework, then probably you are using a very old version of Eclipse. (If you install Eclipse from the Ubuntu 16.04 repository, you get Eclipse version 3.8. Ubuntu 16.04 was released in April 2016, and Eclipse 3.8 was released in June 2012, with subsequent major releases in June 2013, June 2014, and June 2015.) Install the latest version of Eclipse and use it instead.

33.1.2 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 33.2.) 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 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 31.6. 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` or one of the alternatives described in Section 31.6.

- 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 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.

- A "class file for ... not found" error, especially for an inner class in the JDK, is probably due to a JDK version mismatch. To solve the problem, you need to perform compilation with a different Java version or different version of the JDK.

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

```
./gradlew  buildJdk assemble
```

to re-generate files `checker/jdk/jdk{8,9}.jar` and `checker/bin/jdk{8,9}.jar`.
That usually works, but if not, then you should recompile the Checker Framework from source (Section 33.3) 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 31.2) and/or setting the `build.compiler` property to `extJavac`.
  If you are building from source (Section 33.3), 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 like the following

  ```
  error: scoping construct for static nested type cannot be annotated
  error: scoping construct cannot be annotated with type-use annotation
  ```

  then you have probably written something like `@Nullable java.util.List`. The correct Java syntax to write an annotation on a fully-qualified type name is to put the annotation on the simple name part, as in `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`.

- A message of the form

  ```
  warning: StubParser: annotateTypeParameters: mismatched sizes
  ```

  might be because you compiled your project with `-source 1.4`. Java 1.4 doesn't support type parameters, but the Checker Framework expects that classes such as `List` have a type parameter. The discrepancy causes a problem. To fix the problem, supply `-source 1.5` or later, or no `-source` command-line argument.

33.1.3 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 you should formulate a proof about why the property is true. Your proof depends on some set of facts about the program and its operation. State them completely; for example, don't write "the field `f` cannot be null when execution reaches line 22", but justify *why* the field cannot be null.
  Once you have written down your proof, translate each fact into a Java annotation.
  If you are unable to express some aspect of your proof as an annotation, then the type system is not capable of reproducing your proof. You might need to find a different proof, or extend the type system to be more expressive, or suppress the warning.
  If you are able to express your entire proof as annotations, then look at the Checker Framework error messages. One possibility is that the errors indicate that your proof was incomplete or incorrect. Maybe your proof implicitly depended on some fact you didn't write down, such as "method `m` has no side effects." In this case, you should revise your proof and add more annotations to your Java code. Another possibility is that your proof is incorrect, and the errors indicate where. Another possibility is that there is a bug in the type-checker. In this case, you should report it to the maintainers, giving your proof and explaining why the type-checker should be able to make the same inferences that you did.
  Recall that the Checker Framework does modular verification, one procedure at a time; it observes the specifications, but not the implementations, of other methods.
  Also see Section 2.4.6, which explains this same methodology in different words.
- 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 a Java 8 or later 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 25.4.5 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 29). 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 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(literals=LiteralKind.NULL)` 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.

- 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`.
- 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.
- When one formal parameter's annotation references another formal parameter's name, as in this constructor:

  ```
  public String(char value[], @IndexFor("value") int offset, @IndexOrHigh("value") int count) { ... }
  ```

  you will get an error such as

  ```
  [expression.unparsable.type.invalid] Expression in dependent type annotation invalid:
  [error for expression: myParamName error: myParamName: identifier not found]
  ```

  Section 25.5 explains that you need to use a different syntax to refer to a formal parameter:

  ```
  public String(char value[], @IndexFor("#1") int offset, @IndexOrHigh("#1") int count) { ... }
  ```

33.1.4 Unexpected compilation output when running javac without a pluggable type-checker

On rare occasions, javac may issue an error when compiling against the Checker Framework's annotated JDK that you do not get when using the regular JDK. (Recall that the Checker Framework compiler uses the annotated JDK.) This may occur even when you are *not* running any annotation processor. The error is:

```
unchecked method invocation: method myMethod in class C is applied to given types
```

The reason for this is that there are some wildcards in the annotated JDK that have been given an explicit upper bound — they have been changed from, for example, `List<?>` to `List<? extends Object>`. The JDK treats these two types as identical in almost all respects. However, they have a different representation in bytecode. More importantly, `List<?>` is reifiable but `List<? extends Object>` is not; this means that the compiler permits uses of the former in some locations where it issues a warning for the latter. You can suppress the warning.

A message of the form

```
error: annotation values must be of the form 'name=value'
      @LTLengthOf("firstName", "lastName") int index;
                  ^
```

is caused by incorrect Java syntax. When you supply a set of multiple values as an annotation argument, you need to put curly braces around them:

```
      @LTLengthOf({"firstName", "lastName"}) int index;
```

A message of the form

```
Error: cannot find symbol
```

for an annotation type that is imported, and is a applied to a statically-imported enum, is caused by a javac bug, unrelated to the Checker Framework. See `https://bugs.openjdk.java.net/browse/JDK-7101822` and `https://bugs.openjdk.java.net/browse/JDK-8169095`.

33.2 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.) If the problem is with an incorrect or missing annotation on a library, including the JDK, see Section 33.2.1.

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 helping you. 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. Always include plaintext, not just a screenshot. Try to reproduce the problem from the command line as well as from an IDE or within a build system; that helps to indicate whether the problem is with the IDE or build system.
- 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, because some mailers mangle formatting of pasted text. If you encountered a problem while using tool integration such as Maven integration, then try to reproduce the problem from the command line as well — this will indicate whether the problem is with the checker itself or with the tool integration.
- 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).
- Indicate what you expected the result to be, since a bug is a difference between desired and actual outcomes. Also, please indicate **why** you expected that result — explaining your reasoning can help you understand how your reasoning is different than the checker's and which one is wrong. Remember that the checker reasons modularly and intraprocedurally: it examines one method at a time, using only the method signatures of other methods.
- 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 from 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.

When reporting bugs, please focus on realistic scenarios and well-written code. We are sure that you can make up artificial code that stymies the type-checker! Those aren't a good use of your time to report nor the maintainers' time to evaluate and fix.

33.2.1 Problems with annotated libraries

Sometimes, a checker will report a warning because a library is not annotated or because the library contains incorrect annotations. Please do not open GitHub issues for these. Instead, we appreciate pull requests to help us improve the annotations. Alternatively or in the interim, you can use stub files as described in Section 29.5.

If the library is annotated but has some incorrect or missing annotations, then please open a pull request that adds the annotations. You can find the annotated source for non-JDK libraries in `https://github.com/typetools/`. You can find the annotated JDK source in the Checker Framework GitHub repository, either in subdirectories of `checker/jdk` or in files named `jdk.astub` in the checker's implementation directory.

If the library is not annotated at all, you can fork it, add annotations, and propose that your annotated version be moved into `https://github.com/typetools/`.

For either approach, please see Chapter 29, page 176 for tips about writing library annotations.

Thanks for your contribution to the annotated libraries!

33.3 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.

The Checker Framework can be easily built on Unix (Linux and Mac). You might have trouble building on Windows. If so, you can switch to Unix (at least for building, if not for running, the Checker Framework) or you can report the problems to the Checker Framework developers.

33.3.1 Install prerequisites

You need to install several packages in order to build the Checker Framework. Follow the instructions for your operating system.

Ubuntu Run the following commands:

```
sudo apt-get update
sudo apt-get install --yes ant dia git hevea junit4 librsvg2-bin libcurl3-gnutls \
 make maven mercurial openjdk-8-jdk texlive-latex-base texlive-latex-recommended \
 texlive-latex-extra texlive-fonts-recommended unzip
```

Mac OS If you employ homebrew to install packages, run the following commands (you may need to update the version number `2.31`):

```
brew update
brew install git ant hevea maven mercurial librsvg unzip make
brew cask install java
brew cask install mactex
```

Make latex directory and copy hevea.sty file into it.

```
mkdir -p $HOME/Library/texmf/tex/latex
cp -p /usr/local/Cellar/hevea/2.31/lib/hevea/hevea.sty $HOME/Library/texmf/tex/latex/
```

Note: If copy permission denied, try sudo.
Also set the `JAVA_HOME` environment variable in your `.bashrc` file; for example,

```
export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk1.8.0_144.jdk/Contents/Home
```

Windows To build on Windows 10, run `bash` to obtain a version of Ubuntu (provided by the Windows Subsystem for Linux) and follow the Ubuntu instructions.

33.3.2 Obtain the source

Obtain the latest source code from the version control repository:

```
export JSR308=$HOME/jsr308
mkdir -p $JSR308
cd $JSR308
git clone https://github.com/typetools/checker-framework.git checker-framework
```

You might want to add the `export JSR308=$HOME/jsr308` line to your `.bashrc` file.

Set the `JAVA_HOME` environment variable to the location of your JDK 8 installation (not the JRE installation, and not JDK 7 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.)

Ubuntu If you use the bash shell, put the following command in your `.bashrc` file. (This might not work if `java` refers to an executable in the JRE rather than in the JDK, but you need the JDK installed to build the Checker Framework anyway.)

```
export JAVA_HOME=${JAVA_HOME:-$(dirname $(dirname $(dirname $(readlink -f $(/usr/bin/which java)))))}
```

Mac OSX 10.5 or later Add one of the following two commands in your `.bash_profile` or `.profile` file according to the version of your JDK.

```
export JAVA_HOME=$(/usr/libexec/java_home -v 1.8)
export JAVA_HOME=$(/usr/libexec/java_home -v 1.9)
```

33.3.3 Build the Checker Framework

1. Run `./gradlew cloneAndBuildDependencies` to build the Checker Framework dependencies:

   ```
   cd $JSR308/checker-framework
   ./gradlew cloneAndBuildDependencies
   ```

2. Run `./gradlew assemble` to build the Checker Framework:

   ```
   cd $JSR308/checker-framework
   ./gradlew assemble
   ```

3. Once it is built, you may wish to put the Checker Framework's `javac` even earlier in your `PATH`:

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

   The Checker Framework's `javac` ensures that all required libraries are on your classpath and boot classpath, but is otherwise identical to the Type Annotations compiler.
   Putting the Checker Framework's `javac` earlier in your `PATH` will ensure that the Checker Framework's version is used.
4. If you are developing a checker within the Checker Framework, there is a developer version of `javac` in the `bin-devel` directory. This version will use compiled classes from `dataflow/build`, `javacutil/build`, `framework/build`, and `checker/build` in the `checker-framework` directory instead of the compiled jar files, and by default will print stack traces for all errors.
   To use it, set your `PATH` to use `javac` in the `bin-devel` directory:

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

   The developer version of `javac` allows you to not have to rebuild the jar files after every code change, in turn allowing you to test your changes faster. Source files can be compiled using command `ant build` in the `checker` directory, or can be automatically compiled by an IDE such as Eclipse.
5. Test that everything works:
   - Run `./gradlew allTests`:

     ```
     cd $JSR308/checker-framework
     ./gradlew allTests
     ```

   - Run the Nullness Checker examples (see Section 3.5, page 31).

33.3.4 Build the Checker Framework Manual (this document)

1. Install needed packages; see Section 33.3.1 for instructions.
2. Run `make` in the `docs/manual` directory to build both the PDF and HTML versions of the manual.

33.3.5 Configure Eclipse to edit the Checker Framework

These instructions are relevant if you use Eclipse as your IDE.

1. Follow the instructions earlier in Section 33.3 to clone and build all projects from their sources.
2. Download Eclipse from the official Eclipse website and install it.
3. Run Eclipse.

4. Enter the main Eclipse working screen and in the “File” menu, select “Import” → “General” → “Existing Projects into Workspace”.
5. After the “Import Projects” window appears, select “Select Root Directory”, and select the directory pointed to by `$JSR308`.
6. Ensure “Search for nested projects” is selected in the Options panel.
7. Select all the projects in the folder except for “personalblog-demo”, then select “Finish” to import the projects.

Eclipse should successfully build all the imported projects. If Eclipse reports any errors, ensure you followed the instructions in Section 33.3.

33.3.6 Enable Travis continuous integration builds

Travis-CI is a continuous integration system: it runs tests every time you push a commit to GitHub.

If you have forked any of the projects rather than just creating a clone, then we recommend that you enable Travis-CI builds, so that you learn quickly of any errors that creep into your fork.

Run the Travis-CI getting started directions, though note that the `.travis.yml` files already exist, so only the first two steps may be necessary.

To save time, by default the Travis jobs download a pre-built version of the JDK rather than building the JDK from the sources in the Checker Framework repository. This means that if you make a change to the annotated JDK, your tests will fail. Here is how to deal with this issue:

1. In your pull request, edit file `.travis.yml` so that it rebuilds the JDK (some jobs might time out).
2. Verify that all tests pass.
3. A Checker Framework developer needs to update the pre-built JDK from your branch.
4. Undo your edits to file `.travis.yml`.
5. Verify that all tests pass.
6. A Checker Framework developer will review your code and merge your branch.

Debugging Travis continuous integration builds

If a Travis job is failing, you can reproduce the problem locally. The Travis jobs all run within Docker containers. Install Docker on your local computer, then perform commands like the following to get a shell within Docker:

```
docker pull mdernst/ubuntu-for-cf-jdk8
docker run -it mdernst/ubuntu-for-cf-jdk8 /bin/bash
```

Then, you can run arbitrary commands, including those that appear in the `docker run` command in the `.travis.yml` file. For example:

```
export JAVA_HOME=`which javac|xargs readlink -f|xargs dirname|xargs dirname`
git clone -b $BRANCHNAME --depth 9 https://github.com/$USER/checker-framework.git checker-framework
cd checker-framework
./.travis-build.sh all-tests downloadjdk
```

33.3.7 Code style

Code in this project follows the Google Java Style Guide except 4 spaces are used for indentation. If you commit changes to the Git repository, then a pre-commit hook verifies that the changes follow the style by running `ant check-format` in the top-level directory.

If the pre-commit hook fails because of improper formatting, run `ant reformat`, stage the changes, and try the commit operation again.

If the pre-commit hook or `ant reformat` fails with an "Argument list too long" error, then create a `local.properties` file in the top-level directory that sets the property `maxparallel` to 4000. If the error is still issued, reduce the number and try again.

33.4 Contributing fixes (creating a pull request)

Please see the document How to create and review a GitHub pull request for instructions about how to create a pull request, which is the way you can contribute code to the Checker Framework project. Thanks in advance for your participation!

For writing new test cases, see file `checker-framework/checker/tests/README`.

33.5 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 3.9.8) "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 6) See the two papers about the Checker Framework, described above.

Locking (Chapter 7) "Locking discipline inference and checking" (ICSE 2016, `http://homes.cs.washington.edu/~mernst/pubs/locking-inference-checking-icse2016-abstract.html`) describes the Lock Checker.

Fake enumerations, type aliases, and typedefs (Chapter 9) See the ICSE 2011 paper about the Checker Framework, described above.

Regular expressions (Chapter 11) "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 12) "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 15) See the ICSE 2011 paper about the Checker Framework, described above.

GUI Effects (Chapter 16) "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).

Thread locality (Section 23.3) "Loci: Simple thread-locality for Java" [WPM+09] (ECOOP 2009, `http://janvitek.org/pubs/ecoop09.pdf`)

Security (Section 23.8) "Static analysis of implicit control flow: Resolving Java reflection and Android intents" [BJM+15] (ASE 2015, `http://homes.cs.washington.edu/~mernst/pubs/implicit-control-flow-ase2015-abstract.html`) describes the SPARTA toolset and information flow type-checker.

"Boolean formulas for the static identification of injection attacks in Java" [ELM+15] (LPAR 2015, `http://homes.cs.washington.edu/~mernst/pubs/detect-injections-lpar2015-abstract.html`)

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-abstract.html`).

Approximate data (Section 23.6) "EnerJ: Approximate Data Types for Safe and General Low-Power Computation" [SDF+11] (PLDI 2011, `https://homes.cs.washington.edu/~luisceze/publications/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.

IGJ and OIGJ immutability (Section 23.9) "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 (Section 23.9) "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.

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.)

33.6 Comparison to other tools

A pluggable type-checker, such as those created by the Checker Framework, is a verification tool that prevents or detects 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 or verifier 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 Chapter 26.
- A type-checker requires you to annotate your code with type qualifiers, or to run an inference tool that does so for you. Some bug detectors do 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.

Another alternative is to use an IDE tool such as those built into Eclipse or IntelliJ. They are bug-finding tools that give up on precision, soundness, or both. For a more detailed comparison to Eclipse, see Section 32.10.2.

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

33.7 Credits and changelog

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 `https://checkerframework.org/changelog.txt`.

Developers who have contributed code to the Checker Framework include Abraham Lin, Anatoly Kupriyanov, Asumu Takikawa, Bohdan Sharipov, Charlie Garrett, Chris Mackie, Colin Gordon, Dan Brotherston, Dan Brown, David Lazar, David McArthur, Eric Spishak, Google Inc. (via @wmdietlGC), Javier Thaine, Jeff Luo, Jiasen (Jason) Xu, Jonathan Burke, Kivanc Muslu, Konstantin Weitz, Mahmood Ali, Mark Roberts, Matt Mullen, Michael Bayne, Michael Coblenz, Michael Ernst, Michael Sloan, Paul Vines, Paulo Barros, Philip Lai, Renato Athaydes, René Just, Ryan Oblak, Shinya Yoshida, Stefan Heule, Steph Dietzel, Stuart Pernsteiner, Suzanne Millstein, Trask Stalnaker, Vlastimil Dort, Werner Dietl. In addition, too many users to list have provided valuable feedback, which has improved the toolset's design and implementation. Thanks for your help!

33.8 License

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, with the classpath exception. The GPL is the same license that OpenJDK is licensed under. Just as compiling your code with javac does not infect your code with the GPL, type-checking your code with the Checker Framework does not infect your code with the GPL. Running the Checker Framework during development has no effect on your intellectual property or licensing. If you want to ship the Checker Framework as part of your product, then your product must be licensed under the GPL.
- The more permissive MIT License applies to code that you might want to include in your own program, such as the annotations and run-time utility classes.

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.

[BJM+15] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d'Amorim, and Michael D. Ernst. Static analysis of implicit control flow: Resolving Java reflection and Android intents. In *ASE 2015: Proceedings of the 30th Annual International Conference on Automated Software Engineering*, pages 669–679, Lincoln, NE, USA, November 2015.

[CNA+17] Michael Coblenz, Whitney Nelson, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. Glacier: Transitive class immutability for Java. In *ICSE 2017, Proceedings of the 39th International Conference on Software Engineering*, pages 496–506, Buenos Aires, Argentina, May 2017.

[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 2011, Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690, Waikiki, Hawaii, USA, May 2011.

[DEM11] Werner Dietl, Michael D. Ernst, and Peter Müller. Tunable static inference for Generic Universe Types. In *ECOOP 2011 — Object-Oriented Programming, 25th European Conference*, pages 333–357, Lancaster, UK, July 2011.

[EJM+14] Michael D. Ernst, René Just, Suzanne Millstein, Werner 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 *CCS 2014: Proceedings of the 21st ACM Conference on Computer and Communications Security*, pages 1092–1104, Scottsdale, AZ, USA, November 2014.

[ELM+15] Michael D. Ernst, Alberto Lovato, Damiano Macedonio, Ciprian Spiridon, and Fausto Spoto. Boolean formulas for the static identification of injection attacks in Java. In *LPAR 2015: Proceedings of the 20th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 130–145, Suva, Fiji, November 2015.

[Ern08] Michael D. Ernst. Type Annotations specification (JSR 308). `https://checkerframework.org/jsr308/`, September 12, 2008.

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

[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 2013.

[Goe06] Brian Goetz. The pseudo-typedef antipattern: Extension is not type definition. `https://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 2012.

[HMDE12] Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. ReIm & ReImInfer: Checking and inference of reference immutability and method purity. In *OOPSLA 2012, Object-Oriented Programming Systems, Languages, and Applications*, pages 879–896, Tucson, AZ, USA, October 2012.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. In *OOPSLA Companion: Companion to Object-Oriented Programming Systems, Languages, and Applications*, pages 132–136, Vancouver, BC, Canada, October 2004.

[HSP05] David Hovemeyer, Jaime Spacco, and William Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *PASTE 2005: ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2005)*, pages 13–19, Lisbon, Portugal, September 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 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 2008.

[SDE12] Eric Spishak, Werner Dietl, and Michael D. Ernst. A type system for regular expressions. In *FTfJP: 14th Workshop on Formal Techniques for Java-like Programs*, pages 20–26, Beijing, China, June 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 2011.

[SM11] Alexander J. Summers and Peter Müller. Freedom before commitment: A lightweight type system for object initialisation. In *OOPSLA 2011, Object-Oriented Programming Systems, Languages, and Applications*, pages 1013–1032, Portland, OR, USA, October 2011.

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

[TPV10] Daniel Tang, Ales Plsek, and Jan Vitek. Static checking of safety critical Java annotations. In *JTRES 2010: 8th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 148–154, Prague, Czech Republic, August 2010.

[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 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 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 2007.

[ZPL+10] Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. Ownership and immutability in generic Java. In *OOPSLA 2010, Object-Oriented Programming Systems, Languages, and Applications*, pages 598–617, Revo, NV, USA, October 2010.