

The Checker Framework Manual: Custom pluggable types for Java

https://checkerframework.org

Version 4.4.0 (2026-11-02)

For the impatient: Section 1.3 (page 16) describes how to **install and use** pluggable type-checkers.

Contents

1 Introduction **15**
- 1.1 How to read this manual . . . 16
- 1.2 How it works: Pluggable types . . . 16
- 1.3 Installation . . . 16
- 1.4 Example use: detecting a null pointer bug . . . 17

2 Using a checker **18**
- 2.1 Where to write type annotations . . . 18
- 2.2 Running a checker . . . 19
  - 2.2.1 Using annotated libraries . . . 19
  - 2.2.2 Summary of command-line options . . . 20
  - 2.2.3 Checker auto-discovery . . . 23
  - 2.2.4 Shorthand for built-in checkers . . . 23
- 2.3 What the checker guarantees . . . 24
- 2.4 Tips about writing annotations . . . 24
  - 2.4.1 Write annotations before you run a checker . . . 25
  - 2.4.2 How to get started annotating legacy code . . . 25
  - 2.4.3 Annotations indicate non-exceptional behavior . . . 26
  - 2.4.4 Subclasses must respect superclass annotations . . . 28
  - 2.4.5 What to do if a checker issues a warning about your code . . . 29
  - 2.4.6 Calls to unannotated code (legacy libraries) . . . 30

3 Nullness Checker **31**
- 3.1 What the Nullness Checker guarantees . . . 31
  - 3.1.1 Nullness Checker optional warnings . . . 32
- 3.2 Nullness annotations . . . 32
  - 3.2.1 Nullness qualifiers . . . 32
  - 3.2.2 Nullness method annotations . . . 33
  - 3.2.3 Initialization qualifiers . . . 34
  - 3.2.4 Map key qualifiers . . . 34
- 3.3 Writing nullness annotations . . . 34
  - 3.3.1 Implicit qualifiers . . . 34
  - 3.3.2 Default annotation . . . 34
  - 3.3.3 Conditional nullness . . . 35
  - 3.3.4 Nullness and array initialization . . . 35
  - 3.3.5 Nullness and conversions from collections to arrays . . . 36
  - 3.3.6 Run-time checks for nullness . . . 36
  - 3.3.7 Inference of `@NonNull` and `@Nullable` annotations . . . 36
- 3.4 Suppressing nullness warnings . . . 37
  - 3.4.1 Suppressing warnings with assertions and method calls . . . 37

3.4.2 Null arguments to collection classes . . . 38
3.4.3 Conservative nullness annotations on the JDK . . . 38
3.5 Examples . . . 38
3.5.1 Tiny examples . . . 38
3.5.2 Example annotated source code . . . 39
3.5.3 Publications . . . 39
3.6 Tips for getting started . . . 39
3.7 Other tools for nullness checking . . . 40
3.7.1 Which tool is right for you? . . . 40
3.7.2 Incompatibility note about FindBugs and SpotBugs `@Nullable` . . . 40
3.7.3 Relationship to `Optional<T>` . . . 43
3.8 Initialization Checker . . . 44
3.8.1 Initialization qualifiers . . . 45
3.8.2 How an object becomes initialized . . . 45
3.8.3 `@UnderInitialization` examples . . . 46
3.8.4 Partial initialization . . . 47
3.8.5 Method calls from the constructor . . . 49
3.8.6 Initialization of circular data structures . . . 50
3.8.7 How to handle warnings . . . 52
3.8.8 Suppressing warnings . . . 53
3.8.9 More details about initialization checking . . . 53

4 Map Key Checker **54**
4.1 Invoking the Map Key Checker . . . 54
4.2 Map key annotations . . . 54
4.3 Default annotations . . . 55
4.3.1 Default for lower bounds . . . 55
4.3.2 Diagnosing the need for explicit @KeyFor on lower bounds . . . 56
4.4 Examples . . . 56
4.5 Local inference of @KeyFor annotations . . . 57

5 Optional Checker for possibly-present data **59**
5.1 How to run the Optional Checker . . . 59
5.2 Optional annotations . . . 59
5.2.1 Optional method annotations . . . 60
5.3 What the Optional Checker guarantees . . . 60
5.4 Suppressing optional warnings . . . 61
5.4.1 Suppressing warnings with assertions and method calls . . . 61

6 Interning Checker **62**
6.1 Interning annotations . . . 63
6.1.1 Interning qualifiers . . . 63
6.1.2 Interning method and class annotations . . . 63
6.2 Annotating your code with `@Interned` . . . 63
6.3 Interned classes . . . 64
6.3.1 The intern() methods . . . 64
6.3.2 Default qualifiers and qualifiers for literals . . . 65
6.3.3 InternedDistinct: values not equals() to any other value . . . 65
6.4 What the Interning Checker checks . . . 65
6.4.1 Imprecision (false positive warnings) of the Interning Checker . . . 66
6.5 Examples . . . 66
6.6 Other interning annotations . . . 67

7 Called Methods Checker for the builder pattern and more **68**
7.1 How to run the Called Methods Checker . . . 68
7.2 For Lombok users . . . 69
7.3 Specifying your code . . . 69
7.4 Default handling for Lombok and AutoValue . . . 72
7.5 Using the Called Methods Checker for properties unrelated to builders . . . 72
7.6 More information . . . 72

8 Resource Leak Checker for must-call obligations **73**
8.1 How to run the Resource Leak Checker . . . 73
8.2 Resource Leak Checker annotations . . . 74
8.3 Example of how safe resource usage is verified . . . 74
8.4 Aliased references and ownership transfer . . . 75
8.4.1 Owning parameters and exceptions . . . 76
8.4.2 Owning fields . . . 76
8.5 Resource aliasing . . . 77
8.5.1 A complete wrapper type example . . . 78
8.6 Creating obligations (how to re-assign a non-final owning field) . . . 78
8.6.1 Requirements at a call site of a `@CreatesMustCallFor` method . . . 79
8.6.2 Requirements at a declaration of a `@CreatesMustCallFor` method . . . 79
8.7 Ignored exception types . . . 80
8.8 Errors about field initialization . . . 80
8.9 Errors about unknown must-call obligations . . . 81
8.10 Collections of resources . . . 81
8.11 Further reading . . . 81

9 Fake Enum Checker for fake enumerations **83**
9.1 Fake enum annotations . . . 83
9.2 What the Fenum Checker checks . . . 84
9.3 Running the Fenum Checker . . . 84
9.4 Suppressing warnings . . . 85
9.5 Example . . . 85
9.6 The fake enumeration pattern . . . 86
9.7 References . . . 86

10 Lock Checker **87**
10.1 What the Lock Checker guarantees . . . 87
10.2 Lock annotations . . . 87
10.2.1 Type qualifiers . . . 87
10.2.2 Declaration annotations . . . 89
10.3 Type-checking rules . . . 89
10.3.1 Polymorphic qualifiers . . . 89
10.3.2 Dereferences . . . 90
10.3.3 Primitive types, boxed primitive types, and Strings . . . 90
10.3.4 Overriding . . . 90
10.3.5 Side effects . . . 90
10.4 Examples . . . 91
10.4.1 Examples of @GuardedBy . . . 91
10.4.2 @GuardedBy({“a”, “b”}) is not a subtype of @GuardedBy({“a”}) . . . 92
10.4.3 Examples of @Holding . . . 92
10.4.4 Examples of @EnsuresLockHeld and @EnsuresLockHeldIf . . . 93
10.4.5 Example of @LockingFree, @ReleasesNoLocks, and @MayReleaseLocks . . . 93

10.4.6 Polymorphism and method formal parameters with unknown guards . . . 94
10.5 More locking details . . . 95
10.5.1 Two types of locking: monitor locks and explicit locks . . . 95
10.5.2 Held locks and held expressions; aliasing . . . 95
10.5.3 Run-time checks for locking . . . 96
10.5.4 Discussion of default qualifier . . . 96
10.5.5 Discussion of `@Holding` . . . 96
10.6 Other lock annotations . . . 97
10.6.1 Relationship to annotations in *Java Concurrency in Practice* . . . 97
10.7 Possible extensions . . . 97

11 Index Checker for sequence bounds (arrays and strings) 98
11.1 Index Checker structure and annotations . . . 98
11.2 Lower bounds . . . 99
11.3 Upper bounds . . . 100
11.4 Sequence minimum lengths . . . 102
11.5 Sequences of the same length . . . 103
11.6 Binary search indices . . . 103
11.7 Substring indices . . . 104
11.7.1 The need for the `@SubstringIndexFor` annotation . . . 105
11.8 Inequalities . . . 106
11.9 Annotating your own fixed-size datatypes . . . 106
11.10Technical papers . . . 107

12 Tainting Checker 108
12.1 Tainting annotations . . . 108
12.2 Tips on writing `@Untainted` annotations . . . 108
12.3 `@Tainted` and `@Untainted` can be used for many purposes . . . 109
12.4 A caution about polymorphism and side effects . . . 109

13 SQL Quotes Checker 111
13.1 SQL Quotes annotations . . . 111
13.2 What the SQL Quotes Checker checks . . . 111
13.3 Library annotations . . . 112

14 Regex Checker for regular expression syntax 113
14.1 Regex annotations . . . 113
14.2 Annotating your code with `@Regex` . . . 113
14.2.1 Implicit qualifiers . . . 113
14.2.2 Capturing groups . . . 114
14.2.3 Concatenation of partial regular expressions . . . 114
14.2.4 Testing whether a string is a regular expression . . . 115
14.2.5 Suppressing warnings . . . 115

15 Format String Checker 116
15.1 Formatting terminology . . . 116
15.2 Format String Checker annotations . . . 116
15.2.1 Conversion categories . . . 117
15.2.2 Subtyping rules for `@Format` . . . 118
15.3 What the Format String Checker checks . . . 119
15.3.1 Possible false alarms . . . 120
15.3.2 Possible missed alarms . . . 120

15.4 Implicit qualifiers . . . 121
15.5 @FormatMethod . . . 121
15.6 Testing whether a format string is valid . . . 121

16 Internationalization Format String Checker (I18n Format String Checker) 123
16.1 Internationalization Format String Checker annotations . . . 123
16.2 Conversion categories . . . 124
16.3 Subtyping rules for `@I18nFormat` . . . 124
16.4 What the Internationalization Format String Checker checks . . . 125
16.5 Resource files . . . 126
16.6 Running the Internationalization Format Checker . . . 127
16.7 Testing whether a string has an i18n format type . . . 127
16.8 Examples of using the Internationalization Format Checker . . . 127

17 Property File Checker 129
17.1 General Property File Checker . . . 129
17.2 Internationalization Checker (I18n Checker) . . . 130
17.2.1 Internationalization annotations . . . 130
17.2.2 Running the Internationalization Checker . . . 130
17.3 Compiler Message Key Checker . . . 130

18 Signature String Checker for string representations of types 132
18.1 Signature annotations . . . 132
18.1.1 How to choose which annotation to use . . . 134
18.2 What the Signature Checker checks . . . 134

19 GUI Effect Checker 135
19.1 GUI effect annotations . . . 136
19.2 What the GUI Effect Checker checks . . . 136
19.3 Running the GUI Effect Checker . . . 136
19.4 Annotation defaults . . . 136
19.5 Polymorphic effects . . . 137
19.5.1 Defining an effect-polymorphic type . . . 137
19.5.2 Using an effect-polymorphic type . . . 137
19.5.3 Subclassing a specific instantiation of an effect-polymorphic type . . . 138
19.5.4 Subtyping with polymorphic effects . . . 138
19.6 References . . . 139

20 Units Checker 140
20.1 Units annotations . . . 140
20.2 Extending the Units Checker . . . 141
20.3 What the Units Checker checks . . . 142
20.4 Running the Units Checker . . . 142
20.5 Suppressing warnings . . . 143
20.6 References . . . 143

21 Signedness Checker 144
21.1 Annotations . . . 144
21.1.1 Default qualifiers . . . 145
21.2 Prohibited operations . . . 145
21.2.1 Rationale . . . 146
21.2.2 Permitted shifts . . . 146

21.3 Utility routines for manipulating unsigned values . . . 146
21.4 Local type refinement . . . 147
21.5 Instantiating polymorphism . . . 147
21.6 Other signedness annotations . . . 147

22 Modifiability Checker **148**
22.1 Unmodifiable collections . . . 148
22.2 Modifiability annotations . . . 149
22.2.1 Alias annotations . . . 150
22.2.2 Qualifiers . . . 150
22.2.3 Polymorphic qualifiers . . . 152
22.2.4 Modifiability method annotations . . . 152
22.2.5 Inherited implementations . . . 153
22.3 `@Modifiable` and `@Unmodifiable` for types without grow, seq-grow, shrink, and/or replace methods . 153
22.3.1 Alternative design: use negative capability rather than top qualifier . . . 153
22.3.2 Expansion rules for `@Modifiable` and `@Unmodifiable` . . . 154
22.3.3 Example expansions for `@Modifiable` and `@Unmodifiable` . . . 154
22.3.4 Expansion rules for `@PolyModifiable` . . . 154
22.4 Behavior of the `iterator()` method . . . 154
22.5 Examples of Modifiability Checker warnings . . . 155

23 Purity Checker **157**
23.1 Purity annotations . . . 157
23.2 Lambdas, method references, and functional interfaces . . . 157
23.2.1 Functional-interface parameters of methods with purity annotations . . . 158
23.3 Purity annotations are trusted . . . 159
23.4 Overriding methods inherit specifications from overridden methods . . . 160
23.5 Suppressing warnings . . . 160

24 Constant Value Checker **161**
24.1 Annotations . . . 161
24.1.1 Type annotations . . . 161
24.2 Other constant value annotations . . . 162
24.2.1 Compile-time execution of expressions . . . 163
24.2.2 `@StaticallyExecutable` methods and the classpath . . . 163
24.3 Warnings . . . 164
24.4 Unsoundly ignoring overflow . . . 164
24.5 Strings can be null in concatenations . . . 165

25 Returns Receiver Checker **166**
25.1 Annotations . . . 166
25.2 AutoValue and Lombok support . . . 166

26 Reflection resolution **168**
26.1 Reflection resolution example . . . 168
26.2 MethodVal and ClassVal Checkers . . . 169
26.2.1 ClassVal Checker . . . 169
26.2.2 MethodVal Checker . . . 170
26.2.3 MethodVal and ClassVal inference . . . 171

27 Initialized Fields Checker **173**
27.1 Running the Initialized Fields Checker . . . 173
27.2 Motivation: uninitialized fields . . . 173
27.3 Example . . . 174
27.4 Annotations . . . 174
27.5 Comparison to the Initialization Checker . . . 175

28 Aliasing Checker **176**
28.1 Aliasing annotations . . . 176
28.2 Leaking contexts . . . 177
28.3 Restrictions on where `@Unique` may be written . . . 178
28.4 Aliasing type refinement . . . 178

29 Must Call Checker **180**
29.1 Must Call annotations . . . 180
29.2 Writing `@MustCall`/`@InheritableMustCall` on a class . . . 181
29.3 Relationship to lightweight ownership . . . 181
29.4 Assumptions about reflection . . . 182
29.5 Type parameter bounds often need to be annotated . . . 182

30 Subtyping Checker **183**
30.1 Using the Subtyping Checker . . . 183
30.1.1 Compiling your qualifiers and your project . . . 184
30.1.2 Suppressing warnings from the Subtyping Checker . . . 184
30.2 Subtyping Checker example . . . 184
30.3 Type aliases and typedefs . . . 186

31 Third-party checkers **188**
31.1 Determinism checker . . . 188
31.2 Constant Value Inference (Interval Inference) . . . 188
31.3 Crypto Checker . . . 188
31.4 AWS crypto policy compliance checker . . . 189
31.5 AWS KMS compliance checker . . . 189
31.6 PUnits units of measurement . . . 189
31.7 JaTyC typestate checker . . . 189
31.8 NullAway . . . 189
31.9 Nullness Rawness Checker . . . 189
31.10 UI Thread Checker for ReactiveX . . . 190
31.11 Practical Immutability For Classes And Objects (PICO) . . . 190
31.12 Read Checker and Cast Checker for ensuring that EOF is recognized . . . 190
31.13 Ontology type system . . . 190
31.14 Glacier: Class immutability . . . 190
31.15 SQL checker that supports multiple dialects . . . 190
31.16 Immutability checkers: IGJ, OIGJ, and Javari . . . 191
31.17 JCrypt: computation over encrypted data . . . 191
31.18 DroidInfer: information flow . . . 191
31.19 Error Prone linter . . . 191
31.20 SPARTA information flow type-checker for Android . . . 191
31.21 SFlow type system for information flow . . . 191
31.22 CheckLT taint checker . . . 192
31.23 EnerJ checker . . . 192
31.24 ReIm immutability . . . 192

31.25SFlow x ReIm for information flow and reference immutability . . . 192
31.26Generic Universe Types checker . . . 192
31.27Safety-Critical Java checker . . . 192
31.28Thread locality checker . . . 192
31.29Units and dimensions checker . . . 192
31.30Typestate checkers . . . 193
31.30.1 Comparison to flow-sensitive type refinement . . . 193

32 Generics and polymorphism **194**
32.1 Generics (parametric polymorphism or type polymorphism) . . . 194
32.1.1 Raw types . . . 194
32.1.2 Restricting instantiation of a generic class . . . 194
32.1.3 Overriding a generic method . . . 196
32.1.4 Type annotations on a use of a generic type variable . . . 197
32.1.5 Annotations on wildcards . . . 197
32.1.6 Examples of qualifiers on a type parameter . . . 198
32.1.7 Covariant type parameters . . . 199
32.1.8 Method type argument inference and type qualifiers . . . 199
32.1.9 The Bottom type . . . 199
32.2 Qualifier polymorphism for methods . . . 199
32.2.1 Using polymorphic qualifiers in a method signature . . . 200
32.2.2 Relationship to subtyping and generics . . . 200
32.2.3 Using multiple polymorphic qualifiers in a method signature . . . 201
32.2.4 Using a single polymorphic qualifier in a method signature . . . 201
32.3 Class qualifier parameters . . . 202
32.3.1 Resolving polymorphism when the receiver type has a polymorphic qualifier . . . 203
32.3.2 Using class qualifier parameters in the type of a field . . . 203
32.3.3 Local variable defaults for types with qualifier parameters . . . 203
32.3.4 Qualifier parameters by default . . . 204
32.3.5 Types with qualifier parameters as type arguments . . . 204

33 Advanced type system features **205**
33.1 Invariant array types . . . 205
33.2 Context-sensitive type inference for array constructors . . . 205
33.3 Upper bound of qualifiers on uses of a given type (annotations on a class declaration) . . . 206
33.4 The effective qualifier on a type (defaults and inference) . . . 207
33.5 Default qualifier for unannotated types . . . 207
33.5.1 Default for use of a type . . . 208
33.5.2 Controlling defaults in source code . . . 208
33.5.3 Defaulting rules and CLIMB-to-top . . . 209
33.5.4 Inherited defaults . . . 210
33.5.5 Inherited wildcard annotations . . . 211
33.5.6 Default qualifiers for `.class` files (library defaults) . . . 211
33.6 Annotations on constructors . . . 212
33.6.1 Annotations on constructor declarations . . . 212
33.6.2 Annotations on constructor invocations . . . 213
33.7 Type refinement (flow-sensitive type qualifier inference) . . . 213
33.7.1 Type refinement examples . . . 213
33.7.2 Type refinement behavior . . . 214
33.7.3 Which types are refined . . . 214
33.7.4 Run-time tests and type refinement . . . 215
33.7.5 Side effects, determinism, purity, and type refinement . . . 216

33.7.6 Assertions . . . 217
33.7.7 The `var` keyword . . . 218
33.8 Writing Java expressions as annotation arguments . . . 218
33.8.1 Limitations . . . 219
33.9 Field invariants . . . 220
33.10 Unused fields . . . 220
33.10.1 `@Unused` annotation . . . 221

34 Suppressing warnings **222**
34.1 `@SuppressWarnings` annotation . . . 223
34.1.1 `@SuppressWarnings` syntax . . . 223
34.1.2 Where `@SuppressWarnings` can be written . . . 223
34.1.3 Good practices when suppressing warnings . . . 224
34.2 `@AssumeAssertion` string in an `assert` message . . . 225
34.2.1 Suppressing warnings and defensive programming . . . 226
34.3 `-AsuppressWarnings` command-line option . . . 227
34.4 `-AskipUses` and `-AonlyUses` command-line options . . . 227
34.5 `-AskipDefs` and `-AonlyDefs` command-line options . . . 227
34.6 `-AskipFiles` and `-AonlyFiles` command-line options . . . 228
34.7 `-Alint` command-line option . . . 228
34.8 Change the specification of a method . . . 229
34.9 Don't run the processor . . . 229
34.10 Checker-specific mechanisms . . . 229

35 Type inference **230**
35.1 Type inference tools . . . 230
35.2 Whole-program inference . . . 231
35.3 Running whole-program inference on a single project, with buildfile editing . . . 231
35.3.1 Editing your buildfile . . . 231
35.3.2 Running `wpi2.sh` . . . 232
35.4 Running whole-program inference on a single project, without buildfile editing . . . 233
35.4.1 Whole-program inference results . . . 234
35.4.2 Requirements for whole-program inference scripts . . . 234
35.5 Running whole-program inference on many projects . . . 234
35.6 Whole-program inference that inserts annotations into source code . . . 236
35.7 Inference results depend on uses in your program or test suite . . . 236
35.7.1 Whole-program inference ignores some code . . . 237
35.7.2 Manually checking whole-program inference results . . . 237
35.8 How whole-program inference works . . . 237
35.9 Type inference compared to other whole-program analyses . . . 238

36 Annotating libraries **239**
36.1 Tips for annotating a library . . . 240
36.1.1 Don't change the code . . . 240
36.1.2 Library annotations should reflect the specification, not the implementation . . . 240
36.1.3 Report bugs upstream . . . 240
36.1.4 Fully annotate the library, or indicate which parts you did not . . . 240
36.1.5 Verify your annotations . . . 241
36.2 Creating an annotated library . . . 241
36.3 Creating an annotated JDK . . . 242
36.4 Compiling partially-annotated libraries . . . 242

36.4.1 The `-AuseConservativeDefaultsForUncheckedCode=source,bytecode` command-line argument . . . 242
36.5 Using stub classes . . . 243
36.5.1 Using a stub file . . . 243
36.5.2 Multiple specifications for a method . . . 243
36.5.3 Stub methods in subclasses of the declaring class . . . 244
36.5.4 Stub file format . . . 245
36.5.5 Creating a stub file . . . 245
36.5.6 Distributing stub files . . . 246
36.5.7 Troubleshooting stub libraries . . . 247
36.6 Ajava files . . . 248
36.6.1 Using an Ajava file . . . 248
36.6.2 Corresponding source files and ajava files . . . 248
36.7 Troubleshooting/debugging annotated libraries . . . 249

37 How to create a new checker **250**
37.1 How checkers build on the Checker Framework . . . 251
37.2 The parts of a checker . . . 251
37.3 Compiling and using a custom checker . . . 252
37.4 Tips for creating a checker . . . 252
37.5 Annotations: Type qualifiers and hierarchy . . . 254
37.5.1 Defining the type qualifiers . . . 254
37.5.2 Declaratively defining the qualifier hierarchy . . . 256
37.5.3 Procedurally defining the qualifier hierarchy . . . 257
37.5.4 Defining the default annotation . . . 257
37.5.5 Relevant Java types . . . 258
37.5.6 Do not re-use type qualifiers . . . 258
37.5.7 Completeness of the qualifier hierarchy . . . 258
37.5.8 Annotations whose argument is a Java expression (dependent type annotations) . . . 259
37.5.9 Repeatable annotations . . . 260
37.6 The checker class: Compiler interface . . . 260
37.6.1 Indicating supported annotations . . . 260
37.6.2 Bundling multiple checkers . . . 261
37.6.3 Providing command-line options . . . 261
37.7 Visitor: Type rules . . . 262
37.7.1 AST traversal . . . 263
37.7.2 Avoid hardcoding . . . 263
37.8 Type factory: Type introduction rules . . . 264
37.8.1 Procedurally specifying type introduction rules . . . 264
37.8.2 Recursion in `TreeAnnotators` . . . 265
37.9 Dataflow: enhancing flow-sensitive type refinement . . . 265
37.9.1 Determine which expressions to refine . . . 265
37.9.2 Create required class . . . 266
37.9.3 Override methods that handle Nodes of interest . . . 266
37.9.4 Implement the refinement . . . 267
37.9.5 Further customizing flow-sensitive inference . . . 268
37.10 Annotated JDK and other annotated libraries . . . 269
37.11 Testing framework . . . 269
37.12 Debugging options . . . 270
37.12.1 Amount of detail in messages . . . 270
37.12.2 Format of output . . . 270
37.12.3 Stub and JDK libraries . . . 270

37.12.4 Progress tracing . . . 270
37.12.5 Saving the command-line arguments to a file . . . 271
37.12.6 Visualizing the dataflow graph . . . 271
37.12.7 Miscellaneous debugging options . . . 272
37.12.8 Examples . . . 272
37.12.9 Using an external debugger . . . 273
37.13 Documenting the checker . . . 273
37.14 javac implementation survival guide . . . 274
37.14.1 Checker access to compiler information . . . 274
37.14.2 How a checker fits in the compiler as an annotation processor . . . 275
37.15 Integrating a checker with the Checker Framework . . . 276

38 Building an accumulation checker **277**
38.1 Publications . . . 278

39 Integration with external tools **279**
39.1 Android . . . 279
39.2 Android Studio and the Android Gradle Plugin . . . 280
39.3 Ant task . . . 280
39.3.1 Explanation . . . 281
39.4 Bazel . . . 281
39.5 Buck . . . 281
39.5.1 Troubleshooting . . . 282
39.6 Command line, via Checker Framework javac wrapper . . . 282
39.7 Command line, via JDK javac . . . 283
39.7.1 Non-modularized code . . . 283
39.7.2 Modularized code . . . 284
39.8 Eclipse . . . 285
39.8.1 Using an Ant task . . . 285
39.8.2 Troubleshooting Eclipse . . . 285
39.9 Gradle . . . 285
39.10 IntelliJ IDEA . . . 285
39.10.1 Running a checker on every IntelliJ compilation . . . 286
39.10.2 Running a checker on every IntelliJ change or save . . . 286
39.11 javac diagnostics wrapper . . . 286
39.12 Lombok . . . 286
39.12.1 Annotations on generated code . . . 287
39.12.2 Type-checking code with Lombok annotations . . . 287
39.12.3 Maven projects using Lombok . . . 287
39.13 Maven . . . 287
39.13.1 Maven, with a locally-built version of the Checker Framework . . . 289
39.13.2 Maven troubleshooting . . . 289
39.14 NetBeans . . . 289
39.14.1 Adding a checker via the Project Properties window . . . 290
39.14.2 Adding a checker via an Ant target . . . 290
39.15 sbt . . . 291
39.15.1 For modularized code . . . 291
39.16 tIDE . . . 292
39.17 Type inference tools . . . 292

40 Frequently Asked Questions (FAQs) 293
40.1 Motivation for pluggable type-checking . . . 295
40.1.1 I don't make type errors, so would pluggable type-checking help me? . . . 295
40.1.2 Should I use pluggable types (type qualifiers), or should I use Java subtypes? . . . 295
40.2 Getting started . . . 296
40.2.1 How do I get started annotating an existing program? . . . 296
40.2.2 Which checker should I start with? . . . 296
40.2.3 How can I join the checker-framework-dev mailing list? . . . 297
40.3 Usability of pluggable type-checking . . . 297
40.3.1 Are type annotations easy to read and write? . . . 297
40.3.2 Will my code become cluttered with type annotations? . . . 297
40.3.3 Will using the Checker Framework slow down my program? Will it slow down the compiler? 297
40.3.4 How do I shorten the command line when invoking a checker? . . . 298
40.3.5 Method pre-condition contracts, including formal parameter annotations, make no sense for public methods . . . 298
40.4 How to handle warnings and errors . . . 298
40.4.1 What should I do if a checker issues a warning about my code? . . . 298
40.4.2 Why does a checker issue a warning even though the code checks a property? . . . 298
40.4.3 What does a certain Checker Framework warning message mean? . . . 298
40.4.4 What do square brackets mean in a Checker Framework warning message? . . . 299
40.4.5 Can a pluggable type-checker guarantee that my code is correct? . . . 299
40.4.6 What guarantee does the Checker Framework give for concurrent code? . . . 299
40.4.7 How do I make compilation succeed even if a checker issues errors? . . . 300
40.4.8 Why does the checker always say there are 100 errors or warnings? . . . 300
40.4.9 Why does the Checker Framework report an error regarding a type I have not written in my program? . . . 300
40.4.10 Why does the Checker Framework accept code on one line but reject it on the next? . . . 300
40.4.11 How can I do run-time monitoring of properties that were not statically checked? . . . 300
40.5 False positive warnings . . . 300
40.5.1 What is a "false positive" warning? . . . 301
40.5.2 How can I improve the Checker Framework to eliminate a false positive warning? . . . 301
40.5.3 Why doesn't the Checker Framework infer types for fields and method return types? . . . 301
40.5.4 Why are some operations legal in constructors but not in methods? . . . 302
40.5.5 Why doesn't the Checker Framework track relationships between variables? . . . 303
40.5.6 Why isn't the Checker Framework path-sensitive? . . . 304
40.6 Syntax of type annotations . . . 305
40.6.1 What is a "receiver"? . . . 305
40.6.2 What is the meaning of an annotation after a type, such as `@NonNull Object @Nullable`? . . . 306
40.6.3 What is the meaning of array annotations such as `@NonNull Object @Nullable []`? . . . 306
40.6.4 What is the meaning of varargs annotations such as `@English String @NonEmpty ...`? . . . 306
40.6.5 What is the meaning of a type qualifier at a class declaration? . . . 306
40.6.6 How are type qualifiers written on upper and lower bounds? . . . 306
40.6.7 Why shouldn't a qualifier apply to both types and declarations? . . . 306
40.6.8 How do I annotate a fully-qualified type name? . . . 307
40.6.9 What is the difference between type annotations and declaration annotations? . . . 307
40.6.10 How should type annotations be formatted in source code? Where should I write type annotations? 308
40.6.11 How does the Checker Framework handle obsolete declaration annotations? . . . 309
40.6.12 How do I convert from other tools' declaration annotations to type annotations? . . . 310
40.7 Semantics of type annotations . . . 310
40.7.1 How can I handle typestate, or phases of my program with different data properties? . . . 310
40.7.2 Why are explicit and implicit bounds defaulted differently? . . . 311

40.7.3 How should I annotate code that uses generics? . . . 311
40.7.4 Why are type annotations declared with `@Retention(RetentionPolicy.RUNTIME)`? . . . 312
40.8 Creating a new checker . . . 312
40.8.1 How do I create a new checker? . . . 312
40.8.2 What properties can and cannot be handled by type-checking? . . . 312
40.8.3 Why is there no declarative syntax for writing type rules? . . . 313
40.9 Tool questions . . . 313
40.9.1 How does pluggable type-checking work? . . . 313
40.9.2 What classpath is needed to use an annotated library? . . . 313
40.9.3 Why do `.class` files contain more annotations than the source code? . . . 313
40.9.4 Is there a type-checker for managing checked and unchecked exceptions? . . . 314
40.9.5 The Checker Framework runs too slowly . . . 314
40.9.6 What does the Checker Framework version number mean? . . . 315
40.10 Relationship to other tools . . . 315
40.10.1 Why not just use a bug detector (like SpotBugs or Error Prone)? . . . 315
40.10.2 How does the Checker Framework compare with Eclipse's null analysis? . . . 316
40.10.3 How does the Checker Framework compare with NullAway? . . . 316
40.10.4 How does the Checker Framework compare with JSpecify? . . . 316
40.10.5 How does the Checker Framework compare with the EISOP Checker Framework? . . . 317
40.10.6 How does the Checker Framework compare with the JDK's `Optional` type? . . . 317
40.10.7 How does pluggable type-checking compare with JML? . . . 317
40.10.8 Is the Checker Framework an official part of Java? . . . 318
40.10.9 What is the relationship between the Checker Framework and JSR 305? . . . 318
40.10.10 What is the relationship between the Checker Framework and JSR 308? . . . 318

41 Troubleshooting, getting help, and contributing 319
41.1 Common problems and solutions . . . 319
41.1.1 Unable to compile the Checker Framework . . . 319
41.1.2 Unable to run the checker, or checker crashes . . . 319
41.1.3 Unexpected warnings not related to type-checking . . . 321
41.1.4 Unexpected type-checking results . . . 321
41.1.5 Unexpected compilation output when running javac without a pluggable type-checker . . . 324
41.2 How to report problems (bug reporting) . . . 324
41.2.1 Problems with annotated libraries . . . 325
41.3 Building from source . . . 325
41.3.1 Install prerequisites . . . 325
41.3.2 Obtain the source . . . 327
41.3.3 Build the Checker Framework . . . 327
41.3.4 Build the Checker Framework Manual (this document) . . . 328
41.3.5 Code style, IDE configuration, pull requests, etc. . . . 328
41.3.6 Enable continuous integration builds . . . 328
41.3.7 Related repositories . . . 328
41.4 Contributing . . . 329
41.4.1 Contributing fixes (creating a pull request) . . . 329
41.5 Credits and changelog . . . 329
41.6 License . . . 330
41.7 Publications . . . 330

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 compile-time 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 31)
2. Initialization Checker to ensure all `@NonNull` fields are set in the constructor (see Section 3.8, page 44)
3. Map Key Checker to track which values are keys in a map (see Chapter 4, page 54)
4. Optional Checker for errors in using the `Optional` type (see Chapter 5, page 59)
5. Interning Checker for errors in equality testing and interning (see Chapter 6, page 62)
6. Called Methods Checker for the builder pattern (see Chapter 7, page 68)
7. Resource Leak Checker for ensuring that resources are disposed of properly (see Chapter 8, page 73)
8. Fake Enum Checker to allow type-safe fake enum patterns and type aliases or typedefs (see Chapter 9, page 83)
9. Tainting Checker for trust and security errors (see Chapter 12, page 108)
10. SQL Quotes Checker to mitigate SQL injection attacks (see Chapter 13, page 111)
11. Lock Checker for concurrency and lock errors (see Chapter 10, page 87)
12. Index Checker for array accesses (see Chapter 11, page 98)
13. Regex Checker to prevent use of syntactically invalid regular expressions (see Chapter 14, page 113)
14. Format String Checker to ensure that format strings have the right number and type of `%` directives (see Chapter 15, page 116)
15. Internationalization Format String Checker to ensure that i18n format strings have the right number and type of `{}` directives (see Chapter 16, page 123)
16. Property File Checker to ensure that valid keys are used for property files and resource bundles (see Chapter 17, page 129)
17. Internationalization Checker to ensure that code is properly internationalized (see Section 17.2, page 130)
18. Signature String Checker to ensure that the string representation of a type is properly used, for example in `Class.forName` (see Chapter 18, page 132)
19. GUI Effect Checker to ensure that non-GUI threads do not access the UI, which would crash the application (see Chapter 19, page 135)
20. Units Checker to ensure operations are performed on correct units of measurement (see Chapter 20, page 140)
21. Signedness Checker to ensure unsigned and signed values are not mixed (see Chapter 21, page 144)
22. Modifiability Checker to warn about a possible `UnsupportedOperationException` at run time when modifying a collection (see Chapter 22, page 148)
23. Purity Checker to identify whether methods have side effects (see Chapter 23, page 157)
24. Constant Value Checker to determine whether an expression's value can be known at compile time (see Chapter 24, page 161)
25. Reflection Checker to determine whether an expression's value (of type `Method` or `Class`) can be known at

compile time (see Chapter 26, page 168)
26. Initialized Fields Checker to ensure all fields are set in the constructor (see Chapter 27, page 173)
27. Aliasing Checker to identify whether expressions have aliases (see Chapter 28, page 176)
28. Must Call Checker to over-approximate the methods that should be called on an object before it is de-allocated (see Chapter 29, page 180)
29. Subtyping Checker for customized checking without writing any code (see Chapter 30, page 183)
30. Third-party checkers that are distributed separately from the Checker Framework (see Chapter 31, page 188)

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 30 and 37.

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 3–31).
- Skim the advanced material that will enable you to make more effective use of a type system (Chapters 32–41), so that you will know what is available and can find it later. Skip Chapter 37 on creating a new checker.

1.2 How it works: Pluggable types

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 define new type systems and run them 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 part of your program, or the whole thing. Type inference tools exist to help you annotate your code; see Chapter 35.

Most programmers will use type systems created by other people, such as those listed at the start of the introduction (Chapter 1, page 15). Some people, called "type system designers", create new type systems (Chapter 37, page 250). 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" and "type-checking compiler plugin" as synonyms.

1.3 Installation

This section describes how to install the Checker Framework.

- If you use a **build system** that automatically downloads dependencies, such as Gradle or Maven, **no installation is necessary**; just see Chapter 39, page 279.
- If you wish to try the Checker Framework without installing it, use the Checker Framework Live Demo webpage.
- This section describes how to install the Checker Framework from its distribution. The Checker Framework release contains everything that you need, both to run checkers and to write your own checkers.
- Alternately, you can build the latest development version from source (Section 41.3, page 325).

Requirement: You must have a JDK (version 17 or later) installed.
The installation process has two required steps and one optional step.

1. Download the Checker Framework distribution:
   `https://checkerframework.org/checker-framework-4.4.0.zip`
   For example, on Unix you can run: `wget https://checkerframework.org/checker-framework-4.4.0.zip`

2. Unzip it to create a `checker-framework-4.4.0` directory.
   For example, on Unix you can run: `unzip checker-framework-4.4.0.zip`
3. Configure your IDE, build system, or command shell to include the Checker Framework on the classpath. Choose the appropriate section of Chapter 39.

Now you are ready to start using the checkers.

We recommend that you work through the Checker Framework tutorial (`https://checkerframework.org/tutorial/`), which demonstrates the Nullness, Regex, and Tainting Checkers.

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

The Checker Framework is released on a monthly schedule. The minor version (the middle number in the version number) is incremented if there are any incompatibilities with the previous version, including in user-visible behavior or in methods that a checker implementation might call.

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 you can work along with.

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();
    }
}
```

If you run the Nullness Checker (Chapter 3), the compilation completes without any errors.

Now, introduce an error. Modify `ref`'s assignment to:

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

If you run the Nullness Checker again, it emits the following error:

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

This is a trivially simple example. Even an unsound bug-finding tool like SpotBugs or Error Prone could have detected this bug. The Checker Framework's analysis is more powerful than those tools and detects more code defects than they do.

Type qualifiers such as `@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 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 infer annotations that are consistent with their code: see Chapter 35.) It is possible to annotate only part of your code: see Section 2.4.6.
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 33: Advanced type system features
- Chapter 34: Suppressing warnings
- Chapter 36: Annotating libraries
- Chapter 37: How to create a new checker
- Chapter 39: Integration with external tools

There is a tutorial (`https://checkerframework.org/tutorial/`) that walks you through using the Checker Framework on the command line.

2.1 Where to write type annotations

You may write a type annotation 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 type annotations on method signatures, fields, and some type arguments. Most annotations within method bodies are inferred for you; for more details, see Section 33.7.

The Java Language Specification also defines declaration annotations, such as `@Deprecated` and `@Override`, which apply to a class, method, or field but do not apply to the method's return type or the field's type. They should be written on their own line in the source code, before the method's signature.

2.2 Running a checker

To run a checker, run the compiler `javac` as usual, but either pass the `-processor` *`plugin_class`* command-line option, or use auto-discovery as described in Section 2.2.3. (If your project already uses auto-discovery for some annotation processor, such as AutoValue, then you should use auto-discovery.) Two concrete examples of using `-processor` to run the Nullness Checker are

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

where `javac` is as specified in Section 39.6.

You can also run a checker from within your favorite IDE or build system. See Chapter 39 for details about build tools such as Ant (Section 39.3), Buck (Section 39.5), Bazel (Section 39.4), Gradle (Section 39.9), Maven (Section 39.13), and sbt (Section 39.15); IDEs such as Eclipse (Section 39.8), IntelliJ IDEA (Section 39.10), NetBeans (Section 39.14), and tIDE (Section 39.16); 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 and those created by another annotation processor. It may also include other Java files of yours, if they are more recent than the corresponding `.class` file. 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. Type-checking works modularly and intraprocedurally: when verifying a method, it examines only the signature (including annotations) of other methods, not their implementations. When analyzing a variable use, it relies on the type of the variable, not any dataflow outside the current method that produced the value.

After you compile your code while running a checker, the resulting `.class` and `.jar` files can be used for pluggable type-checking of client code.

If you compile code without the `-processor` command-line option, 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, to create `.class` files that will be distributed or compiled against, you should run the type-checkers for all the annotations that you 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 or in a stub file.

Some projects are already distributed with type annotations by their maintainers, so you do not need to do anything special. An example is all the libraries in `https://github.com/plume-lib/`. Over time, this should become more common.

For some other libraries, the Checker Framework developers have provided an annotated version of the library, either as a stub file or as compiled class files. (If some library is not available in either of these forms, you can contribute by annotating it, which will help you and all other Checker Framework users; see Chapter 36, page 239.)

Some stub files are used automatically by a checker, without any action on your part. For others, you must pass the `-Astubs=...` command-line argument. As a special case, if an `.astub` file appears in `checker/src/main/resources/`, then pass the command-line option `-Astubs=checker.jar/`*`stubfilename.astub`*. The "`checker.jar`" should be literal — don't provide a path. This special syntax only works for "`checker.jar`".

The annotated libraries that are provided as class files appear in the `org.checkerframework.annotatedlib` group in the Maven Central Repository. The annotated library has *identical* behavior to the upstream, unannotated version;

the source code is identical other than added annotations. (Some of the annotated libraries are bcel, commons-csv, commons-io, guava, and java-getopt.)

To use an annotated library:

- If your project stores `.jar` files locally, then download the `.jar` file from the Maven 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("org.apache.bcel:bcel:6.3.1")
  api("commons-io:commons-io:2.8")
  ```

  to

  ```
  api("org.checkerframework.annotatedlib:bcel:6.3.1")
  api("org.checkerframework.annotatedlib:commons-io:2.8.0.1")
  ```

  Usually 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 but that version is not available in `org.checkerframework.annotatedlib`, then open an issue requesting that the `org.checkerframework.annotatedlib` version be updated.

There is one special case. If an `.astub` file is shipped with the Checker Framework in `checker/src/main/resources/`, then you can use `-Astubs=checker.jar/`*`stubfilename.astub`*. The "`checker.jar`" should be literal — don't provide a path. (This special syntax only works for "`checker.jar`".)

2.2.2 Summary of command-line options

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

To pass an option to only a particular checker, prefix the option with the canonical or simple name of a checker, followed by an underscore "_". Such an option will apply only to a checker with that name or any subclass of that checker. For example, you can use

```
-ANullnessChecker_lint=redundantNullComparison
-Aorg.checkerframework.checker.guieffect.GuiEffectChecker_lint=debugSpew
```

to pass different lint options to the Nullness and GUI Effect Checkers. A downside is that, in this example, the Nullness Checker will issue a "The following options were not recognized by any processor" warning about the second option and the GUI Effect Checker will issue a "The following options were not recognized by any processor" warning about the first option.

Unsound checking: ignore some errors

- `-AsuppressWarnings` Suppress all errors and warnings matching the given key; see Section 34.3.
- `-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 34.4.
- `-AskipDefs`, `-AonlyDefs` Suppress all errors and warnings within the definition of given classes — or everywhere except within the definition of given classes. See Section 34.5.
- `-AskipFiles`, `-AonlyFiles` Suppress all errors and warnings within given files or directories/folders — or everywhere except within given files or directories/folders. See Section 34.6.
- `-AassumeSideEffectFree`, `-AassumeDeterministic`, `-AassumePure`, `-AassumePureGetters` Unsoundly assume that every method is side-effect-free, deterministic, or both; or that every getter method is pure. See Section 23.5 and Section 33.7.5.
- `-AassumeAssertionsAreEnabled`, `-AassumeAssertionsAreDisabled` Whether to assume that assertions (Java `assert` statements) are enabled or disabled; see Section 33.7.6.

- `-AignoreRangeOverflow` Ignore the possibility of overflow for range annotations such as `@IntRange`; see Section 24.4.
- `-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.
- `-AignoreInvalidAnnotationLocations` Ignore annotations in bytecode that have invalid annotation locations.

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

- `-AcheckPurityAnnotations` Check the bodies of methods and constructors marked `@SideEffectFree`, `@Deterministic`, and `@Pure` to ensure the method or constructor satisfies the annotation. By default, the Checker Framework unsoundly trusts the annotation. See Section 33.7.5.
- `-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 33.1.
- `-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 32.1.7 and Section 33.1.
- `-AuseConservativeDefaultsForUncheckedCode` Enables conservative defaults, and suppresses all type-checking warnings, in unchecked code. Takes arguments "source,bytecode". "-source,-bytecode" is the (unsound) default setting.
  - "bytecode" specifies whether the checker should apply conservative defaults to bytecode (that is, to already-compiled libraries); see Section 33.5.6.
  - Outside the scope of any relevant `@AnnotatedFor` annotation, "source" specifies whether conservative default annotations are applied to source code and suppress all type-checking warnings; see Section 36.4.
- `-AconcurrentSemantics` Whether to assume concurrent semantics (field values may change at any time) or sequential semantics; see Section 40.4.6.
- `-AignoreRawTypeArguments=false` Do not ignore subtype tests for type arguments that were inferred for a raw type. See Section 32.1.1.
- `-processor org.checkerframework.common.initializedfields.InitializedFieldsChecker,...` Ensure that all fields are initialized by the constructor. See Chapter 27, page 173.

Type-checking modes: enable/disable functionality

- `-Alint` Enable or disable optional checks; see Section 34.7.
- `-AwarnRedundantAnnotations` Warn about redundant annotations. A warning is issued if an explicitly written annotation is the same as the default annotation for that location. This feature does not warn about all redundant annotations, only some.
- `-AsuggestPureMethods` Suggest methods that could be marked `@SideEffectFree`, `@Deterministic`, or `@Pure`; see Section 33.7.5.
- `-AresolveReflection` Determine the target of reflective calls, and perform more precise type-checking based on that information; see Chapter 26. `-AresolveReflection=debug` causes debugging information to be output.
- `-Ainfer=`*`outputformat`* Output suggested annotations for method signatures and fields. These annotations may reduce the number of type-checking errors on subsequent type-checking runs. This option is typically used by whole-program inference (WPI; see Section 35.2) rather than by programmers. Using `-Ainfer=jaifs` produces `.jaif` files. Using `-Ainfer=stubs` produces `.astub` files. Using `-Ainfer=ajava` produces `.ajava` files. You must also supply `-Awarns`, or the inference output may be incomplete.
- `-AinferOutputDirectory` The directory into which to write inference output (e.g., `.ajava` files). It defaults to `build/whole-program-inference`. When using a build system (e.g., Gradle), the default may be interpreted relative to a build system directory (e.g., `$HOME/.gradle/workers`) rather than the project root directory, so you may need to provide an absolute path.
- `-AinferOutputOriginal` When outputting `.ajava` files when running with `-Ainfer=ajava`, also output a copy of the original file with no inferred annotations, but with the formatting of a `.ajava` file, to permit use of `diff` to view the inferred annotations. Must be combined with `-Ainfer=ajava`.

- `-AshowSuppressWarningsStrings` With each warning, show all possible strings to suppress that warning.
- `-AwarnUnneededSuppressions` Issue an `unneeded.suppression` warning for each `@SuppressWarnings` that did not suppress a warning issued by the checker. This only warns about `@SuppressWarnings` strings that contain a checker name other than `"allcheckers"` (for syntax, Section 34.1.1). The `-ArequirePrefixInWarningSuppressions` command-line argument ensures that all `@SuppressWarnings` strings contain a checker name. An `unneeded.suppression` warning can be suppressed only by `@SuppressWarnings("unneeded.suppression")` or `@SuppressWarnings("checkername`, not by `@SuppressWarnings("checkername")` and not by a partial message key such as `@SuppressWarnings("suppression"`
- `-AwarnUnneededSuppressionsExceptions=regex` disables `-AwarnUnneededSuppressions` for `@SuppressWarnings` strings that contain a match for the regular expression. Most users don't need this.
- `-ArequirePrefixInWarningSuppressions` Require that the string in a warning suppression annotation begin with a checker name. Otherwise, the warning suppression annotation does not suppress any warnings. For example, if this command-line option is supplied, then `@SuppressWarnings("assignment")` has no effect, but `@SuppressWarnings("nullness:assignment")` does.
- `-AshowPrefixInWarningMessages` When issuing an error or warning, prefix the warning suppression key by the checker name. For instance, output "error: [nullness:assignment] ..." instead of "error: [assignment]". This makes it easy to tell, from the suppression key only, which checker issued the error or warning.

Partially-annotated libraries

- `-Astubs` List of stub files or directories; see Section 36.5.1.
- `-AstubWarnIfNotFound`, `-AstubNoWarnIfNotFound`, `-AstubWarnIfNotFoundIgnoresClasses`, `-AstubWarnIfRedundantWi` `-AstubWarnNote` Warn about problems with stub files; see Section 36.5.7.
- `-AmergeStubsWithSource` If both a stub file and a source file for a class are available, trust both and use the greatest lower bound of their annotations. The default behavior (without this flag) is to ignore types from the stub file if source is available. See Section 36.5.2.
- `-AuseConservativeDefaultsForUncheckedCode=source` Outside the scope of any relevant `@AnnotatedFor` annotation, use conservative default annotations and suppress all type-checking warnings; see Section 36.4.

Debugging

- `-AprintAllQualifiers`, `-AprintVerboseGenerics`, `-Anomsgtext`, `-Aonelinemsg`, `-AdumpOnErrors`, `-AexceptionLineSep` Amount of detail in messages; see Section 37.12.1.
- `-Adetailedmsgtext` Format of diagnostic messages; see Section 37.12.2.
- `-Aignorejdkastub`, `-ApermitMissingJdk`, `-AparseAllJdk`, `-AstubDebug` Stub and JDK libraries; see Section 37.12.3.
- `-Afilenames`, `-Ashowchecks`, `-AshowWpiFailedInferences` Progress tracing; see Section 37.12.4.
- `-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 (such as a build system). That command line makes it easier to reproduce, report, and debug issues. For example, the command line can be modified to enable attaching a debugger. See Section 37.12.5.
- `-Aflowdotdir`, `-Averbosecfg`, `-Acfgviz` Draw a visualization of the CFG (control flow graph); see Section 37.12.6.
- `-AresourceStats`, `-AatfDoNotCache`, `-AatfCacheSize` Miscellaneous debugging options; see Section 37.12.7.
- `-AslowTypecheckingSeconds=N` Print a warning for any program construct, such as a method or class, whose type-checking takes more than N seconds (default 45). Often, generic type inference is the slowest part of type-checking, and you can significantly speed up type-checking by explicitly writing a few generic type arguments.
- `-Aversion` Print the Checker Framework version.
- `-AprintGitProperties` Print information about the git repository from which the Checker Framework was compiled.

Some checkers support additional options, which are described in that checker's manual section. For example, `-Aquals` tells the Subtyping Checker (see Chapter 30) 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 "processor" or "proc", because in javac jargon, a checker is an "annotation processor".

- `-processor` Names the checker to be run; see Sections 2.2 and 2.2.4. May be a comma-separated list of multiple checkers. Note that javac stops processing an indeterminate time after detecting an error. When providing multiple checkers, if one checker detects any error, subsequent checkers may not run.
- `-processorpath` Indicates where to search for the checker. This should also contain any classes used by type-checkers, such as qualifiers used by the Subtyping Checker (see Section 30.2) and classes that define statically-executable methods used by the Constant Value Checker (see Section 24.2.2).
- `-proc:{none,only}` Controls whether checking happens; `-proc:none` means to skip checking; `-proc:only` means to do only checking, without any subsequent compilation; see Section 2.2.3.
- `-implicit:class` Suppresses warnings about implicitly compiled files (not named on the command line); see Section 39.3.
- `-J` Supply an argument to the JVM that is running javac; for example, `-J-Xmx4g` 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.

2.2.3 Checker auto-discovery

"Auto-discovery" makes the `javac` compiler always run an annotation processor, such as a checker plugin, without explicitly passing the `-processor` command-line option. This can make your command line shorter, and it ensures that your code is checked even if you forget the command-line option.

If the `javac` command line specifies any `-processor` command-line option, then auto-discovery is disabled. This means that if your project currently uses auto-discovery, you should use auto-discovery for the Checker Framework too. (Alternately, if you prefer to use a `-processor` command-line argument, you will need to specify all annotation processors, including ones that used to be auto-discovered.)

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 checkers to be used, listed one per line. For instance, to run the Nullness Checker and the Interning Checker automatically, the configuration file should contain:

```
org.checkerframework.checker.nullness.NullnessChecker
org.checkerframework.checker.interning.InterningChecker
```

You can disable this auto-discovery mechanism by passing the `-proc:none` command-line option to `javac`, which disables all annotation processing including all pluggable type-checking.

2.2.4 Shorthand for built-in checkers

Ordinarily, javac's `-processor` flag requires fully-qualified class names. When using the Checker Framework javac wrapper (Section 39.6), 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 also works when multiple checkers are specified. Their names are separated by commas, with no surrounding space. 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 guarantees two things: type annotations reflect facts about run-time values, and illegal operations are not performed.

For example, the Nullness Checker (Chapter 3) guarantees lack of null pointer exceptions (Java `NullPointerException`). More precisely, it guarantees that expressions whose type is annotated with `@NonNull` never evaluate to null, and it forbids other expressions from being dereferenced.

As another example, the Interning Checker (Chapter 6) guarantees that correct equality tests are performed. More precisely, it guarantees that every expression whose type is an `@Interned` type evaluates to an interned value, and it forbids == on other expressions.

The guarantee holds only if you run the checker on every part of your program and the checker issues no warnings anywhere in the code. You can also verify just part of your program.

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. This includes external libraries supplied as a `.class` file and native methods (because the implementation is not Java code, it cannot be checked).
  - Code compiled with the `-AskipUses`, `-AonlyUses`, `-AskipDefs`, `-AonlyDefs`, `-AskipFiles`, or `-AonlyFiles` command-line arguments (see Chapter 34).
  - Dynamically generated code, such as code generated by Spring or MyBatis. Its bytecode is directly generated and run, not compiled by javac and not visible to the Checker Framework.

  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.
- You can suppress warnings, such as via the `@SuppressWarnings` annotation (Chapter 34, page 222). If you do so incorrectly, the checker's guarantee no longer holds.
- The Checker Framework is, by default, unsound in a few places where a conservative analysis would issue too many false positive warnings. These are listed in Section 2.2.2. You can supply a command-line argument to make the Checker Framework sound for each of these cases.
- Specific checkers may have other limitations; see their documentation for details.

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

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

The second call to `toString` cannot possibly throw a null pointer exception — `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. (Often, 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 41.2).

2.4 Tips about writing annotations

Section 36.1 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. While adding annotations, you will spend most of your time understanding the code, and less time actually writing annotations or running the checker.

Don't annotate the whole program, but work module by module. Start annotating classes at the leaves of the call tree — that is, start with classes/packages that have few dependencies on other code. Annotate supertypes before you annotate 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. Sections 34.4 and 34.5 give ways to skip checking of some files, directories, or packages. Section 2.4.6 gives advice about handling calls from annotated code into unannotated code.

When annotating, be systematic; we recommend annotating an entire class or module 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 use means you determined that the default is desirable, or it means you didn't yet examine that type use.

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 40.1.2).

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 (Chapter 3), then you can search the documentation for "null" and then add `@Nullable` anywhere appropriate. Start by annotating signatures and fields, but not 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`.

The specification should state all facts that are relevant to callers. When checking a method, the checker uses only the specification, not the implementation, of other methods. (Equivalently, type-checking is "modular" or "intraprocedural".) When analyzing a variable use, the checker relies on the type of the variable, not any dataflow outside the current method that produced the value.

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 (assuming that you understand the code, of course).

It is usually not a good idea to experiment with adding and removing annotations in order to understand their effect. It is better to reason about the desired design. However, to avoid having to manually examine all callers, a more automated approach is to save the checker output before changing an annotation, then compare it to the checker output after changing the annotation.

Chapter 36 tells you how to annotate libraries that your code uses. Section 2.4.5 and Chapter 34 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 clear enough for the type-checker to verify, then it will also be easier for programmers to understand. When you verify your code, a side benefit is improving your code's structure.

Your code should compile cleanly under the regular Java compiler. As a specific example, your code should not use raw types like `List`; use parameterized types like `List<String>` instead (Section 32.1.1). If you suppress Java compiler warnings, then the Checker Framework will issue more warnings, and its messages will be more confusing. (Also, if you are not willing to write code that type-checks in Java, then you might not be willing to use an even more powerful type system.)

Do not write unnecessary annotations.

- Do not annotate local variables unless necessary. The checker infers annotations for local variables (see Section 33.7). 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 (usually on type arguments or array element types, rather than on top-level types).
- Do not write annotations that are redundant with defaults. For example, when checking nullness (Chapter 3, page 31), the default annotation is `@NonNull`, in most locations other than some type bounds (Section 33.5.3). 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 2.4.5 discusses what to do when a checker issues a warning about your code.

2.4.3 Annotations indicate non-exceptional 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.

Methods that crash when passed certain values

Nullness example As an example, consider the Nullness Checker. Its goal is to guarantee that your program does not crash due to a null value.

This method crashes if `null` is passed to it:

```
/** @throws NullPointerException if arg is null */
void m1(Object arg) {
  arg.toString();
  ...
}
```

Therefore, the type of `arg` should be `@NonNull Object` — you can write this as just `Object`, because `@NonNull` is the default. The Nullness Checker (Chapter 3, page 31) prevents null pointer exceptions by warning you whenever a client passes a value that might cause `m1` to crash.

Here is another method:

```
/** @throws NullPointerException if arg is null */
void m2(Object arg) {
  Objects.requireNonNull(arg);
  ...
}
```

Method `m2` behaves just like `m1` in that it throws `NullPointerException` if a client passes `null`. Therefore, their specifications should be identical (the formal parameter type is annotated with `@NonNull`), so the checker will issue the same warning if a client might pass `null`.

The same argument applies to any method that is guaranteed to throw an exception if it receives `null` as an argument. Examples include:

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

Their formal parameter types are annotated as `@NonNull`, because otherwise the program might crash. Adding a call to a method like `requireNonNull` never prevents a crash: your code still crashes, but with a slightly different stack trace. In order to prevent all exceptions in your program caused by null pointers, you need to prevent those thrown by methods including `requireNonNull`.

(One might argue that the formal parameter should be annotated as `@Nullable` because passing `null` has a well-defined semantics (throw an exception) and such an execution may be possible if your program has a bug. However, it is never the programmer's intent for `null` to flow there. Preventing such bugs is the purpose of the Nullness Checker.)

A method like `requireNonNull` is useless for making your code correct, but it does have a benefit: its stack trace may help developers to track down the bug. (For users, the stack trace is scary, confusing, and usually non-actionable.) But if you are using the Checker Framework, you can prevent errors rather than needing extra help in debugging the ones that occur at run time.

Optional example Another example is the Optional Checker (Chapter 5, page 59) and the `orElseThrow()` method. The goal of the Optional Checker is to ensure that the program does not crash due to use of a non-present Optional value. Therefore, the receiver of `orElseThrow()` is annotated as `@Present`, and the Optional Checker issues an error if the client calls `orElseThrow()` on a `@MaybePresent` value. (For details, see Section 5.3.)

Permitting crashes in some called methods You can make a checker ignore crashes in library code, such as `assertNotNull()`, that occur as a result of misuse by your code. This invalidates the checker's guarantee that your program will not crash. (Programmers and users typically care about all crashes, no matter which method is at the top of the call stack when the exception is thrown.) The checker will still warn you about crashes in your own code.

- The `-AskipUses` command-line argument (Section 34.4) skips checking all method calls to one or more classes.
- A stub file (Section 36.5) can override the library's annotations, for one or more methods.
  As a special case, if you want the Nullness Checker to prevent most null pointer exceptions in your code, but to *permit* null pointer exceptions at nullness assertion methods, you can pass `-Astubs=permit-nullness-assertion-exception.astu`
- Don't type-check clients of the method. For example, JUnit's `assertNotNull()` is typically called only in test code; its clients are the tests. If you type-check only your main program, then the annotation on `assertNotNull()` is irrelevant.

Methods that sometimes crash when passed certain values

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` (the annotation was subsequently changed to the more correct `@ParametricNullness`):

```
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 the 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 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! If you have trouble understanding a Checker Framework warning message, you can search for its text in this manual. There are three general causes for the warnings:

You found a bug There is a bug in your code, such as a possible null dereference. Fix your code to prevent that crash.

Wrong annotations The annotations are too strong (they are incorrect) or too weak (they are imprecise). Improve the annotations, usually by writing more annotations in order to better express the specification. Only write annotations that accurately describe the intended behavior of the software — don't write inaccurate annotations just for the purpose of eliminating type-checker warnings.

Usually you need to improve the annotations in your source code. Sometimes you need to improve annotations in a library that your program uses (see Chapter 36, page 239).

Type-checker weakness 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. (Recall that the checker works modularly: when type-checking a method `m`, it relies on the types and signatures of variables and methods used by `m`, but not the initialization expressions or the method bodies.)

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 34, page 222); be sure to include a code comment explaining how you know the code is correct even though the type-checker cannot deduce that fact.

Do not add an `if` test that can never fail, just to suppress a warning. Adding a gratuitous `if` clutters the code and confuses readers. A reader should assume that every `if` condition can evaluate to true or false. There is one exception to this rule: an `if` test may have a condition that you think will never evaluate to true, if its body just throws a descriptive error message.

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. (Also see Section 41.1.4 and Chapter 41, Troubleshooting. In particular, Section 41.1.4 explains this same methodology in different words.)

Step 1: Explain correctness: write a proof Write an explanation of why your code is correct and why it never suffers the error at run time. In other words, this is an informal proof that the type-checker's warning is incorrect. Write it in natural language, such as English.

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).

Step 2: Translate the proof into annotations Here are some examples of the translation.

- 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)`.
- If your proof includes "method `m` has no side effects", then annotate `m`'s declaration with `@SideEffectFree`.
- If your proof includes "each call to method `m` returns the same value", then annotate `m`'s declaration with `@Deterministic`.

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 36, page 239.

Don't omit any parts of your proof. When the Checker Framework analyzes a method, it examines only the signature/specification (not the implementation) of other methods.

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 indices might be `null`". In this case, you have two choices. First, you can suppress the warning (Chapter 34, page 222); be sure to write a comment explaining your reasoning for suppressing the warning. You may wish to submit a feature request (Section 41.2) 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.

Step 3: Re-run the checker 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 34, page 222). The comment is an excerpt from your informal proof, and the proof guides you to the best place to suppress the warning. Please submit a bug report so that the checker can be improved in the future (Section 41.2).

2.4.6 Calls to unannotated code (legacy libraries)

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. Or, the code you are annotating may call into unannotated libraries.

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 Object 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 call *never* returns null, you have two choices: annotate the library or suppress warnings.

1. To annotate the library:
   - If the unannotated code is in your program, you can write annotations but not type-check them yet. Two ways to prevent the type-checking are via a `@SuppressWarnings` annotation (Section 34.1) or not running the checker on that file, for example via the `-AskipDefs` command-line option (Section 34.5).
   - To annotate a library whose source code you do not have or cannot change, see Chapter 36.
2. To suppress all warnings related to uses of `unannotated_method`, use the `-AskipUses` command-line option (Section 34.4). Beware: a carelessly-written regular expression may suppress more warnings than you intend.

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. In other words, the Nullness Checker prevents all `NullPointerException`s. See Section 3.1 for more details about the guarantee and what is checked.

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

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

The NullnessChecker is actually an ensemble of three pluggable type-checkers that work together: the Nullness Checker proper (which is the main focus of this chapter), the Initialization Checker (Section 3.8), and the Map Key Checker (Chapter 4, page 54). Their type hierarchies are completely independent, but they work together to provide precise nullness checking.

3.1 What the Nullness Checker guarantees

If the Nullness Checker type-checks your program without errors, then your program will not crash with a `NullPointerException` that is caused by misuse of `null` in checked code. Section 2.3 notes some limitations to guarantees made by the Checker Framework.

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 a `@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 initialization (see Section 3.8) and of map key annotations (see Chapter 4, page 54).

The checker performs additional checks if certain `-Alint` command-line options are provided. (See Section 34.7 for more details about the `-Alint` command-line option.)

3.1.1 Nullness Checker optional warnings

1. Options that control soundness:
   - If you supply the `-Alint=soundArrayCreationNullness` 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.
   - If you supply the `-Astubs=collection-object-parameters-may-be-null.astub` command-line option, then in JDK collection classes, the checker unsoundly permits null as an argument for any key or value formal parameter whose type is `Object` (instead of the element type). See Section 3.4.2.
   - If you supply the `-Alint=trustArrayLenZero` command-line option, then the checker will trust `@ArrayLen(0)` annotations. See Section 3.3.5 for a discussion.
   - If you supply the `-AassumeKeyFor` command-line option, then the checker will unsoundly assume that the argument to `Map.get` is a key for the receiver map. It will not do any checking of `@KeyFor` and related qualifiers.
   - If you supply the `-AinvocationPreservesArgumentNullness` command-line option, the Nullness Checker unsoundly assumes that arguments passed to non-null parameters in an invocation remain non-null after the invocation. This assumption is unsound in general, but holds for most code. For other ways to suppress warnings related to method side effects, see Section 33.7.5.
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. Options that enable checking modes:
   - If you supply the `-Alint=permitClearProperty` command-line option, then the checker permits calls to `System.setProperties()` and calls to `System.clearProperty` that might clear one of the built-in properties.
     By default, the checker forbids calls to those methods, and also special-cases type-checking of calls to `System.getProperty()` and `System.setProperties()`. A call to one of these methods can return null in general, but by default the Nullness Checker treats it as returning non-null if the argument is one of the literal strings listed in the documentation of `System.getProperties()`. To make this behavior sound, the Nullness Checker forbids calls that might clear any built-in property, as described above.

3.2 Nullness annotations

The Nullness Checker uses three separate type hierarchies: one for nullness, one for initialization (Section 3.8), and one for map keys (Chapter 4, page 54). The Nullness Checker has four varieties of annotations: nullness type qualifiers, nullness method annotations, initialization type qualifiers, and map key type qualifiers.

3.2.1 Nullness qualifiers

The nullness hierarchy contains these qualifiers:

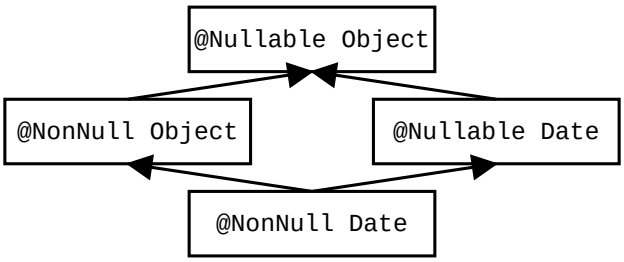


Figure 3.1: Partial type hierarchy for the Nullness type system. Java's `Object` is expressed as `@Nullable Object`. Programmers can omit most type qualifiers, because the default annotation (Section 3.3.2) is usually correct. The Nullness Checker verifies three type hierarchies: this one for nullness, one for initialization (Section 3.8), and one for map keys (Chapter 4, page 54).

`@Nullable` indicates a type that includes the null value. For example, the Java type `Boolean` is nullable: a variable of type `Boolean` always has one of the values `TRUE`, `FALSE`, or `null`. (Since `@NonNull` is the default type annotation, you would actually write this type as `@Nullable Boolean`.)

`@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 qualifier polymorphism, see Section 32.2.

`@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, for field initialization that occurs in a lifecycle method other than the constructor (e.g., an override of `android.app.Activity.onCreate`), and other uses. `@MonotonicNonNull` is typically written on field types, but not elsewhere.
The benefit of `@MonotonicNonNull` over `@Nullable` is that 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. By contrast, for a `@Nullable` field, the Nullness Checker assumes that most method calls might set it to `null`. (Exceptions are calls to methods that are `@SideEffectFree`, or that have an `@EnsuresNonNull` or `@EnsuresNonNullIf` annotation.)
A `@MonotonicNonNull` field may be initialized 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=noInitForMonotonicNonNu` on the command line), then `@MonotonicNonNull` fields are not allowed to have initializers at their declarations.
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.
In the type system, `@MonotonicNonNull` is a supertype of `@NonNull` and a subtype of `@Nullable`.

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.)

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 to be non-null when the method is invoked. Don't use this for formal parameters (just annotate their type as `@NonNull`). `@RequiresNonNull` is appropriate for a field that is `@Nullable` in general, but some method requires the field to be non-null.

@EnsuresNonNull indicates a method postcondition. With `@EnsuresNonNull`, the successful return (i.e., a non-exceptional return) of the annotated method results in the given expressions being non-null. See the Javadoc for examples of its use.

@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 3.5 shows their type hierarchy. For details, see Section 3.8.

3.2.4 Map key qualifiers

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

This annotation is checked by a Map Key Checker (Chapter 4, page 54) 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`.

The command-line argument `-AassumeKeyFor` makes the Nullness Checker not run the Map Key Checker. The Nullness Checker will unsoundly assume that the argument to `Map.get` is a key for the receiver map. That is, the second condition above is always considered to be `true`.

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 33.4). For example, enum types are implicitly non-null, so you never need to write `@NonNull MyEnumType`.

If you want details 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. All types default to `@NonNull`, except that `@Nullable` is used for casts, locals, instanceof, and implicit bounds (see Section 33.5.3). 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 dereference 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 33.8. More examples of the use of these annotations appear in the Javadoc for `@EnsuresNonNullIf`.

Java programs sometimes contain more complex nullness invariants. When these invariants are more complex than handled by the Nullness Checker, you will need to suppress a warning (see Section 3.4).

3.3.4 Nullness and array initialization

Suppose that you declare an array to contain non-null elements:

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

(recall that `Object` means the same thing as `@NonNull Object`). By default, the Nullness Checker unsoundly permits this code.

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

With the option enabled, you can write your code to create a nullable or `@MonotonicNonNull` 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 Nullness and conversions from collections to arrays

The nullness semantics of `Collection.toArray(T[])` cannot be captured by just the nullness type system, though the Nullness Checker contains special-case code to type-check calls to `toArray`. Therefore, you will probably have to write `@SuppressWarnings("nullness")` on any overriding definitions of `toArray`.

The nullness type of the returned array depends on the *size* of the passed parameter. In particular, the returned array's component type is `@NonNull` if the following conditions hold:

- The receiver collection's type argument (that is, the element type) is `@NonNull`, and
- The passed array size is less than or equal to the collection size. The Nullness Checker uses these heuristics to handle the most common cases:
  - the argument has length 0:
    - an empty array initializer, e.g., `c.toArray(new String[] {})`, or
    - array creation tree of size 0, e.g., `c.toArray(new String[0])`.
  - array creation tree with a collection `size()` method invocation as argument `c.toArray(new String[c.size()])`.

Additionally, when you supply the `-Alint=trustArrayLenZero` command-line option, a call to `Collection.toArray` will be estimated to return an array with a non-null component type if the argument is a field access where the field declaration has a `@ArrayLen(0)` annotation. This trusts the `@ArrayLen(0)` annotation, but does not verify it. Run the Constant Value Checker (see Chapter 24, page 161) to verify that annotation.

Note: The nullness of the returned array doesn't depend on the nullness of the passed array. This is a fact about `Collection.toArray(T[])`, not a limitation of this heuristic.

3.3.6 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 33.7.

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 33.7), which infers more specific types 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 33.5.3) or the NonNull default, then use the AnnotateNullable tool of the Daikon invariant detector.
- Inference of `@NonNull`: If your code uses the Nullable default (this is unusual), use one of these tools:
  - 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 34). 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 string, `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, expressions, static initializers, etc. Here are three ways to suppress a warning in such cases:

- Create a local variable to hold a subexpression, and suppress a warning on the local variable declaration.
- Use the `@AssumeAssertion` string in an `assert` message (see Section 34.2).
- Write a call to the `NullnessUtil.castNonNull` method.

The rest of this section discusses the `castNonNull` method. It is useful if you wish to suppress a warning within an expression.

The Nullness Checker considers both the return value and 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 recommended, 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 34.2.1.

To use the `castNonNull` method, the `checker-util.jar` file must be on the classpath at run time.

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 34.2.1 for a discussion of the distinction between warning suppression and defensive programming.

3.4.2 Null arguments to collection classes

For collection methods with `Object` formal parameter type, such as `contains`, `indexOf`, and `remove`, the annotated JDK forbids null as an argument.

The reason is that some implementations (like `ConcurrentHashMap`) throw `NullPointerException` if `null` is passed. It would be unsound to permit `null`, because it could lead to a false negative: the Checker Framework issuing no warning even though a `NullPointerException` can occur at run time.

However, many other common implementations permit such calls, so some users may wish to sacrifice soundness for a reduced number of false positive warnings. To permit `null` as an argument to these methods, pass the command-line argument `-Astubs=collection-object-parameters-may-be-null.astub`.

3.4.3 Conservative nullness annotations on the JDK

The JDK contains nullness annotations that preserve the Nullness Checker's guarantee (see Section 3.1) that your program will not crash with a `NullPointerException`. A formal parameter may be null in some circumstances, but must be non-null in other circumstances, and those circumstances are not expressible using the Nullness Checker's annotations.

An example is restrictions on collection arguments (see Section 3.4.2).

Another example is this `WeakReference` constructor:

```
  ...
  * @param q the queue with which the reference is to be registered,
  *          or {@code null} if registration is not required
  */
 public WeakReference(@Nullable T referent, ReferenceQueue<? super T> q) {
   ...
```

For some calls, `q` must be non-null. Therefore, `q` is annotated as `@NonNull` (which is the default and need not be explicitly written).

These JDK annotations reflect a verification philosophy: a verification tool finds all possible errors, but it sometimes issues a false positive warning. An alternate philosophy is a bug-finding philosophy: permit all calls that *might* be correct at run time, but sometimes miss a real error. If you wish to use the Checker Framework with the bug-finding philosophy (though the Checker Framework is still much more thorough than other bug-finders), you can do so by passing the command-line argument `-Astubs=sometimes-nullable.astub`.

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, see Section 39.6 for details):

```
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 Java projects in the plume-lib GitHub organization. Type-checking occurs on each build.
- The Daikon invariant detector. Run the command `make check-nullness`.

3.5.3 Publications

The papers "Practical pluggable types for Java" [PAC+08] (ISSTA 2008, `https://homes.cs.washington.edu/~mernst/pubs/pluggable-checkers-issta2008.pdf`) and "Building and using pluggable type-checkers" [DDE+11] (ICSE 2011, `https://homes.cs.washington.edu/~mernst/pubs/pluggable-checkers-icse2011.pdf`) describe case studies in which the Nullness Checker found previously-unknown errors in real software.

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. Also see the Checker Framework tutorial (`https://checkerframework.org/tutorial/`), which includes an example of using the Nullness Checker.

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 rarely need to annotate local variables (except their type arguments or array element types).

Only after this step should you run 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 34.2):

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

- To indicate that a routine returns the same value every time it is called, use `@Pure` (see Section 33.7.5).
- 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 Other tools for nullness checking

The Checker Framework's nullness annotations are similar to annotations used in 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. Most of the other tools are 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 40.10.1 for a comparison to other tools.

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 Figures 3.2, 3.3, and 3.4. If the other annotation is a declaration annotation, it may be moved; see Section 40.6.10.

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 SpotBugs's non-standard meaning for `@Nullable` (Section 3.7.2).

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`.

3.7.1 Which tool is right for you?

Different tools are appropriate in different circumstances. Section 40.10.1 compares verification tools like the Checker Framework with bug detectors like SpotBugs and Error Prone. In brief, a bug detector is easier to use because it requires fewer annotations, but it misses lots of real bugs that a verifier finds. You should use whichever tool is appropriate for the importance of your code.

You may also choose to use multiple tools, especially since each tool focuses on different types of errors. If you know that you will eventually want to do verification for some particular task (say, nullness checking), there is little point using the nullness analysis of a bug detector such as SpotBugs first. It is easier to go straight to using the Checker Framework.

If some other tool discovers a nullness error that the Checker Framework does not, please report it to us (see Section 41.2) so that we can enhance the Checker Framework. For example, SpotBugs might detect an error that the Nullness Checker does not, if you are using an unannotated library (including an unannotated part of the JDK) and running the Checker Framework in an unsound mode (see Section 2.2.2).

3.7.2 Incompatibility note about FindBugs and SpotBugs `@Nullable`

FindBugs and SpotBugs have a non-standard definition of `@Nullable`. This treatment is not documented in the annotation's 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, SpotBugs's `@Nullable` is detrimental rather than useful as documentation. In practice, your best bet is to not rely on SpotBugs for nullness analysis, even if you find SpotBugs useful for other purposes.

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

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

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

android.annotation.NonNull		
android.support.annotation.NonNull		
android.support.annotation.RecentlyNonNull		
androidx.annotation.NonNull		
androidx.annotation.RecentlyNonNull		
com.android.annotations.NonNull		
com.google.firebase.database.annotations.NotNull		
com.google.firebase.internal.NonNull		
com.mongodb.lang.NonNull		
com.sun.istack.NotNull		
com.sun.istack.internal.NotNull		
com.unboundid.util.NotNull		
edu.umd.cs.findbugs.annotations.NonNull		
io.micrometer.core.lang.NonNull		
io.micronaut.core.annotation.NonNull		
io.reactivex.annotations.NonNull		
io.reactivex.rxjava3.annotations.NonNull		
jakarta.annotation.Nonnull		
javax.annotation.Nonnull		
javax.validation.constraints.NotNull	$\Rightarrow$	org.checkerframework.checker.nullness.qual.NonNull
libcore.util.NonNull		
lombok.NonNull		
net.bytebuddy.agent.utility.nullability.NeverNull		
net.bytebuddy.utility.nullability.NeverNull		
org.antlr.v4.runtime.misc.NotNull		
org.checkerframework.checker.nullness.compatqual.NonNullDecl		
org.checkerframework.checker.nullness.compatqual.NonNullType		
org.chromium.build.annotations.OptimizeAsNonNull		
org.codehaus.commons.nullanalysis.NotNull		
org.eclipse.jdt.annotation.NonNull		
org.eclipse.jgit.annotations.NonNull		
org.eclipse.lsp4j.jsonrpc.validation.NonNull		
org.jetbrains.annotations.NotNull		
org.jmlspecs.annotation.NonNull		
org.jspecify.annotations.NonNull		
org.jspecify.nullness.NonNull		
org.netbeans.api.annotations.common.NonNull		
org.springframework.lang.NonNull		
reactor.util.annotation.NonNull		

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

android.annotation.Nullable	
android.support.annotation.Nullable	
android.support.annotation.RecentlyNullable	
androidx.annotation.Nullable	
androidx.annotation.RecentlyNullable	
com.android.annotations.Nullable	
com.beust.jcommander.internal.Nullable	
com.google.api.server.spi.config.Nullable	
com.google.firebase.database.annotations.Nullable	
com.google.firebase.internal.Nullable	
com.google.gerrit.common.Nullable	
com.google.protobuf.Internal.ProtoMethodAcceptsNullParameter	
com.google.protobuf.Internal.ProtoMethodMayReturnNull	
com.mongodb.lang.Nullable	
com.sun.istack.Nullable	
com.sun.istack.internal.Nullable	
com.unboundid.util.Nullable	
edu.umd.cs.findbugs.annotations.CheckForNull	
edu.umd.cs.findbugs.annotations.Nullable	
edu.umd.cs.findbugs.annotations.PossiblyNull	
edu.umd.cs.findbugs.annotations.UnknownNullness	
io.micrometer.core.lang.Nullable	
io.micronaut.core.annotation.Nullable	
io.reactivex.annotations.Nullable	
io.reactivex.rxjava3.annotations.Nullable	
io.vertx.codegen.annotations.Nullable	
jakarta.annotation.Nullable	
javax.annotation.CheckForNull	
javax.annotation.Nullable	⇒ org.checkerframework.checker.nullness.qual.Nullable
junitparams.converters.Nullable	
libcore.util.Nullable	
net.bytebuddy.agent.utility.nullability.AlwaysNull	
net.bytebuddy.agent.utility.nullability.MaybeNull	
net.bytebuddy.agent.utility.nullability.UnknownNull	
net.bytebuddy.utility.nullability.AlwaysNull	
net.bytebuddy.utility.nullability.MaybeNull	
net.bytebuddy.utility.nullability.UnknownNull	
org.apache.avro.reflect.Nullable	
org.apache.cxf.jaxrs.ext.Nullable	
org.apache.shindig.common.Nullable	
org.checkerframework.checker.nullness.compatqual.NullableDecl	
org.checkerframework.checker.nullness.compatqual.NullableType	
org.chromium.build.annotations.Nullable	
org.codehaus.commons.nullanalysis.Nullable	
org.eclipse.jdt.annotation.Nullable	
org.eclipse.jgit.annotations.Nullable	
org.jetbrains.annotations.Nullable	
org.jetbrains.annotations.UnknownNullability	
org.jmlspecs.annotation.Nullable	
org.jspecify.annotations.Nullable	
org.jspecify.nullness.Nullable	
org.jspecify.nullness.NullnessUnspecified	
org.netbeans.api.annotations.common.CheckForNull	
org.netbeans.api.annotations.common.NullAllowed	
org.netbeans.api.annotations.common.NullUnknown	
org.springframework.lang.Nullable	
reactor.util.annotation.Nullable	

Figure 3.3: Correspondence between other nullness annotations and the Checker Framework's `Nullable` annotation.

com.google.protobuf.Internal.ProtoPassThroughNullness	⇒ org.checkerframework.checker.nullness.qual.PolyNull

Figure 3.4: Correspondence between other nullness annotations and the Checker Framework's `PolyNull` annotation.

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

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

With SpotBugs, 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 SpotBugs supports only one and gives the programmer less flexibility.

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

FindBugs and SpotBugs made a poor choice of names. The choice of names should have made 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 with other tools, and intuitively clear to programmers. The FindBugs/SpotBugs choices make the SpotBugs annotations less helpful to people, and much less useful for other tools.

Another problem is that the SpotBugs `@Nullable` annotation is a declaration annotation rather than a type annotation. This means that it cannot be written in important locations such as type arguments, and it is misleading when written on a field of array type or a method that returns an array.

Overall, it is best to stay away from the SpotBugs nullness annotations and analysis, and use a tool with a more principled design.

3.7.3 Relationship to `Optional<T>`

Many null pointer exceptions occur because the programmer forgets to check whether a reference is null before dereferencing it. Java 8's `Optional<T>` class provides a partial solution: a programmer must call the `get` method to access the value, and the designers of `Optional` hope that the syntactic occurrence of the `get` method will remind programmers to first check that the value is present. This is still easy to forget, however.

The Checker Framework contains an Optional Checker (see Chapter 5) that guarantees that programmers use `Optional` correctly, such as calling `isPresent` before calling `get`.

There are some limitations to the utility of `Optional`, which might lead you to choose to use regular Java references rather than `Optional`. (For more details, see the article "Nothing is better than the `Optional` type".)

- It is still possible to call `get` on a non-present `Optional`, leading to a `NoSuchElementException`. In other words, `Optional` doesn't solve the underlying problem — it merely converts a `NullPointerException` into a `NoSuchElementException`, and in either case your code crashes.
- `NullPointerException` is still possible in code that uses `Optional`.
- `Optional` adds syntactic complexity, making your code longer and harder to read.
- `Optional` adds time and space overhead.
- `Optional` does not address important sources of null pointer exceptions, such as partially-initialized objects and calls to `Map.get`.

The Nullness Checker does not suffer these limitations. Furthermore, 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.

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. To use the `Opt` class, the `checker-util.jar` file must be on the classpath at run time.

3.8 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`. (The Initialization Checker focuses on `@NonNull` fields, to detect null pointer exceptions when using them. It does not currently ensure that primitives or `@Nullable` fields are initialized. Use the Initialized Fields Checker (Chapter 27, page 173) to check initialization with respect to properties other than nullness.)

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, see Section 3.8.8.

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) {
    // 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, int y) {
    // 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();
    f = new Object();
  }
  public MyClass(int x, int y, int z) {
    m();
    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();
  }
}
```

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.

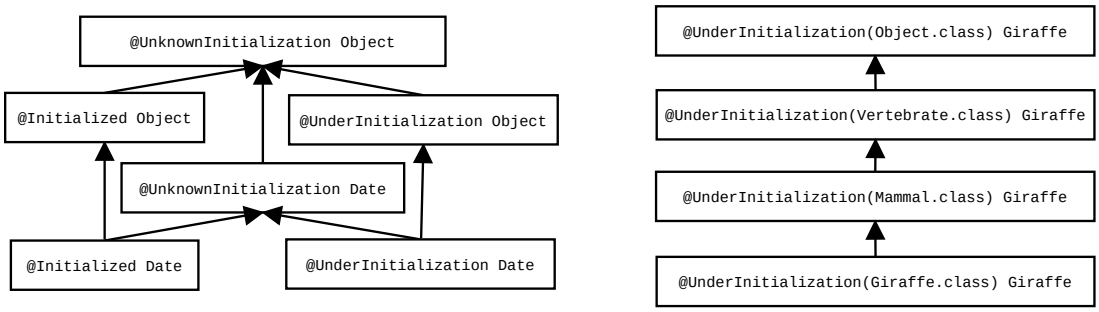


Figure 3.5: Partial type hierarchy for the Initialization type system. `@UnknownInitialization` and `@UnderInitialization` each take an optional parameter indicating how far initialization has proceeded, and the right side of the figure illustrates their type hierarchy in more detail.

The Initialization 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.

3.8.1 Initialization qualifiers

The initialization hierarchy is shown in Figure 3.5. 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 field 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 3.6 contains some examples.

3.8.2 How an object becomes initialized

Within the constructor, `this` starts out with `@UnderInitialization` type. As soon as all of the `@NonNull` fields in class *C* have been initialized, then `this` is treated as `@UnderInitialization(C)`. This means that `this` is still

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 3.6: 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.

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.

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.

3.8.3 `@UnderInitialization` examples

The most common use for the `@UnderInitialization` annotation is for a helper routine that is called by a 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 receiver's
  // class is a subclass of MyClass and not all of the subclass fields are initialized.
  private void checkRep(@UnderInitialization(MyClass.class) MyClass this) {
    ...
  }


}
```

At the end of the constructor, `this` is not fully initialized. Rather, it is `@UnderInitialization(`*`CurrentClass.class`*`)`. The reason is that there might be subclasses with uninitialized fields. The following example illustrates this:

```
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 were to type 'this' as @Initialized, then the following call is valid:
       doSomething();
   }
   void doSomething() {}
}

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

3.8.4 Partial initialization

So far, we have discussed initialization as if it is an all-or-nothing property: an object is uninitialized 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.
4. `B`'s constructor completes. The fields declared in `A` and `B` are initialized. However, the type system cannot assume that the object is fully initialized — 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 uninitialized classes. The `@UnderInitialization(`*`classliteral`*`)` annotation 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 subclasses of `OtherClass`.

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`

Note that neither `@UnderInitialization(A.class) A` nor `@UnderInitialization(A.class) B` may be used where `@Initialized A` is required. Permitting that would be unsound. For example, consider this code, where all types are `@NonNull` because `@NonNull` is the default annotation:

```
class A {
  Object aField;
  A() {
    aField = new Object();
  }
```

```
  Object get() {
    return aField;
  }
}
class B extends A {
  Object bField;
  B() {
    super();
    bField = new Object();
  }
  @Override
  Object get() {
    return bField;
  }
}
```

Given these declarations:

```
@Initialized A w;
@Initialized B x;
@UnderInitialization(A.class) A y;
@UnderInitialization(A.class) B z;
```

the expressions `w.get()` and `x.get()` evaluate to a non-null value, but `y.get()` and `z.get()` might evaluate to `null`. (`y.get()` might evaluate to `null` because the run-time type of `y` might be `B`. That is, `y` and `z` might refer to the same value, just as `w` and `x` might refer to the same value.)

3.8.5 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] 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 3.7: 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.process(arg);
    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.

3.8.6 Initialization of circular data structures

There is one final aspect of the initialization type system to be considered: the rules governing reading from and writing to objects that are currently under initialization (both reading from fields of objects under initialization and writing objects under initialization to fields). By default, only fully-initialized objects can be stored in a field of another object. If this were 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 from 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 3.7. Essentially, nothing is known about the initialization status of the value returned unless the receiver was `@Initialized`.

Similarly, Figure 3.8 shows under which conditions an assignment `x.f = y` is allowed for a `@NotOnlyInitialized` field `f`. If the receiver `x` is `@UnderInitialization`, then `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 3.8: 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<>(this);
  }

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

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

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

  @NotOnlyInitialized
  Node<T> next;

  @NotOnlyInitialized
  List<T> parent;

  @Nullable
  T data;

  // for sentinel construction
  Node(@UnderInitialization List<T> 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<>(this, this.next, data);
      this.next.prev = n;
      this.next = n;
    }
  }
```

3.8.7 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:

```
MyFile.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 40.6.1) 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 `@Initialized` and `@UnderInitialization`.

3.8.8 Suppressing warnings

To suppress most warnings related to partially-initialized values, use the warning suppression string "initialization". You can write `@SuppressWarnings("initialization")` on a field, constructor, or class, or pass the command-line argument `-AsuppressWarnings=initialization` when running the Nullness Checker. (For more about suppressing warnings, see Chapter 34, page 222.) 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.

This suppresses warnings that are specific to the Initialization Checker, but does not suppress warnings issued by the Checker Framework itself, such as about assignments or method overriding.

It is not possible to completely disable initialization checking while retaining nullness 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.

3.8.9 More details about initialization checking

Use of method annotations A method with an uninitialized 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 based on "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). The benefit (in terms of reduced false positive initialization warnings) from supporting partial initialization is greater than the benefit from adopting the Freedom Before Commitment system.

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 31) to treat calls to `Map.get` more precisely by refining its result to `@NonNull` in some cases.

4.1 Invoking the Map Key Checker

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

You can unsoundly suppress warnings related to map keys with `@SuppressWarnings("keyfor")`, or everywhere by using command-line option `-AsuppressWarnings=keyfor`; see Chapter 34, page 222.

The command-line argument `-AassumeKeyFor` makes the Map Key Checker unsoundly assume that the argument to `Map.get` is a key for the receiver map. This is like declaring the `Map.get` method as `V get(Object key)` rather than `@Nullable V get(Object key)`. (Just changing the JDK declaration would not work, however, because the Map Key Checker has special-case logic for `Map.get`. This is different than suppressing warnings, because it changes a method's return type. This is not the same as assuming that the return value is `@NonNull`, because the map's value type might be `@Nullable`, as in `Map<String, @Nullable Integer>`.)

4.2 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. There are no values of this type (not even `null`).

The following method annotations can be used to establish a method postcondition that ensures that a certain expression is a key for a map:

`@EnsuresKeyFor(String[] value, String[] map)` When the method with this annotation returns, the expression (or all the expressions) given in the `value` element is a key for the given maps. More precisely, the expression has the `@KeyFor` qualifier with the `value` arguments taken from the `map` element of this annotation.

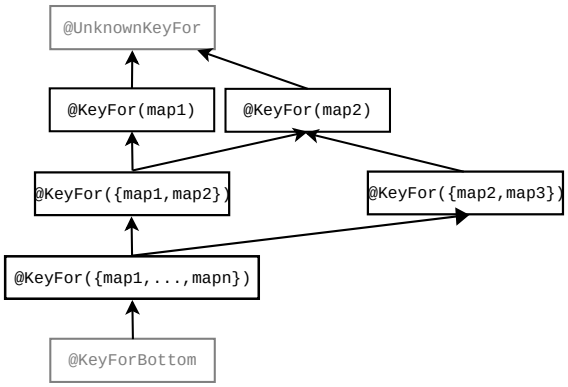


Figure 4.1: The subtyping relationship of the Map Key Checker's qualifiers. `@KeyFor(A)` is a supertype of `@KeyFor(B)` if and only if `A` is a subset of `B`. Qualifiers in gray are used internally by the type system but should never be written by a programmer.

`@EnsuresKeyForIf(String[] expression, boolean result, String[] map)` If the method with this annotation returns the given boolean value, then the given expression (or all the given expressions) is a key for the given maps.

4.3 Default annotations

The qualifier for the type of the `null` literal is `@UnknownKeyFor`. If `null` were `@KeyForBottom`, that would mean that `null` is guaranteed to be a key for every map (which is not necessarily true).

4.3.1 Default for lower bounds

Lower bounds are defaulted to `@UnknownKeyFor`. However, in `java.*` packages, the default for lower bounds is `@KeyForBottom`.

It is challenging to choose a default for lower bounds of type variables and wildcards.

Here is a comparison of two choices for lower bounds:

`@KeyForBottom` default	`@UnknownKeyFor` default (current choice)
`class MyClass1<@UnknownKeyFor T> {` `  T var = null; // OK`	`class MyClass1<T> {` `  T var = null; // OK`
`class MyClass2<T> {` `  @UnknownKeyFor T var = null; // OK`	
`class MyClass3<T> {` `  T var = null; // ERROR`	
	`class MySet1<T> implements Set<T> { }` `MySet1<@KeyFor("m") String> s1; // ERROR`

`class Set<E> { }`
`class MySet2<T> implements Set<T> { }`
`MySet2<@KeyFor("m") String> s2; // OK`
	`class Set<@KeyForBottom E> { }`
	`class MySet2<@KeyForBottom T> implements Set<T> { }`
	`MySet2<@KeyFor("m") String> s2; // OK`

If lower bounds are defaulted to `@KeyForBottom` (which is not currently the case), then whenever `null` is assigned to a variable whose type is a type variable, programmers must write an `@UnknownKeyFor` annotation on either the type variable declaration or the variable declarations, as shown in `MyClass1` and `MyClass2`. A disadvantage of this default is that the Map Key Checker may issue warnings in code that has nothing to do with map keys, and in which no map key annotations are used.

If lower bounds are defaulted to `@UnknownKeyFor` (which is currently the case), then whenever a client might use a `@KeyFor` type argument, programmers must write a `@KeyForBottom` annotation on the type parameter, as in `MySet2` (and `Set`).

4.3.2 Diagnosing the need for explicit @KeyFor on lower bounds

Under the current defaulting (lower bounds default to `@UnknownKeyFor`), suppose you write this code:

```
public class Graph<N> {
  Map<N, Integer> nodes = new HashMap<>();
}

class Client {
  @Nullable Graph<@KeyFor("g.nodes") String> g;
}
```

The Nullness Checker issues this error message:

```
Graph.java:14: error: [type.argument] incompatible types in type argument.
  @Nullable Graph<@KeyFor("g.nodes") String> g;
                  ^
  found   : @KeyFor("this.g.nodes") String
  required: [extends @UnknownKeyFor Object super @UnknownKeyFor null]
```

Note that the upper and lower bounds are both `@UnknownKeyFor`. You can make the code type-check by writing a lower bound, which is written before the type variable name (Section 32.1.2):

```
public class Graph<@KeyForBottom N> {
...
```

4.4 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
```

4.5 Local inference of @KeyFor annotations

Within a method body, you usually do not have to write `@KeyFor` explicitly (except sometimes on type arguments), 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 33.7.

Suppose we have these declarations:

```
Map<String,Date> m = new HashMap<>();
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

Use of the Optional Checker guarantees that your program will not suffer a `NoSuchElementException` when calling methods on an expression of `Optional` type. The Optional Checker also enforces Stuart Marks's style guidelines (see below).

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. Most notably, the Optional Checker guarantees that your code will not suffer a `NoSuchElementException` due to use of an empty `Optional`.

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 31). 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.)

5.1 How to run the Optional Checker

The standard way to run the Optional Checker is one of these command lines:

```
javac -processor optional MyFile.java ...
javac -processor org.checkerframework.checker.optional.OptionalChecker MyFile.java ...
```

5.2 Optional annotations

These qualifiers make up the Optional type system:

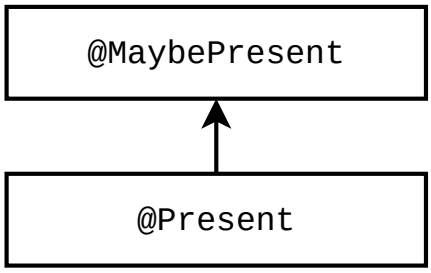


Figure 5.1: The subtyping relationship of the Optional Checker's qualifiers.

`@MaybePresent` The annotated `Optional` container may or may not contain a value. This is the default type, so programmers do not have to write it.

`@Present` The annotated `Optional` container definitely contains a (non-null) value.

`@PolyPresent` indicates qualifier polymorphism. For a description of qualifier polymorphism, see Section 32.2.

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

5.2.1 Optional method annotations

The Optional 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.

`@RequiresPresent` indicates a method precondition. The annotated method expects the specified expression to be a present Optional when this method is invoked. `@RequiresPresent` is a useful annotation for a method that requires a `@MaybePresent` field to be `@Present`.

`@EnsuresPresent` indicates a method postcondition. The successful return (i.e., a non-exceptional return) of the annotated method results in the given Optional expression being present. See the Javadoc for examples of its use.

`@EnsuresPresentIf` indicates a method postcondition. With `@EnsuresPresent`, the given Optional expression is present after the method returns. With `@EnsuresPresentIf`, if the annotated method returns the given boolean value (true or false), then the given Optional expression is present. See the Javadoc for examples of its use.

5.3 What the Optional Checker guarantees

The Optional Checker guarantees that your code will not throw a `NoSuchElementException` due to use of an absent `Optional` where a present `Optional` is needed. More specifically, the Optional Checker will issue an error if you call `get` or `orElseThrow()` on a `@MaybePresent Optional` receiver, because the receiver may be absent and each of these methods throws a `NoSuchElementException` if the receiver is empty at run time.

By contrast, the Optional Checker does not issue an error if you call `orElseThrow(Supplier)` with a possibly-absent `Optional`. That method call does not throw `NoSuchElementException`. The Optional Checker assumes that the programmer has mechanisms in place to handle whatever exception it throws. If you wish for the Optional Checker to warn about calling `orElseThrow(Supplier)` on a possibly-absent `Optional`, then you can use a stub file (Section 36.5, page 243) to annotate its receiver as `@Present`.

The Optional Checker does not check nullness properties, such as requiring that the argument to `of` is non-null or guaranteeing that the result of `get` is non-null. To obtain such a guarantee, run both the Optional Checker and the Nullness Checker (Chapter 3, page 31).

As with any checker, the guarantee is subject to certain limitations (see Section 2.3).

5.4 Suppressing optional warnings

It is often best to change the code or annotations when the Optional Checker reports a warning. Alternatively, you might choose to suppress the warning. This does not change the code but prevents the warning from being presented to you.

The Checker Framework supplies several ways to suppress warnings. The `@SuppressWarnings("optional")` annotation is specific to warnings raised by the Optional Checker. See Chapter 34 for additional usages. An example use is

```
// might return a possibly-empty Optional
Optional<T> wrapWithOptional(...) { ... }

void myMethod() {
  @SuppressWarnings("optional") // with argument x, wrapWithOptional always returns a present Optional
  @Present Optional<T> optX = wrapWithOptional(x);
}
```

The Optional Checker also permits the use of method calls and assertions to suppress warnings; see immediately below.

5.4.1 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, expressions, static initializers, etc. Here are three ways to suppress a warning in such cases:

- Create a local variable to hold a subexpression, and suppress a warning on the local variable declaration.
- Use the `@AssumeAssertion` string in an `assert` message (see Section 34.2).
- Write a call to the `OptionalUtil.castPresent` method.

The rest of this section discusses the `castPresent` method. It is useful if you wish to suppress a warning within an expression.

The Optional Checker considers both the return value and the argument to be a present Optional after the `castPresent` method call. The Optional Checker issues no warnings in any of the following code:

```
// One way to use castPresent as a cast:
@Present Optional<String> optString = castPresent(possiblyEmpty1);

// Another way to use castPresent as a cast:
castPresent(possiblyEmpty2).toString();

// It is possible, but not recommended, to use castPresent as a statement:
// (It would be better to write an assert statement with @AssumeAssertion
// in its message, instead.)
castPresent(possiblyEmpty3);
possiblyEmpty3.toString();
```

The `castPresent` method throws `AssertionError` if Java assertions are enabled and the argument is an empty `Optional`. However, it is not intended for general defensive programming; see Section 34.2.1.

To use the `castPresent` method, the `checker-util.jar` file must be on the classpath at run time.

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 problems in your code. First, it prevents using == on non-interned values, which can result in subtle bugs. For example:

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

Second, the Interning Checker 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`, and `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. (For some classes, every instance is interned; however, in other cases 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.
    - When searching for an element (say, in a collection), == may be appropriate.
    - Some routines return either their argument, or a modified version of it. Your code might compare `s == s.toLowerCase()` to see whether a string contains any upper-case characters.

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 6.5.

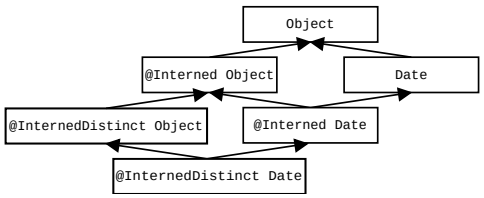


Figure 6.1: Type hierarchy for the Interning type system.

6.1 Interning annotations

6.1.1 Interning qualifiers

These qualifiers are part of the Interning type system:

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

`@InternedDistinct` indicates that the value is not `equals()` to any other Java value. This is a stronger (more restrictive) property than `@Interned`, but is a weaker property than writing `@Interned` on a class declaration. For details, see Section 6.3.3.

`@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. For a description of qualifier polymorphism, see Section 32.2.

6.1.2 Interning method and class annotations

`@UsesObjectEquals` is a class annotation (not a type annotation) that indicates that this class's `equals` method is the same as that of `Object`. 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.
Two ways to satisfy this annotation are: (1) neither this class nor any of its superclasses overrides the `equals` method, or (2) this class defines `equals` with body `return this == o;`.

`@InternMethod` is a method declaration annotation that indicates that this method returns an interned object and may be invoked on an uninterned object. See Section 6.3.1 for more details.

`@EqualsMethod` is a method declaration annotation that indicates that this method has a specification like `equals()`. The Interning Checker permits use of `this == arg` within the body.

`@CompareToMethod` is a method declaration annotation that indicates that this method has a specification like `compareTo()`. The Interning Checker permits use of `if (arg1 == arg2) { return 0; }` within the body.

`@FindDistinct` is a formal parameter declaration annotation that indicates that this method uses `==` to perform comparisons against the annotated formal parameter. A common reason is that the method searches for the formal parameter in some data structure, using `==`. Any value may be passed to the method.

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`.

6.3 Interned classes

An `@Interned` annotation on a class declaration indicates that all objects of that type are interned *except for newly created objects*. That means that all uses of such types are `@Interned` by default and the type `@UnknownInterned MyClass` is an invalid type.

An exception is *constructor results*. Constructor results and `this` within the body of the constructor are `@UnknownInterned` by default. Although `@UnknownInterned InternedClass` is not a legal type, no "type.invalid" error is issued at constructor declarations. Instead, an "interned.object.creation" error is issued at the invocation of the constructor. The user should inspect this location and suppress the warning if the newly created object is interned.

For example:

```
@Interned class InternedClass {
  @UnknownInterned InternedClass() {
    // error, "this" is @UnknownInterned.
    @Interned InternedClass that = this;
  }

  @SuppressWarnings("interning") // Only creation of an InternedClass object.
  static final InternedClass ONE = new InternedClass();
}
```

6.3.1 The intern() methods

Some interned classes use an `intern()` method to look up the interned version of the object. These methods must be annotated with the declaration annotation `@InternMethod`. This allows the checker to verify that a newly created object is immediately interned and therefore not issue an interned object creation error.

```
new InternedClass().intern() // no error
```

Because an `intern` method is expected to be called on uninterned objects, the type of `this` in `intern` is implicitly `@UnknownInterned`. This will cause an error if `this` is used someplace where an interned object is expected. Some of these warnings will be false positives that should be suppressed by the user.

```
@InternMethod
public InternedClass intern() {
  // Type of "this" inside an @InternMethod is @UnknownInterned
  @Interned InternedClass that = this; // error

  if (!pool.contains(this)) {
    @SuppressWarnings("interning:assignment")
    @Interned InternedClass internedThis = this;
    pool.add(internedThis);
  }
  return pool.get(this);
}
```

Some interned classes do not use an intern method to ensure that every object of that class is interned. For these classes, the user will have to manually inspect every constructor invocation and suppress the "interned.object.creation" error.

If the result of every invocation of a constructor is guaranteed to be interned, then the user should annotate the constructor result with `@Interned` and suppress a warning at the constructor.

```
@Interned class AnotherInternedClass {
  // manually verified that all constructor invocations are used such that all
  // new objects are interned
  @SuppressWarnings("super.invocation")
  @Interned AnotherInternedClass() {}
}
```

6.3.2 Default qualifiers and qualifiers for literals

The Interning Checker adds qualifiers to unannotated types, reducing the number of annotations that must appear in your code (see Section 33.4).

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

6.3.3 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` annotation is trusted (not verified), because it would be too complex to analyze the `equals()` method to ensure that no other value is `equals()` to an `@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 (whose return type is annotated with `@InternedDistinct`) returns the canonical version for certain values.
- The class is final, so that subclasses cannot violate these properties.

The result of a constructor call (that is, `new MyClass(...)`) is not necessarily `@InternedDistinct` because the `equals()` method of `MyClass` might return true for two distinct objects.

6.4 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 operand 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 Interning Checker's warnings look like

```
MyFile.java:716: error: [interning:not.interned] attempting to use a non-@Interned comparison operand
        if (date == idate)
            ^
```

To resolve a `not.interned` error, you should change the argument that is passed to `==`, or use `.equals()` instead of `==`, or suppress the warning.

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`.

To restrict which types the checker should type-check, pass a canonical name (fully-qualified name) 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.

6.4.1 Imprecision (false positive warnings) 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.)

6.5 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 (`https://plse.cs.washington.edu/daikon/`) is also annotated with `@Interned`. From directory `java/`, run `make check-interning`.

The paper "Building and using pluggable type-checkers" [DDE+11] (ICSE 2011, `https://homes.cs.washington.edu/~mernst/pubs/pluggable-checkers-icse2011.pdf`) describes case studies in which the Interning Checker found previously-unknown errors in real software.

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.

6.6 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 40.6.10.

Chapter 7

Called Methods Checker for the builder pattern and more

The Called Methods Checker tracks the names of methods that have definitely been called on an object. This checker is useful for checking any property of the form "call method A before method B". For the purpose of this checker, a method has "definitely been called" if it is *invoked*: a method that might never return or that might throw an exception has definitely been called on every path after the call, including exceptional paths. The checker also assumes that the program is free of null-pointer dereferences. You can verify that all pointer dereferences are safe by running the Nullness Checker (Chapter 3, page 31).

The Called Methods Checker provides built-in support for one such property: that clients of the builder pattern for object construction always provide all required arguments before calling `build()`. The builder pattern is a flexible and readable way to construct objects, but it is error-prone. Failing to provide a required argument causes a run-time error that manifests during testing or in the field, instead of at compile time as for regular Java constructors. The Called Methods Checker verifies at compile time that your code correctly uses the builder pattern, never omitting a required argument. The Called Methods Checker has built-in support for Lombok (see the caveats about Lombok in Section 7.2) and AutoValue.

You can verify other builders, or verify other properties of the form "`foo()` must be called before `bar()`", by writing method specifications. Section 7.5 describes another example related to a security property.

If the checker issues no warnings, then you have a guarantee that your code supplies all the required information to the builder. The checker might yield a false positive warning when your code is too tricky for it to verify. Please submit an issue if you discover this.

7.1 How to run the Called Methods Checker

```
javac -processor calledmethods MyFile.java ...
javac -processor org.checkerframework.checker.calledmethods.CalledMethodsChecker MyFile.java ...
```

The Called Methods Checker supports the following optional command-line arguments:

- The `-ACalledMethodsChecker_disableBuilderFrameworkSupports` option disables automatic annotation inference for builder frameworks. Section 7.4 describes its syntax. Supply this if you are uninterested in errors in the use of builders, but are using the Called Methods Checker to detect errors in other types of code.
- The `-ACalledMethodsChecker_disableReturnsReceiver` option disables the Returns Receiver Checker (Chapter 25, page 166), which ordinarily runs as a subchecker of the Called Methods Checker. If the code being checked does not use fluent APIs, then you can supply this option and the Called Methods Checker will run much faster. This option is the default when the Called Methods Checker is run as part of the Resource Leak Checker (Chapter 8, page 73).

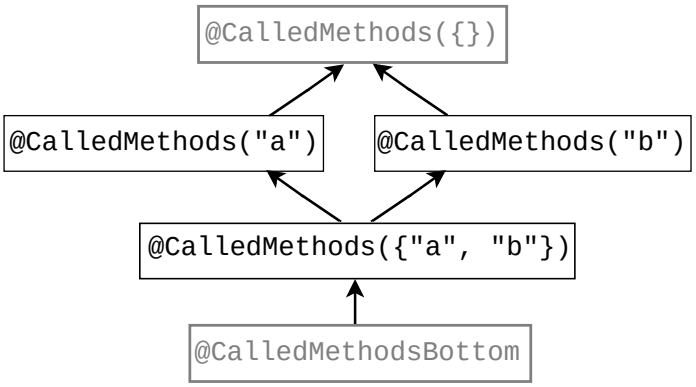


Figure 7.1: The type hierarchy for the Called Methods type system, for an object with two methods: `a()` and `b()`. Types displayed in gray should rarely be written by the programmer.

- The `-ACalledMethodsChecker_useValueChecker` option improves precision when analyzing code that uses the AWS SDK's `DescribeImageRequest` API. See Section 7.5.

7.2 For Lombok users

The Called Methods Checker supports projects that use Lombok via the io.freefair.lombok Gradle plugin automatically. However, note that the checker's error messages refer to Lombok's output, which is a variant of your source code that appears in a `delombok` directory. To fix issues, you should edit your original source code, **not** the files in the checker's error messages.

If you use Lombok with a build system other than Gradle, you must configure it to do two tasks. If either of these is not done, the checker will not issue any errors on Lombok code.

- set Lombok configuration option `lombok.addLombokGeneratedAnnotation = true`
- delombok the code before passing it to the checker

7.3 Specifying your code

The Called Methods Checker reads method specifications (contracts) that state what a method requires when it is called. It warns if method arguments do not satisfy the method's specification.

If you use AutoValue or Lombok, most specifications are automatically inferred by the Called Methods Checker, from field annotations such as `@Nullable` and field types such as `Optional`. Section 7.4 gives defaulting rules for Lombok and AutoValue.

In some cases, you may need to specify your code. You do so by writing one of the following type annotations (Figure 7.1):

`@CalledMethods(String[] methodNames)` The annotated type represents values on which all the given methods were definitely called. (Other methods might also have been called.) `@CalledMethods()`, with no arguments, is the default annotation.
Suppose that the method `build` is annotated as

```
class MyObjectBuilder {
  MyObject build(@CalledMethods({"setX", "setY"}) MyObjectBuilder this) { ... }
}
```

Then the receiver for any call to `build()` must have had `setX()` and `setY()` called on it.
A typical case in which Lombok users must manually write this annotation is when performing extra builder input validation. Performing validation can be done through subclassing generated builders and overriding the generated `build()` method as follows.

```
class MyObjectSubBuilder extends MyObjectBuilder {
  @Override
  MyObject build(@CalledMethods({"setX", "setY"}) MyObjectSubBuilder this) {
    MyObject o = super.build();
    if ((o.getX() == null) != (o.getY() == null)) {
      throw new IllegalArgumentException("Nullness for x and y should be equal");
    }
    return o;
  }
}
```

`@CalledMethodsPredicate(String expression)` The boolean expression specifies the required method calls. The string is a boolean expression composed of method names, disjunction (`||`), conjunction (`&&`), not (`!`), and parentheses.
For example, the annotation `@CalledMethodsPredicate("x && y || z")` on a type represents objects such that either both the `x()` and `y()` methods have been called on the object, **or** the `z()` method has been called on the object.
A note on the not operator (`!`): the annotation `@CalledMethodsPredicate("!m")` means "it is not true that `m` was definitely called"; equivalently "there is some path on which `m` was not called". The annotation `@CalledMethodsPredicate("!m")` does *not* mean "`m` was not called".
The Called Methods Checker does not have a way of expressing that a method must never be called. You can do unsound bug-finding for such a property by using the `!` operator. The Called Methods Checker will detect if the method was always called, but will silently approve the code if the method is called on some but not all paths.

`@This` `@This` may only be written on a method return type, and means that the method returns its receiver. This is helpful when type-checking fluent APIs. This annotation is defined by the Returns Receiver Checker (Chapter 25, page 166), but is particularly useful for the Called Methods Checker because many builders are fluent APIs.
For Lombok users it is important to (manually) add `@This` to any custom method that calls any other generated builder method. For instance, in the following example, it is important to annotate `setXMod42()` with `@This`, since the added method calls `setX()` (which returns the current builder instance).

```
class MyObjectBuilder {

  @This
  MyObjectBuilder setXMod42(int x) {
    return setX(x % 42);
  }
}
```

`@CalledMethodsBottom` The bottom type for the Called Methods hierarchy. Conceptually, this annotation means that all possible methods have been called on the object. Programmers should rarely, if ever, need to write this annotation — write an appropriate `@CalledMethods` annotation instead. The type of `null` is `@CalledMethodsBottom`.

There are also method annotations:

`@EnsuresCalledMethods` This declaration annotation specifies a post-condition on a method, indicating the methods it guarantees to be called. This annotation is repeatable, meaning that you can write multiple copies of it (with different arguments) on the same method, and the checker will check all of them.
For example, this specification:

```
@EnsuresCalledMethods(value = "#1", methods = {"x", "y"})
void m(Param p) { ... }
```

guarantees that `p.x()` and `p.y()` will always be called before `m` returns. The body of `m` must satisfy that property, and clients of `m` can depend on the property.
Sometimes, you need to provide information to enable the Called Methods Checker to verify the property. Consider this example:

```
@EnsuresCalledMethods(value="#1", methods="close")
public void closeSocket(Socket sock) throws IOException {
    sock.close();
    m();
}
```

If `m()` might have side-effects (i.e., it is not annotated as `@SideEffectFree`, as `@Pure`), then the Called Methods Checker issues an error because it cannot make any assumptions about the call to `m()`, and therefore assumes the worst: that all information it knows about in-scope variables (including that `close()` was called on `sock`) is stale and must be discarded. Here are possible fixes:

- add a `@SideEffectFree` or `@Pure` annotation to `m()`, if `m()` is in fact side-effect free or pure.
- re-order the calls to `sock.close()` and `m()` so that the call to `sock.close()` appears last in `closeSocket()`.

`@EnsuresCalledMethodsIf` This declaration annotation specifies a post-condition on a method, indicating the methods it guarantees to be called if it returns a given result.
For example, this specification:

```
@EnsuresCalledMethodsIf(expression = "#1", methods = {"x", "y"}, result=true)
boolean m(Param p) { ... }
```

guarantees that `p.x()` and `p.y()` will always be called if `m` returns `true`. The body of `m` must satisfy that property, and clients of `m` can depend on the property.

`@EnsuresCalledMethodsVarargs` This version of `@EnsuresCalledMethods` always applies to the varargs parameter of the annotated method. It has only one argument, which is the list of methods that are guaranteed to be called on the varargs parameter's elements before the method returns. This annotation currently cannot be verified, and an `ensuresvarargs.unverified` error is always issued when it is used. When annotating a method as `@EnsuresCalledMethodsVarargs`, you should verify that the named methods are actually called on every element of the varargs parameter via some other method (such as manual inspection) and then suppress the warning.

`@EnsuresCalledMethodsOnException` This declaration annotation specifies a post-condition on a method, indicating the methods it guarantees to call when it throws an exception. Like most post-condition annotations, the other Called Methods post-condition annotations (`@EnsuresCalledMethods`, `@EnsuresCalledMethodsIf`, and `@EnsuresCalledMethodsVarargs`) only specify behavior for normal returns. The annotation `@EnsuresCalledMethodsOnExcept` allows you to write stronger specifications indicating that a method *always* calls the given methods:

```
@EnsuresCalledMethods(value = "#1", methods = {"close"})
@EnsuresCalledMethodsOnException(value = "#1", methods = {"close"})
void closeIfNonNull(Closeable p) {
  if (p != null) {
    p.close();
  }
}
```

`@EnsuresCalledMethodsOnException` can also be used together with the Resource Leak Checker's `@Owning` annotation. See Section 8.4.1.

`@RequiresCalledMethods` This declaration annotation specifies a pre-condition on a method, indicating that the expressions in its `value` argument must have called-methods types that include all the methods named in its `methods` argument. If the expression is a parameter of the annotated method, you should use a `@CalledMethods` annotation on the parameter instead.

7.4 Default handling for Lombok and AutoValue

This section explains how the Called Methods Checker infers types for code that uses the Lombok and AutoValue frameworks. Most readers can skip these details.

You can disable support for builder frameworks by specifying them in a comma-separated lowercase list to the command-line flag `disableBuilderFrameworkSupports`. For example, to disable both Lombok and AutoValue support, use:

```
-ACalledMethodsChecker_disableBuilderFrameworkSupports=autovalue,lombok
```

The Called Methods Checker automatically assumes default annotations for code that uses builders generated by Lombok and AutoValue. There are three places where annotations are usually assumed:

- A `@CalledMethods` annotation is placed on the receiver of the `build()` method, indicating the setter methods that must be invoked on the builder before calling `build()`. For Lombok, this annotation's argument is the set of `@lombok.NonNull` fields that do not have default values. For AutoValue, it is the set of fields that are not `@Nullable`, `Optional`, or a Guava Immutable Collection.
- The return type of a `toBuilder()` method (for example, if the `toBuilder = true` option is passed to Lombok's `@Builder` annotation) is annotated with the same `@CalledMethods` annotation as the receiver of `build()`, using the same rules as above.
- A `@This` annotation is placed on the return type of each setter in the builder's implementation.

If your program directly defines any of these methods (for example, by adding your own setters to a Lombok builder), you may need to write the annotations manually.

Minor notes/caveats on these rules:

- Lombok fields annotated with `@Singular` will be treated as defaulted (i.e., not required), because Lombok will set them to empty collections if the corresponding setter is not called.
- If you manually provide defaults to a Lombok builder (for example, by defining the builder yourself and assigning a default value to the builder's field), the checker will treat that field as defaulted *most* of the time. In particular, it will not treat it as defaulted if it is defined in bytecode rather than in source code.

7.5 Using the Called Methods Checker for properties unrelated to builders

The Called Methods Checker can be used to verify any property of the form "always call A before B", even if the property is unrelated to builders.

For example, consider the AWS EC2 `describeImages` API, which clients use during the process of initializing a new cloud instance. CVE-2018-15869 describes how an improperly-configured request to this API can make the requesting client vulnerable to a "machine-image sniping" attack that would allow a malicious third party to control the operating system image used to initialize the machine. To prevent this attack, clients must specify some trusted source for the image by calling the `withOwners` or `withImageIds` methods on the request prior to sending it to AWS. Using a stub file for the `describeImages` API (DescribeImages.astub), the Called Methods Checker can prove that a client is not vulnerable to such an attack.

To improve precision, you can specify the `-ACalledMethodsChecker_useValueChecker` command-line option, which instructs the checker to treat provably-safe calls to the `withFilters` method of a `DescribeImagesRequest` as equivalent to the `withOwners` or `withImageIds` methods.

7.6 More information

The paper "Verifying Object Construction" [KRS+20] (ICSE 2020, `https://homes.cs.washington.edu/~mernst/pubs/object-construction-icse2020-abstract.html`) gives more information about the Called Methods Checker, such as theoretical underpinnings and results of experiments. (The paper uses an earlier name, "Object Construction Checker".)

Chapter 8

Resource Leak Checker for must-call obligations

The Resource Leak Checker guarantees that every resource is closed rather than leaked. Examples of resources are a channel, executor, `ExecutionControl`, file, `FileLock`, `Formatter`, reader, `Scanner`, socket, stream, writer, etc.

More precisely, the Resource Leak Checker guarantees that the program fulfills every object's must-call obligations before the object is de-allocated.

A resource leak occurs when a program does not explicitly dispose of some finite underlying resource, such as a socket, file descriptor, or database connection. To dispose of the resource, the program should call some method on an object. (De-allocating or garbage-collecting the object is not sufficient.) For example, the program must call `close()` on every object that implements the interface `java.io.Closeable`.

The Resource Leak Checker can check any property of the form "the programmer must call each method in a set of methods *M* at least once on object *O* before *O* is de-allocated". For resource leaks, by default *M* is the set containing `close()` and *O* is any object that implements the interface `java.io.Closeable`. You can extend this guarantee to other types and methods by writing `@MustCall` or `@InheritableMustCall` annotations, as described in Section 29.1.

The Resource Leak Checker works in three stages:

1. The Must Call Checker (Chapter 29, page 180) over-approximates each expression's must-call methods as a `@MustCall` type.
2. The Called Methods Checker (Chapter 7, page 68) under-approximates each expression's definitely-called methods as a `@CalledMethods` type.
3. When any program element goes out of scope (i.e., it is ready to be de-allocated), the Resource Leak Checker compares the types `@MustCall(`*MC*`)` and `@CalledMethods(`*CM*`)`. It reports an error if there exists some method in *MC* that is not in *CM*.

8.1 How to run the Resource Leak Checker

Run one of these lines:

```
javac -processor resourceleak MyFile.java ...
javac -processor org.checkerframework.checker.resourceleak.ResourceLeakChecker MyFile.java ...
```

The Resource Leak Checker supports all the command-line arguments listed in Section 7.1 for the Called Methods Checker, plus five others:

`-ApermitStaticOwning` See Section 8.4.2.
`-AresourceLeakIgnoredExceptions=...` See Section 8.7.
`-ApermitInitializationLeak` See Section 8.8.

-AenableReturnsReceiverForRlc Enables the Returns Receiver Checker (Chapter 25) when running this checker. By default, the Returns Receiver Checker is disabled when running the Called Methods Checker as part of the RLC, because it has a significant type-checking cost but provides little benefit in the common case, since resources are rarely combined with fluent builders.

-AenableWpiForRlc See Section 8.11.

If you are running the Resource Leak Checker, then there is no need to run the Must Call Checker (Chapter 29, page 180), because the Resource Leak Checker does so automatically.

8.2 Resource Leak Checker annotations

The Resource Leak Checker relies on the type qualifiers of two other checkers: the Must Call Checker (Section 29.1) and the Called Methods Checker (Section 7.3). You might need to write qualifiers from either type hierarchy. The most common annotations from these checkers that you might need to write are:

@MustCall(String[] value) for example on an element with compile-time type `Object` that might contain a `Socket`. See Section 29.1.

@InheritableMustCall(String[] value) on any classes defined in your program that have must-call obligations. See Section 29.2.

@EnsuresCalledMethods and/or @EnsuresCalledMethodsOnException on a method in your code that fulfills a must-call obligation of one of its parameters or of a field. See Section 7.3.

The Resource Leak Checker supports annotations that express aliasing patterns related to resource leaks:

@Owning

@NotOwning expresses ownership. When two aliases exist to the same Java object, `@Owning` and `@NotOwning` indicate which of the two is responsible for fulfilling must-call obligations. Constructor results are always `@Owning`. Method returns default to `@Owning`. Formal parameters and fields default to `@NotOwning`. For more details, see Section 8.4.

@MustCallAlias represents a "resource-aliasing" relationship. Resource aliases are distinct Java objects that control the same resource(s): fulfilling the must-call obligations of one object also fulfills the obligations of the other object. For details, see Section 8.5.

The Resource Leak Checker also supports an annotation to permit re-assigning fields or re-opening resources:

@CreatesMustCallFor(String value) is a declaration annotation that indicates that after a call to a method with this annotation, none of the must-call obligations of the in-scope, owning expression listed in `value` have been met. In other words, the annotated method "resets" the must-call obligations of the expression. Multiple `@CreatesMustCallFor` annotations can be written on the same method. Section 8.6 explains how this annotation permits re-assignment of owning fields or the re-opening of resources.

8.3 Example of how safe resource usage is verified

Consider the following example of safe use of a `Socket`, in which the comments indicate the inferred Must Call and Called Methods type qualifiers for `s`:

```
{
  Socket s = null;
  // 1. @MustCall({}) @CalledMethodsBottom
  try {
    s = new Socket(myHost, myPort);
    // 2. @MustCall("close") @CalledMethods({})
```

```
  } catch (Exception e) {
    // do nothing
  } finally {
    if (s != null) {
      s.close();
      // 3. @MustCall("close") @CalledMethods("close")
    } else {
      // do nothing
      // 4. @MustCall("close") @CalledMethodsBottom
    }
    // 5. @MustCall("close") @CalledMethods("close")
  }
  // 6. @MustCall("close") @CalledMethods("close")
}
```

At point 1, `s`'s type qualifiers are the type qualifiers of `null`: `null` has no must-call obligations (`@MustCall({})`), and methods cannot be called on it (`@CalledMethodsBottom`).

At point 2, `s` is a new `Socket` object, which has a must-call obligation (`@MustCall("close")`) and has had no methods called on it (`@CalledMethods({})`).

At point 3, `close()` has definitely been called on `s`, so `s`'s Called Methods type is updated. Note that the Must Call type does not change.

At point 4, `s` is definitely `null` and its type is adjusted accordingly.

At point 5, `s`'s type is the least upper bound of the types at points 3 and 4.

At point 6, `s` goes out of scope. The Resource Leak Checker reports a `required.method.not.called` error if the Must Call set contains any element that the Called Methods set does not.

8.4 Aliased references and ownership transfer

Resource leak checking is complicated by aliasing. Multiple expressions may evaluate to the same Java object, but each object only needs to be closed once. (Section 8.5 describes a related situation called "resource aliasing", when multiple Java objects refer to the same underlying resource.)

For example, consider the following code that safely closes a `Socket`:

```
void example(String myHost, int myPort) throws IOException {
  Socket s = new Socket(myHost, myPort);
  closeSocket(s);
}
void closeSocket(@Owning @MustCall("close") Socket t) {
  try {
    t.close();
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

There are two aliases for a socket object: `s` in `example()` and `t` in `closeSocket()`. Ordinarily, the Resource Leak Checker requires that `close()` is called on every expression of type `Socket`, but that is not necessary here. The Resource Leak Checker should not warn when `s` goes out of scope in `example()`, because `closeSocket()` takes ownership of the socket — that is, `closeSocket()` takes responsibility for closing it. The `@Owning` annotation on `t`'s declaration expresses this fact; it tells the Resource Leak Checker that `t` is the reference that must be closed, and its alias `s` need not be closed.

Constructor results are always `@Owning`. Method returns default to `@Owning`, and parameters and fields default to `@NotOwning`. This treatment of parameter and return types ensures sound handling of unannotated third-party libraries: any object returned from such a library will be tracked by default, and the checker never assumes that passing an object to an unannotated library will satisfy its obligations.

`@Owning` and `@NotOwning` always *transfer* must-call obligations: must-call obligations are conserved (i.e., neither created nor destroyed) by ownership annotations. Writing `@Owning` or `@NotOwning` can never make the checker unsound: a real warning can never be hidden by them. As with any annotation, incorrect or missing annotations can lead to false positive warnings.

8.4.1 Owning parameters and exceptions

When `@Owning` is written on a method parameter, the method only takes ownership of the parameter when it returns normally. In this example, the Resource Leak Checker will report an error in the `example` method and allow the definition of `closeSocket`:

```
void example(String myHost, int myPort) throws Exception {
  // Error: `s` is not closed on all paths
  Socket s = new Socket(myHost, myPort);

  // `closeSocket` does not have to close `s` when it throws IOException.
  // Instead, this method has to catch the exception and close `s`.
  closeSocket(s);
}

void closeSocket(@Owning Socket t) throws IOException {
  throw new IOException();
}
```

Sometimes a method really does promise to call some methods on an `@Owning` parameter, even if it throws an exception. The annotation `@EnsuresCalledMethodsOnException` can overcome this limitation. For example, a constructor that throws an exception might choose to close an `@Owning` parameter instead of letting ownership remain with the caller:

```
@EnsuresCalledMethodsOnException(value = "#1", methods = "close")
public Constructor(@Owning Closeable resource) {
  this.resource = resource;
  try {
    initialize();
  } catch (Exception e) {
    resource.close();
    throw e;
  }
}
```

8.4.2 Owning fields

Unannotated fields are treated as non-owning.

For **final, non-static owning fields**, the Resource Leak Checker enforces the "resource acquisition is initialization (RAII)" programming idiom. Some destructor-like method `d()` must satisfy the field's must-call obligation (and this fact must be expressed via an `@EnsuresCalledMethods` annotation on `d()`), and the enclosing class must have a `@MustCall("d")` obligation to ensure the destructor is called. In addition to the `@EnsuresCalledMethods` annotation, which guarantees that the field(s) it references have their must-call obligations satisfied on non-exceptional paths, the

Resource Leak Checker requires those fields to have their must-call obligations satisfied on all paths in (only) the destructor, and will issue a `contracts.exceptional.postcondition` error if they are not satisfied. Resolve this error by ensuring that the required methods are called on all exceptional paths.

Non-final, non-static owning fields usually require one or more `@CreatesMustCallFor` annotations when they might be re-assigned. See Section 8.6 for more details on how to annotate a non-final, non-static owning field.

Owning fields are treated slightly differently in constructors versus normal methods. In normal methods, assigning a value to an owning field always satisfies the object's must-call obligations. However, within a constructor, those obligations are only satisfied if the constructor returns normally. If the constructor throws an exception, the constructed object will not be accessible afterward, and therefore its fields need to be closed before it exits.

This constructor safely closes the object it allocates before throwing an exception:

```
private final @Owning Socket socket;

public ConstructorThatCanThrow() throws IOException {
  Socket s = new Socket(myHost, myPort);
  try {
    initialize(s); // may throw IOException
  } catch (Exception e) {
    s.close();
    throw e;
  }
  this.socket = s;
}
```

An assignment to a **static owning field** does not satisfy a must-call obligation; for example,

```
static @Owning PrintWriter debugLog = new PrintWriter("debug.log");
```

The Resource Leak Checker issues a warning about every assignment of an object with a must-call annotation into a static owning field, indicating that the obligation of the field's content might not be satisfied. When those fields are used throughout execution, until the program exits, there is no good place to dispose of them, so these warnings might not be useful. The `-ApermitStaticOwning` command-line argument suppresses warnings related to static owning fields. This can help in checking legacy code. It permits only a small number of resources to be retained throughout execution, related to the number of such fields and assignments to them.

8.5 Resource aliasing

A *resource alias* set is a set of Java objects that correspond to the same underlying system resource. Calling a must-call method on any member of a resource-alias set fulfills that obligation for all members of the set. Members of the set may have different Java types.

Programmers most often encounter resource aliasing when using *wrapper types*. For example, the Java `BufferedOutputStream` wrapper adds buffering to a delegate stream. The wrapper's `close()` method invokes `close()` on the delegate. Calling `close()` on either object has the same effect: it closes the underlying resource.

A resource aliasing relationship is expressed in source code via a pair of `@MustCallAlias` annotations: one on a parameter of a method or constructor, and another on its return type. For example, the annotated JDK contains this constructor of `BufferedOutputStream`:

```
@MustCallAlias BufferedOutputStream(@MustCallAlias OutputStream out);
```

When a pair of `@MustCallAlias` annotations is written on a method or constructor `m`'s return type and its parameter `p`, the Resource Leak Checker requires one of the following:

1. `p` is passed to another method or constructor (including `super`) in a `@MustCallAlias` position, and `m` returns that method's result, or
2. `p` is stored in the only `@Owning` field of the enclosing class (a class with more than one `@Owning` field cannot have a resource alias relationship).

8.5.1 A complete wrapper type example

Here is a complete example of a type `InputStreamWrapper` that wraps an `InputStream` as a resource alias. Defining a wrapper type typically involves combined usage of `@InheritableMustCall`, `@EnsuresCalledMethods`, an `@Owning` field, and `@MustCallAlias`. The `test` method shows that the checker is able to verify code that releases an `InputStream` using either the `InputStream` directly or a wrapping `InputStreamWrapper`.

```
@InheritableMustCall("dispose")
class InputStreamWrapper {
  private final @Owning InputStream stream;

  @MustCallAlias InputStreamWrapper(@MustCallAlias InputStream stream) {
    this.stream = stream;
  }

  @EnsuresCalledMethods(value = "this.stream", methods = "close")
  public void dispose() throws IOException {
    this.stream.close();
  }

  /** Shows that either the stream or the wrapper can be closed. */
  static void test(@Owning InputStream stream, boolean b) throws IOException {
    InputStreamWrapper wrapper = new InputStreamWrapper(stream);
    if (b) {
      stream.close();
    } else {
      wrapper.dispose();
    }
  }
}
```

8.6 Creating obligations (how to re-assign a non-final owning field)

Consider a class that has must-call obligations; that is, the class declaration is annotated with `@MustCall(...)`. Every constructor implicitly creates obligations for the newly-created object. Non-constructor methods may also create obligations when re-assigning non-final owning fields or allocating new system-level resources.

A post-condition annotation, `@CreatesMustCallFor`, indicates for which expression an obligation is created. If you write `@CreatesMustCallFor(`*T*`)` on a method *N* that overrides a method *M*, then *M* must also be annotated as `@CreatesMustCallFor(`*T*`)`. (*M* may also have other `@CreatesMustCallFor` annotations that *N* does not.)

`@CreatesMustCallFor` allows the Resource Leak Checker to verify uses of non-final fields that contain a resource, even if they are re-assigned. Consider the following example:

```
  @MustCall("close") // default qualifier for uses of SocketContainer
  class SocketContainer {
    private @Owning Socket sock;

    public SocketContainer() { sock = ...; }

    void close() { sock.close(); }

    @CreatesMustCallFor("this")
    void reconnect() {
```

```
      if (!sock.isClosed()) {
        sock.close();
      }
      sock = ...;
    }
  }
```

In the lifetime of a `SocketContainer` object, `sock` might be re-assigned arbitrarily many times: once at each call to `reconnect()`. This code is safe, however: `reconnect()` ensures that `sock` is closed before re-assigning it.

Sections 8.6.1 and 8.6.2 explain how the Resource Leak Checker verifies uses and declarations of methods annotated with `@CreatesMustCallFor`.

8.6.1 Requirements at a call site of a `@CreatesMustCallFor` method

At a call site to a method annotated as `@CreatesMustCallFor(`*expr*`)`, the Resource Leak Checker:

1. Treats any existing `@MustCall` obligations of *expr* as *satisfied*.
2. Creates a fresh obligation to check, as if *expr* were assigned to a newly-allocated object (i.e., as if *expr* were a constructor result).
3. Un-refines the type in the Called Methods Checker's type hierarchy for *expr* to `@CalledMethods({})`, if it had any other Called Methods type.
4. Requires that the expression corresponding to *expr* (that is, *expr* viewpoint-adapted to the method call site) is owned; that is, it is annotated or defaulted as `@Owning`. Otherwise, the checker will issue a `reset.not.owning` error at the call site. You can avoid this error by extracting *expr* into a new local variable (because locals are `@Owning` by default) and replacing all instances of *expr* in the call with references to the new local variable.

Treating the obligation before the call as satisfied is sound: the checker creates a new obligation for calls to `@CreatesMustCallFor` methods, and the Must Call Checker (Chapter 29, page 180) ensures the `@MustCall` type for the target expression will have a *superset* of any methods present before the call. Intuitively, calling a `@CreatesMustCallFor` method "resets" the obligations of the target expression, so whether they were satisfied before the call or not is irrelevant.

If a `@CreatesMustCallFor` method *n* is invoked within a method *m* that has a `@CreatesMustCallFor` annotation, and the `@CreatesMustCallFor` annotations on *n* and *m* have the same target expression — imposing the obligation produced by calling *n* on the caller of *m* — then the newly-created obligation is treated as satisfied immediately at the call site of *n* in the body of *m* (because it is imposed at call sites of *m* instead).

8.6.2 Requirements at a declaration of a `@CreatesMustCallFor` method

Any method that re-assigns a non-final, owning field of some object `obj` must be annotated `@CreatesMustCallFor("obj")`. Other methods may also be annotated with `@CreatesMustCallFor`.

The Resource Leak Checker enforces two rules to ensure that re-assignments to non-final, owning fields (like `sock` in method `reconnect` above) are sound:

- any method that re-assigns a non-final, owning field of an object must be annotated with a `@CreatesMustCallFor` annotation whose expression is a reference to that object.
- when a non-final, owning field *f* is re-assigned at statement *s*, at the program point before *s*, *f*'s must-call obligations must have been satisfied.

The first rule ensures that `close()` is called after the last call to `reconnect()`, and the second rule ensures that `reconnect()` safely closes `sock` before re-assigning it. Because the Called Methods Checker treats calls to a `@CreatesMustCallFor` method like `reconnect()` as if the call might cause arbitrary side effects, after such a call the only method known to have been definitely called is the `@CreatesMustCallFor` method: previous called methods (including `close()`) do not appear in the `@CalledMethods` type qualifier.

8.7 Ignored exception types

The Resource Leak Checker checks that an element's must-call obligations are fulfilled when that element may go out of scope: at the end of its lexical scope or when control may be transferred to the end of its lexical scope, such as via a `break` or `continue` statement or via throwing an exception. As an example of an exception, consider the following method:

```
void foo() {
  Socket s = ...;
  bar();
  s.close();
}
```

If `bar` is declared to throw an exception, the Resource Leak Checker warns that a `Socket` may be leaked. If `bar` throws an exception, the only reference to `s` is lost, which could lead to a resource leak.

The Resource Leak Checker ignores control flow due to some exceptions.

- By default, the Resource Leak Checker ignores run-time errors that can occur unpredictably at most points in the program. For example, the JVM can throw an `OutOfMemoryError` on any allocation. Similarly, `ClassCircularityError`, `ClassFormatError`, and `NoClassDefFoundError` may occur at any reference to a class. Such exceptions usually terminate the program, and in that case unclosed resources do not matter. Furthermore, any method can throw `RuntimeException`, and the Checker Framework pessimistically assumes one can be thrown at every call site. Accounting for such exceptions would lead to vast numbers of false positive warnings, so the Resource Leak Checker assumes they are never thrown. Strictly speaking, this is an unsoundness: it can lead to false negatives (missed resource leaks) if the programmer catches these exceptions, which is a discouraged practice.
- The Resource Leak Checker also ignores exception types that can be verified to never occur. In particular, the Resource Leak Checker ignores `NullPointerException`s (use the Nullness Checker, Chapter 3, page 31) and `ArrayIndexOutOfBoundsException`s and `NegativeArraySizeException`s (use the Index Checker, Chapter 11, page 98). Other exception types may be added to this list in the future. Please let us know if there is a type that you think should be ignored by filing an issue listing both the exception type and the verification tool.

The set of ignored exception types can be controlled with the option `-AresourceLeakIgnoredExceptions=...`. The option takes a comma-separated list of fully-qualified exception types. A type can be prefixed with `=` to ignore exactly that type and not its subclasses. For example, for a very pedantic set of ignored exceptions, use either of these, which have the same effect:

```
-AresourceLeakIgnoredExceptions="java.lang.Error,=java.lang.NullPointerException"
-AresourceLeakIgnoredExceptions="=java.lang.NullPointerException,java.lang.Error"
```

Both of these ignore `java.lang.Error` (and all its subclasses) as well as `java.lang.NullPointerException` (but not its subclasses).

The keyword `default` will expand to the default set of ignored exceptions. So, to add an additional exception to the set of ignored exceptions, use one of these:

```
-AresourceLeakIgnoredExceptions="default,package.MyCustomException"
-AresourceLeakIgnoredExceptions="default,=package.MyCustomException"
```

8.8 Errors about field initialization

The Resource Leak Checker warns about re-assignments to owning fields, because the value that was overwritten might not have had its obligations satisfied. Such a warning is not necessary on the first assignment to a field, since the field

had no content before the assignment. Sometimes, the Resource Leak Checker is unable to determine that an assignment is the first one, so it conservatively assumes the assignment is a re-assignment and issues an error.

One way to prevent this false positive warning is to declare the field as `final`.

Alternately, to suppress all warnings related to field assignments in the constructor and in initializer blocks, pass the `-ApermitInitializationLeak` command-line argument. This makes the checker unsound: the Resource Leak Checker will not warn if the constructor and initializers set a field more than once. The amount of leakage is limited to how many times the field is set.

8.9 Errors about unknown must-call obligations

The Resource Leak Checker issues a `required.method.not.known` error when a variable with the type `@MustCallUnknown` has a must-call obligation. `@MustCallUnknown` rarely occurs, but if you encounter this error, usually the right thing to do is to write an explicit `@MustCall` annotation on the indicated expression (e.g., as a cast), because the Must Call Checker will only use `@MustCallUnknown` as a default when encountering a language feature that it is unable to reason about.

8.10 Collections of resources

The Resource Leak Checker cannot verify code that stores a collection of resources in a generic collection (e.g., `java.util.List`) and then resolves the obligations of each element of the collection at once (e.g., by iterating over the `List`). In the future, the checker will support this. The remainder of this section explains the implementation issues; most users can skip it.

The first implementation issue is that `@Owning` and `@NotOwning` are declaration annotations rather than type qualifiers, so they cannot be written on type arguments. It is possible under the current design to have an `@Owning List<Socket>`, but not a `List<@Owning Socket>`. It would be better to make `@Owning` a type annotation, but this is a challenging design problem.

The second implementation issue is the defaulting rule for `@MustCall` on type variable upper bounds. Currently, this default is `@MustCall({})`, which prevents many false positives in code with type variables that makes no use of resources — an important design principle. However, this defaulting rule does have an unfortunate consequence: it is an error to write `List<Socket>` or any other type with a concrete type argument where the type argument itself isn't `@MustCall({})`. A programmer who needs to write such a type while using the Resource Leak Checker has a few choices, all of which have some downsides:

- Write `List<? extends Socket>`. This rejects calls to `add()` or other methods that require an instance of the type variable, but it preserves some of the behavior (e.g., calls to `remove()` are permitted). This is the best choice most of the time if the `List` is not intended to be owning.
- Write `List<@MustCall Socket>`. This makes it an error to add a Socket to the list, since the type of the Socket is `@MustCall("close")` but the list requires `@MustCall({})`.
- Suppress one or more warnings.

The recommended way to use the Resource Leak Checker in this situation is to rewrite the code to avoid a `List` of owning resources. If rewriting is not possible, the programmer will probably need to suppress a warning and then verify the code using a method other than the Resource Leak Checker.

8.11 Further reading

The paper "Lightweight and Modular Resource Leak Verification" [KSSE21] (ESEC/FSE 2021, `https://homes.cs.washington.edu/~mernst/pubs/resource-leak-esecfse2021.pdf`) gives more details about the Resource Leak Checker.

The Resource Leak Checker uses a specialized algorithm to infer annotations when whole program inference (WPI, Section 35.2) is enabled. The algorithm is described in the paper "Inference of Resource Management Specifications" [SGT+23] (OOPSLA 2023, `https://homes.cs.washington.edu/~mernst/pubs/resource-inference-oopsla2023-abst` `html`).

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 30, page 183), 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 retains 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 `@Fenum` type qualifier 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;
   ```

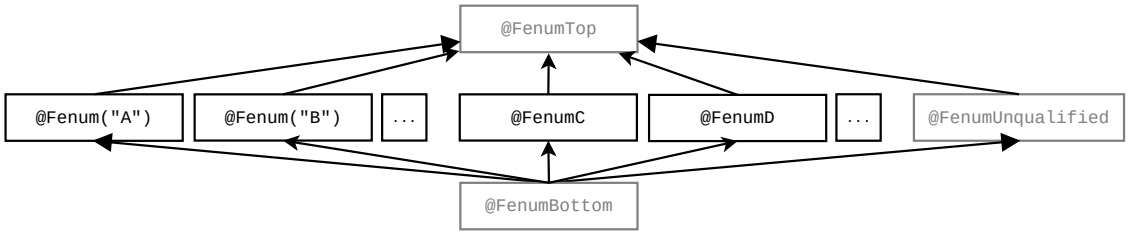


Figure 9.1: Partial type hierarchy for the Fenum type system. There are two forms of fake enumeration annotations — above, illustrated by `@Fenum("A")` and `@FenumC`. See Section 9.1 for descriptions of how to introduce both types of fenums. The type qualifiers in gray (`@FenumTop`, `@FenumUnqualified`, 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 37.5.1.

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 31) 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. Before you run the Fenum Checker with `javac`, they must be compiled and on the same path (the classpath or processorpath) as the Checker Framework. 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, if the Checker Framework is on the classpath rather than the processorpath:

```
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") // 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")   // 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 fenumTest() {
    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.

The paper "Building and using pluggable type-checkers" [DDE+11] (ICSE 2011, `https://homes.cs.washington.edu/~mernst/pubs/pluggable-checkers-icse2011.pdf`) describes case studies of the Fake Enum Checker.

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 a 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, `https://homes.cs.washington.edu/~mernst/pubs/pluggable-checkers-icse2011.pdf#page=3`)
- Java Language Specification on enums:
  `https://docs.oracle.com/javase/specs/jls/se25/html/jls-8.html#jls-8.9`
- Tutorial trail on enums:
  `https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html`
- Java Tip 122: Beware of Java typesafe enumerations:
  `https://www.infoworld.com/article/2077487/java-tip-122--beware-of-java-typesafe-enumerations.html`

Chapter 10

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 40.4.6).

10.1 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 a lock when correctness requires holding it throughout a computation. And, there are other concurrency errors that cannot, or should not, be solved with locks.

10.2 Lock annotations

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

10.2.1 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.

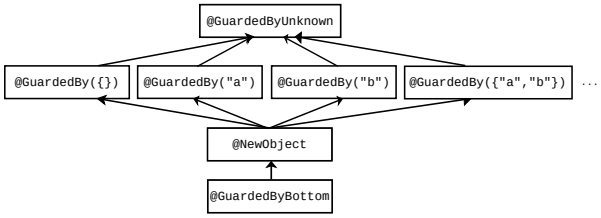


Figure 10.1: The subtyping relationship of the Lock Checker's qualifiers. `@GuardedBy({})` is the default type qualifier for unannotated types (except all CLIMB-to-top locations other than upper bounds and exception parameters — see Section 33.5.3).

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 33.8 explains which expressions the Lock Checker is able to analyze as lock expressions. These include `<self>`, 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 33.5.3). (Section 10.5.4 discusses this choice.) It is also the conservative default type qualifier for method parameters in unannotated libraries (see Chapter 36, page 239).

`@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 36, page 239).

`@NewObject` This type represents a newly-constructed value, such as the result of calling a constructor or factory method. The client can use the value as any `@GuardedBy` type.

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

Figure 10.1 shows the type hierarchy of these qualifiers. All `@GuardedBy` annotations are incomparable: if *exprSet1* ≠ *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 10.4.2 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 32.2), 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 10.4.6 for an example of a use of `@GuardSatisfied`.

10.2.2 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[] value) All the given lock expressions are held at the method call site.

`@EnsuresLockHeld`(String[] value) 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[] expression, 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 10.4.4 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 36, page 239).

10.3 Type-checking rules

In addition to the standard subtyping rules enforcing the subtyping relationship described in Figure 10.1, the Lock Checker enforces the following additional rules.

10.3.1 Polymorphic qualifiers

`@GuardSatisfied` The overall rules for polymorphic qualifiers are given in Section 32.2.
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 an 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`.

10.3.2 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.

10.3.3 Primitive types, boxed primitive types, and Strings

Primitive types, boxed primitive types (such as `java.lang.Integer`), and type `java.lang.String` are 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({})`.

10.3.4 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 those in *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
```

10.3.5 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 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 33.7.5.

10.4 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 locks named in the `@GuardedBy` annotation.

10.4.1 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 variable to which the value was originally assigned 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.
    }
  }
}
```

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

@GuardedBy(exprSet)

The following example demonstrates the reason the Lock Checker enforces this 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.

10.4.3 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
    }
  }
```

10.4.4 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();
    }

    ...
}
```

10.4.5 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
  }
}
```

10.4.6 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 locks 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);
```

10.5 More locking details

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

The paper "Locking discipline inference and checking" [ELM+16] (ICSE 2016, `https://homes.cs.washington.edu/~mernst/pubs/locking-inference-checking-icse2016-abstract.html`) gives additional details.

10.5.1 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 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`.

10.5.2 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 33.8 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.

10.5.3 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 33.7.

10.5.4 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.

10.5.5 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 of checking them.

Method summary that simplifies reasoning `@Holding` can be a method summary that simplifies reasoning. In this case, the `@Holding` annotation 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 given 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 than to write the extra synchronized block. Here are the 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.

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

Figure 10.2: Correspondence between other lock annotations and the Checker Framework's annotations.

- 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 when it succeeds.

10.6 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 10.2. If the other annotation is a declaration annotation, it may be moved; see Section 40.6.10.

10.6.1 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 as 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.)

10.7 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 11

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 type.

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 11.9).

To run the Index Checker, run either of these commands:

```
javac -processor index MyJavaFile.java
javac -processor org.checkerframework.checker.index.IndexChecker 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) []
```

11.1 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 11.2)
- the upper bound on an integer, such as whether it is less than the length of a given sequence (Section 11.3)
- whether an integer came from calling the JDK's binary search routine on an array (Section 11.6)
- whether an integer came from calling a string search routine (Section 11.7)

and about sequence lengths:

- the minimum length of a sequence, such as "`myArray` contains at least 3 elements" (Section 11.4)
- whether two sequences have the same length (Section 11.5)

The Index Checker checks 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") int 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 32.2.

`@PolyLength` is a special polymorphic qualifier that combines `@PolySameLen` and `@PolyValue` from the Constant Value Checker (see Chapter 24, page 161). `@PolyLength` exists as a shorthand for these two annotations, since they often appear together.

11.2 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 11.1) 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 32.2.

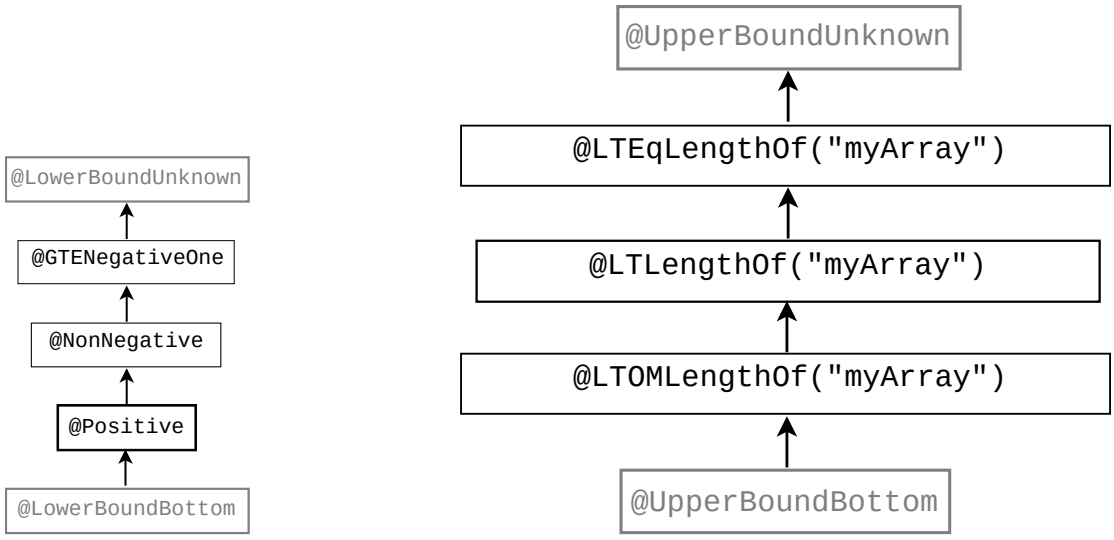


Figure 11.1: The two type hierarchies for integer types used by the Index Checker. On the left is a type system for lower bounds. On the right is a type system for upper bounds. Qualifiers written in gray should never be written in source code; they are used internally by the type system.
In the Upper Bound type system, subtyping rules depend on both the array name (`"myArray"`, in the figure) and on the offset (which is 0, the default, in the figure). Another qualifier is `@UpperBoundLiteral`, whose subtyping relationships depend on its argument and on offsets for other qualifiers.

`@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` There are no values of this type. This is the bottom type, which should never need to be written by the programmer.

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

It uses a type system (Figure 11.1) with the following qualifiers:

`@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".)
`@LTEqLengthOf({"a"})` = `@LTLengthOf(value={"a"}, offset=-1)`, and
`@LTEqLengthOf(value={"a"}, offset=x)` = `@LTLengthOf(value={"a"}, offset=x-1)` for any x.

`@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 be 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`".)
`@LTOMLengthOf({"a"})` = `@LTLengthOf(value={"a"}, offset=1)`, and
`@LTOMLengthOf(value={"a"}, offset=x)` = `@LTLengthOf(value={"a"}, offset=x+1)` for any x.

`@UpperBoundLiteral` represents a constant value, typically a literal written in source code. Its subtyping relationship is: `@UpperBoundLiteral(lit)` <: `@LTLengthOf(value="myArray", offset=off)` if `lit+off` $\leq$ -1.

`@PolyUpperBound` indicates qualifier polymorphism. For a description of qualifier polymorphism, see Section 32.2.

`@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 method postcondition that 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 precisely, 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.

There is one declaration annotation that indicates the relationship between two sequences:

`@HasSubsequence(String[] value, String[] from, String[] to)` indicates that a subsequence (from `from` to `to`) of the annotated sequence is equal to some other sequence, named by `value`.
For example, to indicate that `shorter` is a subsequence of `longer`:

```
int start;
int end;
int[] shorter;
@HasSubsequence(value="shorter", from="this.start", to="this.end")
int[] longer;
```

Thus, a valid index into `shorter` is also a valid index (between `start` and `end-1` inclusive) into `longer`. More generally, if `x` is `@IndexFor("shorter")` in the example above, then `start + x` is `@IndexFor("longer")`.

If `y` is `@IndexFor("longer")` and `@LessThan("end")`, then `y - start` is `@IndexFor("shorter")`. Finally, `end - start` is `@IndexOrHigh("shorter")`.
This annotation is in part checked and in part trusted. When an array is assigned to `longer`, three facts are checked: that `start` is non-negative, that `start` is less than or equal to `end`, and that `end` is less than or equal to the length of `longer`. This ensures that the indices are valid. The programmer must manually verify that the value of `shorter` equals the subsequence that the annotation describes.

11.4 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 24, page 161). 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:

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

The following type qualifiers (from Chapter 24, page 161) 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)`. A `@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 10 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 method postcondition that 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.

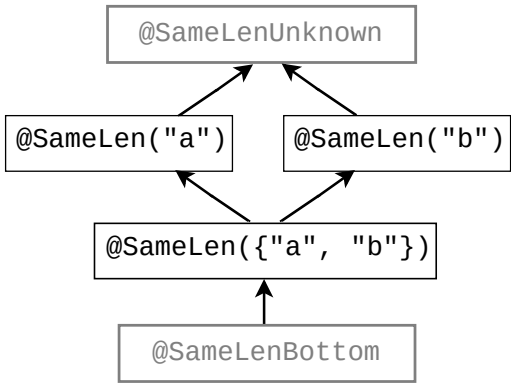


Figure 11.2: The type hierarchy for arrays of equal length ("a" and "b" are assumed to be in-scope sequences). Qualifiers written in gray should never be written in source code; they are used internally by the type system.

11.5 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]) {
      return false;
    }
  }
  return false;
}
```

When needed, you can specify which sequences have the same length using the following type qualifiers (Figure 11.2):

`@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 32.2.

`@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.

11.6 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 11.3) has four type qualifiers:

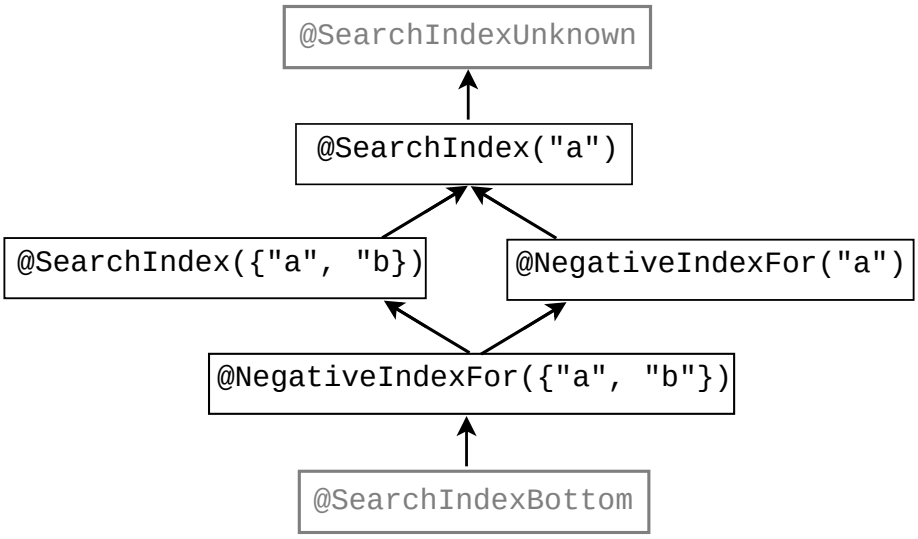


Figure 11.3: The type hierarchy for the Index Checker's internal type system that captures information about the results of calls to `Arrays.binarySearch`.

`@SearchIndexFor(String[] names)` An expression with this type represents an integer that could have been produced by calling `Arrays.binarySearch`: for each array `a` specified 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 `@SearchIndexFor` and 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.

`@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.

11.7 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;
}
```

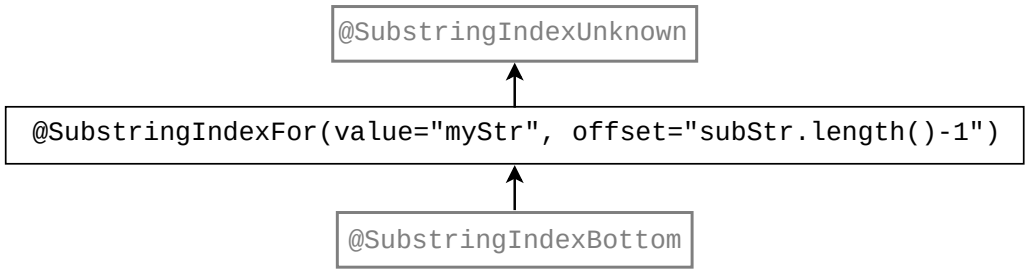


Figure 11.4: The type hierarchy for the Substring Index Checker, which captures information about the results of calls to `String.indexOf` and `String.lastIndexOf`.

The `@SubstringIndexFor` annotation is implemented in a Substring Index Checker that runs together with the Index Checker and has its own type hierarchy (Figure 11.4) 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 `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.

11.7.1 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 11.2) and upper-bound (Section 11.3) 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.

11.8 Inequalities

The Index Checker estimates which expressions' 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 of this type relative to other expressions. This is the top type, and should not be written by the programmer.

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

11.9 Annotating your own fixed-size datatypes

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 type-check the data structure's implementation and to type-check 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 a call to a method annotated with `@LengthOf("this")` like `arr.length` for arrays and `str.length()` for strings.

With these annotations, client code like the following type-checks 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);
  }
}
```

11.10 Technical papers

The paper "Lightweight Verification of Array Indexing" (ISSTA 2018, `https://homes.cs.washington.edu/~mernst/pubs/array-indexing-issta2018-abstract.html`) gives more details about the Index Checker. "Enforcing correct array indexes with a type system" [San16] (FSE 2016) describes an earlier version.

Chapter 12

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` or `-processor org.checkerframework.checker.taint` command-line option to javac.

12.1 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` indicates qualifier polymorphism. For a description of qualifier polymorphism, see Section 32.2.

12.2 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 33.5 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 where 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 36 describes how to annotate external libraries):

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

12.3 `@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 30). 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 37 for more details.

12.4 A caution about polymorphism and side effects

Misuse of polymorphism can lead to unsoundness with the Tainting Checker and other similar information flow checkers. To understand the potential problem, consider the `append` function in `java.lang.StringBuffer`:

```
public StringBuffer append(StringBuffer this, String toAppend);
```

Given these declarations:

```
@Tainted StringBuffer tsb;
@Tainted String ts;
@Untainted StringBuffer usb;
@Untainted String us;
```

both of these invocations should be legal:

```
tsb.append(ts);
usb.append(us);
```

That suggests that perhaps the function should be annotated as polymorphic:

```
// UNSOUND annotation -- do not do this!
public @PolyTainted StringBuffer append(@PolyTainted StringBuffer this, @PolyTainted String toAppend);
```

The problem with the above annotation is that it permits the undesirable invocation:

```
usb.append(ts); // undesirable invocation
```

This invocation is permitted because, in the expression, all `@PolyTainted` annotations on formal parameters are instantiated to `@Tainted`, the top annotation, and each argument is a subtype of the corresponding formal parameter.

Beware this problem both in code you write, and also in annotated libraries (such as stub files). The correct way to annotate this class is to add a class qualifier parameter; see Section 32.3.

(Side note: if `append` were purely functional (had no side effects and returned a new `StringBuffer`), the method call would be acceptable, because the return type is instantiated to `@Tainted StringBuffer` for the expression `usb.append(ts)`. However, the `append` method works via side effects, and only returns a reference to the buffer as a convenience for writing "fluent" client code.)

Chapter 13

SQL Quotes Checker

The SQL Quotes Checker helps prevent SQL injection vulnerabilities. Many SQL injection attacks involve malicious user input containing unescaped special characters such as the single quote, which can change whether subsequent concatenations are interpreted as SQL command code or as SQL query values.

The SQL Quotes Checker determines whether a string contains an unescaped single quote character. Furthermore, it marks user-supplied Strings as unsafe for SQL query use, because their quoting is unknown; the program must *sanitize* or *validate* all such values before concatenating them to a SQL query for execution. Sanitization can be done by quoting user input as necessary to ensure special characters are properly escaped. The SQL Quotes Checker follows the ANSI standard and considers two consecutive single quote characters as one escaped single quote. (Some databases support the use of the backslash as an escape character for single quotes. The SQL Quotes Checker does not support that syntax because it is not safe for all SQL database implementations.)

The SQL Quotes Checker guarantees that no unchecked values are passed to a SQL query execution statement. It does *not* guarantee that SQL queries written by the code author are safe for execution; e.g., the checker will not warn at compile time about syntax errors or improper uses of `DELETE` or `DROP` statements.

When possible, it is better code style to use a prepared statement or other mechanisms that automatically handle quoting without the need for the programmer to do any work. The SQL Quotes Checker is useful for the large quantity of code that still uses string concatenation to create SQL queries.

To run the SQL Quotes Checker, supply the `-processor SqlQuotesChecker` or `-processor org.checkerframework.checker.s` command-line option to javac.

13.1 SQL Quotes annotations

The SQL Quotes type system uses the following annotations:

`@SqlEvenQuotes` indicates a String literal that contains an even number of single quotes (possibly zero).
`@SqlOddQuotes` indicates a String literal that contains an odd number of single quotes.
`@SqlQuotesUnknown` indicates a String whose quoting is unknown. It is the default qualifier for non-literal Strings.
`@SqlQuotesBottom` is the bottom qualifier. Programmers rarely need to write it.

The subtyping hierarchy of the SQL Quotes Checker's qualifiers is shown in Figure 13.1.

13.2 What the SQL Quotes Checker checks

Concatenation of `@SqlEvenQuotes` and `@SqlOddQuotes` Strings parallels integer parity over addition. In other words, concatenation of `@SqlOddQuotes` and `@SqlEvenQuotes` Strings is most narrowly typed as follows:

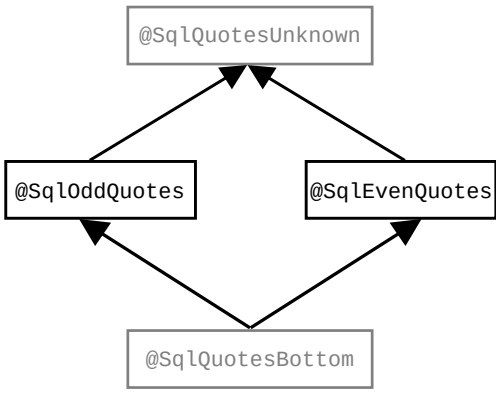


Figure 13.1: The subtyping relationship of the SQL Quotes Checker's qualifiers. The type qualifiers are applicable to Strings. Qualifiers in gray are used internally by the type system but should never be written by a programmer.

```
@SqlOddQuotes + @SqlOddQuotes = @SqlEvenQuotes
@SqlOddQuotes + @SqlEvenQuotes = @SqlOddQuotes
@SqlEvenQuotes + @SqlOddQuotes = @SqlOddQuotes
@SqlEvenQuotes + @SqlEvenQuotes = @SqlEvenQuotes
```

13.3 Library annotations

The programmer needs to write trusted annotations on SQL-related methods. This is already done for you, for commonly-used libraries such as `java.sql` and `org.springframework.security.crypto.bcrypt`. (If you annotate other libraries, please contribute them to the Checker Framework project. Thanks!)

A SQL query execution method such as `Statement.executeQuery` must be annotated to take a `@SqlEvenQuotes` string as a parameter.

A sanitization or quoting method should be annotated to take a `@SqlQuotesUnknown` parameter and to return a `@SqlEvenQuotes` result.

Given such annotations, the type system will issue an error if an un-sanitized value can ever flow into a database query. (You might need to write some annotations in your own code as well.)

Chapter 14

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. 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). These problems cause run-time errors.

Regular expressions in Java also have capturing groups, which are delimited by parentheses and allow for extraction from text. If a programmer uses an incorrect index (larger than the number of capturing groups), an `IndexOutOfBoundsException` is thrown.

The Regex Checker warns about these problems at compile time, guaranteeing that your program does not crash due to incorrect use of regular expressions.

For further details, including case studies, see the paper "A type system for regular expressions" [SDE12] (FTfJP 2012, `https://homes.cs.washington.edu/~mernst/pubs/regex-types-ftfjp2012-abstract.html`).

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

14.1 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. For a description of qualifier polymorphism, see Section 32.2.

The subtyping hierarchy of the Regex Checker's qualifiers is shown in Figure 14.1.

14.2 Annotating your code with `@Regex`

14.2.1 Implicit qualifiers

The Regex Checker adds implicit qualifiers, reducing the number of annotations that must appear in your code (see Section 33.4). If a `String` literal is a valid regex, the checker implicitly adds the `@Regex` qualifier with the argument set to the correct number of capturing groups. The Regex Checker allows the `null` literal to be assigned to any type qualified with the `Regex` qualifier.

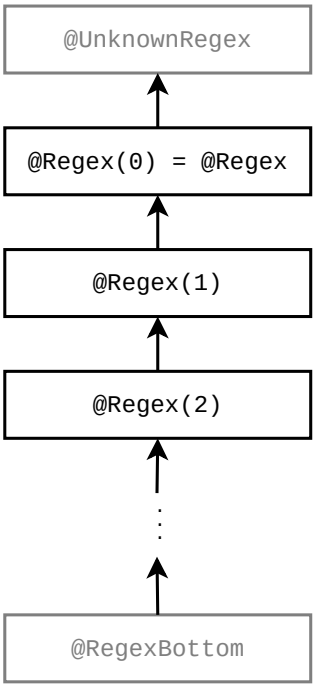


Figure 14.1: The subtyping relationship of the Regex Checker's qualifiers. The type qualifiers are applicable to `CharSequence` and its subtypes. Because the parameter to a `@Regex` qualifier is at least the number of capturing groups in a regular expression, a `@Regex` qualifier with more capturing 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 14.2: An example of the Regex Checker's support for concatenation of non-regular-expression Strings to produce valid regular expression Strings.

14.2.2 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 33.7), but you may need to add `@Regex` annotations with a capturing group count to `Pattern` and `Matcher` fields and parameters.

14.2.3 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 14.2.

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

Figure 14.3: Example use of `RegexUtil` methods.

14.2.4 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 14.3 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.

A 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 a `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 that it is checked to guarantee that the error will be handled up the call stack. For more details, see the Javadoc for `RegexUtil.CheckedPatternSyntaxException`.

To use the `RegexUtil` class, the `checker-util.jar` file must be on the classpath at run time.

14.2.5 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 14.3 should support most cases, so the `asRegex` method should be used rarely.

Chapter 15

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 15.3 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.

The paper "A type system for format strings" [WKSE14] (ISSTA 2014, `https://homes.cs.washington.edu/~mernst/pubs/format-string-issta2014-abstract.html`) gives more details about the Format String Checker and the Internationalization Format String Checker (Chapter 16, page 123).

15.1 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.

15.2 Format String Checker annotations

The `@Format` qualifier on a string type indicates 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 15.2.1. The type qualifier for `"%d %f"` is, for example, `@Format({INT, FLOAT})`.

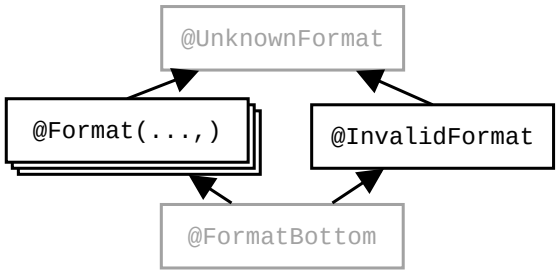


Figure 15.1: The Format String Checker type qualifier hierarchy. The type qualifiers are applicable to `CharSequence` and its subtypes. The figure does not show the subtyping rules among different `@Format(...)` qualifiers; see Section 15.2.2.

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.

```
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 15.1 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.

There is also a `@FormatMethod` annotation; see Section 15.5.

15.2.1 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 15.3.2.) 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.

All conversion categories accept `null`. Furthermore, `null` is always a legal argument array, because it is treated as supplying `null` to each format specifier. For example, `String.format("%d %f %s", (Object[]) null)` evaluates to `"null null 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 to 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`. For example, the format string `"%1$f %1$c"` requires that the first format argument be `null`. Passing either `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 15.2 summarizes the subset relationship among all conversion categories.

15.2.2 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);
```

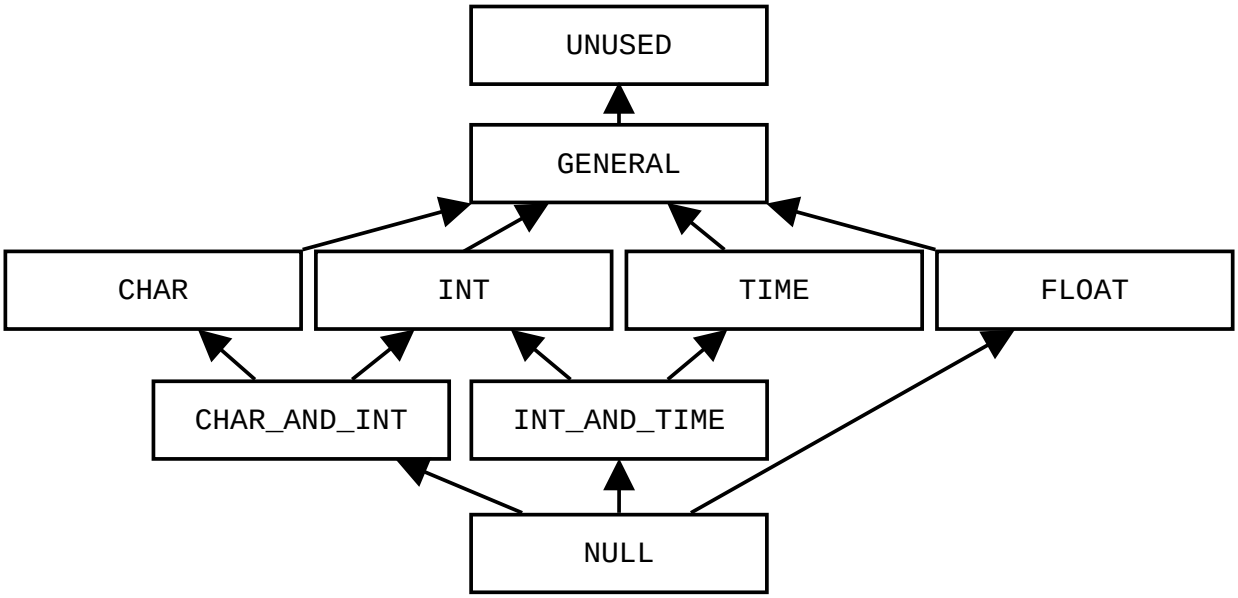


Figure 15.2: The subset relationship among conversion categories.

15.3 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 argument (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 of 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)
```

15.3.1 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.

15.3.2 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)
```

15.4 Implicit qualifiers

The Format String Checker adds implicit qualifiers, reducing the number of annotations that must appear in your code (see Section 33.4). The checker implicitly adds the `@Format` qualifier with the appropriate conversion categories to any String literal that is a valid format string.

15.5 @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.

15.6 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);
}
// `format` is now known to be of type: @Format({GENERAL, INT}) String
System.out.printf(format, "hello", 1337);
```

To use the `FormatUtil` class, the `checker-util.jar` file must be on the classpath at run time.

Chapter 16

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 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 16.6.

The Internationalization Checker or I18n Checker (Section 17.2, page 130) 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.

The paper "A type system for format strings" [WKSE14] (ISSTA 2014, `https://homes.cs.washington.edu/~mernst/pubs/format-string-issta2014-abstract.html`) gives more details about the Internationalization Format String Checker and the Format String Checker (Chapter 15, page 116).

16.1 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 16.1 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

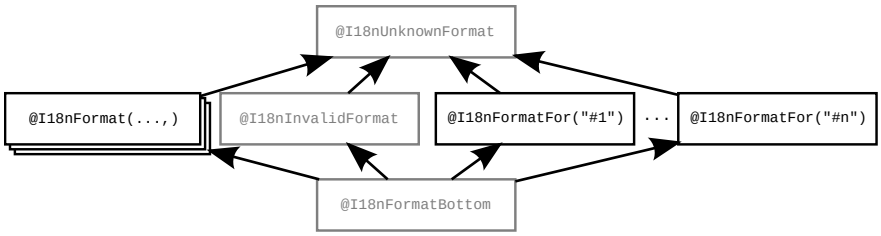


Figure 16.1: The Internationalization Format String Checker type qualifier hierarchy. The type qualifiers are applicable to `CharSequence` and its subtypes. The figure does not show the subtyping rules among different `@I18nFormat(...)` qualifiers; see Section 16.2. All `@I18nFormatFor` annotations are unrelated by subtyping, unless they are identical. The qualifiers in gray are used internally by the checker and should never be written by a programmer.

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 16.2.

`@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 33.8. 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.

16.2 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`, or `choice`. These four format types correspond 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 16.2 summarizes the subset relationship among all conversion categories.

16.3 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.

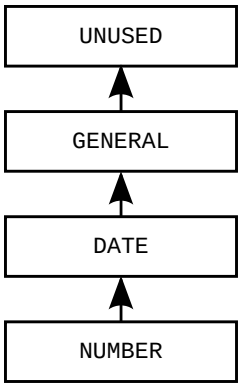


Figure 16.2: The subset relationship among i18n conversion categories.

- 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:

```
@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}`, are `(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` or `Number` type or a subclass of them, including `Time`, `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`.

16.4 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 34).

(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 a warning will be issued.
format = "{0}";

// Error, the format string has too many arguments.
format = "{0} {1} {2}";
// Error, the format string is more restrictive. NUMBER is a subtype of GENERAL.
format = "{0, number} {1, number}";
```

16.5 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 17, page 129.

16.6 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
  ```

16.7 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 16.8.

To use the `I18nFormatUtil` class, the `checker-util.jar` file must be on the classpath at run time.

16.8 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;
s1 = "{0}";
s1 = "{0, number}";        // error: incompatible types in assignment
```

Chapter 17

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. An access without a valid key does not fail in an obvious way: `ResourceBundle.getString` throws a `MissingResourceException`, and `Properties.getProperty` returns `null` or the default value supplied by the caller. This can lead to a `NullPointerException` or hard-to-trace behavior. The Property File Checker (Section 17.1, page 129) ensures that the keys used are found in the corresponding property file or resource bundle.

We also provide two specialized checkers. An Internationalization Checker (Section 17.2, page 130) verifies that code is properly internationalized. A Compiler Message Key Checker (Section 17.3, page 130) 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.

17.1 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 `CharSequence` 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 `CharSequence` (including `String`) variable to be eventually used as a key, you also need to annotate all these variables with `@PropertyKey`.

The checker can be invoked by running the following command:

```
javac -processor org.checkerframework.checker.propkey.PropertyKeyChecker
      -Abundlenames=MyResource MyFile.java ...
```

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

1. `-Abundlenames=`*`resource_name`*
   resource_name is the name of the resource to be used with `ResourceBundle.getBundle()`. The checker uses the default `Locale` and `ClassLoader` in the compilation system. (For a tutorial about `ResourceBundles`, see `https://docs.oracle.com/javase/tutorial/i18n/resbundle/concept.html`.) Multiple resource bundle names are separated by colons ":".
2. `-Apropfiles=`*`prop_file`*
   prop_file is the name of a properties file that maps keys to values. The file format is described in the Javadoc for `Properties.load()`. Multiple files are separated by Java's `File.pathSeparator` (semicolon ";" on Windows, colon ":" on Linux and Mac).

17.2 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 text 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 text. 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 16). 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.

17.2.1 Internationalization annotations

The Internationalization Checker supports two type systems:

`@Localized` indicates that the qualified `CharSequence` is a message that has been localized and/or formatted with respect to the used locale. It is a subtype of `@UnknownLocalized`, which is the default and which programmers do not need to write.

`@LocalizableKey` indicates that the qualified `CharSequence` or `Object` is a valid key found in the localization resource. This annotation is a specialization of the `@PropertyKey` annotation, which gets checked by the general Property Key Checker. It is a subtype of `@UnknownLocalizableKey`, which is the default and which programmers do not need to write.

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

17.2.2 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 text. 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.

17.3 Compiler Message Key Checker

The Checker Framework uses compiler message keys to output error messages. These keys are replaced 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 message 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.DiagMessage` uses `@CompilerMessageKey` in its constructor and in method `error`. 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 text. 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 18

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 18.1), 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.

The paper "Building and using pluggable type-checkers" [DDE+11] (ICSE 2011, `https://homes.cs.washington.edu/~mernst/pubs/pluggable-checkers-icse2011.pdf`) describes case studies of the Signature String Checker.

18.1 Signature annotations

Java defines six formats for the string representation of a type. There is an annotation for each of these representations. Figure 18.1 shows how they are related; examples appear in a table below.

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

`@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.

`@FieldDescriptor` A *field descriptor* (JVMS §4.3.2), such as `Lmypackage/Outer$Inner;`, is used in a `.class` file's constant pool, for example to refer to other types. It abbreviates primitive types and array types. It uses internal form (binary names, but with `/` instead of `.`; see JVMS §4.2) for class names. See examples below.

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

`@InternalForm` The *internal form* (JVMS §4.2), such as `mypackage/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`.

`@FqBinaryName` An extension of binary name format to represent primitives and arrays. It is like `@FullyQualifiedName`, but using "`$`" instead of "`.`" to separate nested classes from their enclosing classes. For example, `"pkg.Outer$Inner"`

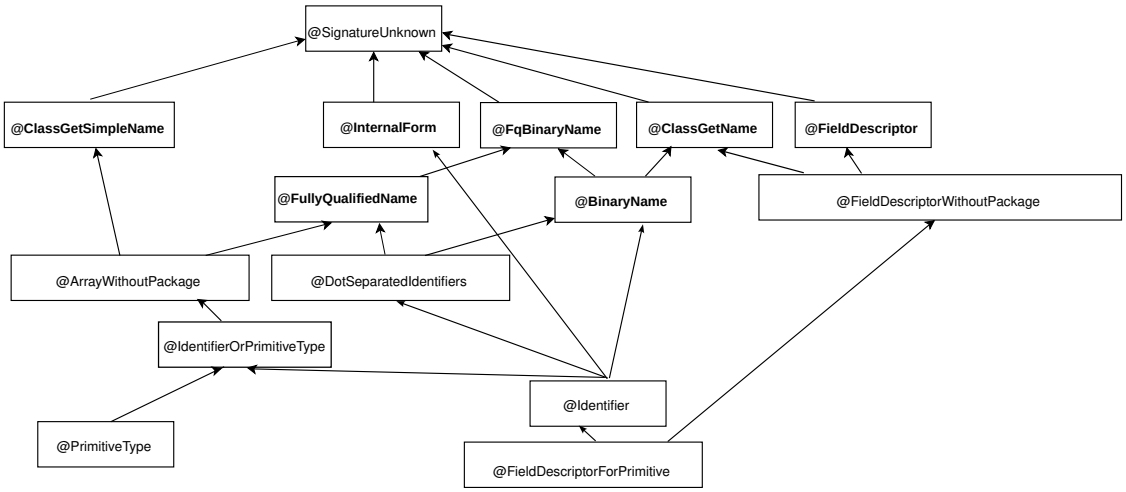


Figure 18.1: Partial type hierarchy for the Signature type system, showing string representations of a Java type. The type qualifiers are applicable to `CharSequence` and its subtypes. Programmers usually only need to write the boldfaced qualifiers. The other qualifiers (and some not shown) are included to improve the internal handling of String literals.

or "`pkg.Outer$Inner[][]`" or "`int[]`".

`@CanonicalName` Syntactically identical to `@FullyQualifiedName`, but some classes have multiple fully-qualified names, only one of which is canonical (see JLS §6.7).

Other type qualifiers are the intersection of two or more qualifiers listed above; for example, a `@DotSeparatedIdentifiers` is a string that is a valid fully-qualified name *and* a valid binary name. A programmer should rarely or never use these qualifiers, and you can ignore them as implementation details of the Signature Checker, though you might occasionally 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, notably qualified names (JLS §6.2). The Signature Checker does not include annotations for these.

Here are examples of the supported formats:

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

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;
```

18.1.1 How to choose which annotation to use

Sometimes, there are multiple valid annotations for a value. As an example, a non-primitive non-array type is represented identically by `@BinaryName`, `@ClassGetName`, and `@FqBinaryName`. Using the lowest type in the type hierarchy (in this case, `@BinaryName`) has two advantages. First, it acts as documentation that the value is never a primitive or array. Second, it permits the value to be used in any of the three contexts: as a `@BinaryName`, a `@ClassGetName`, or a `@FqBinaryName`.

Casting to `@BinaryName` from one of the other types adds clutter if it is not necessary. Suppose that a method returns a `@ClassGetName` and the value will only be used in contexts that require a `@ClassGetName` (say, it is passed to a method that requires an argument of that type). Then there is no point in casting to `@BinaryName` in between, even if you know the type being represented is not a primitive or an array.

18.2 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 19

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 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 19.1),
- class or package annotations controlling the default effect (Section 19.4), and
- *effect-polymorphism*: code that works for both the safe effect and the UI effect (Section 19.5).

19.1 GUI effect annotations

There are two primary GUI effect annotations:

`@SafeEffect` is a method annotation marking code that must not access UI objects.

`@UIEffect` is a method annotation marking code that may access UI objects. Most UI object methods (e.g., methods of `JPanel`) are annotated as `@UIEffect`.

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

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

19.2 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).

19.3 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 ...
```

19.4 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 19.5.2.)

`@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.

19.5 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;`.

19.5.1 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.

19.5.2 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.

Note that no annotation is required on the anonymous class declaration itself (e.g., `new Runnable(){...}` does not require a type use annotation, although the variable, field, or argument it ends up being assigned to might). Instead, the GUI Effect Checker will infer the effect qualifier based on the method being called from within the members of that specific anonymous class.

19.5.3 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.)

19.5.4 Subtyping with polymorphic effects

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

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

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

This induces a subtyping hierarchy on type qualifiers:

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

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

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 32.2).

`@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 an `@AlwaysSafe` receiver.

As noted earlier (Section 32.1.7), 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 implements Dispatcher {
    @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 an `@AlwaysSafe` receiver. That change is permissible. But when that is done, the polymorphic arguments may no longer be interchangeable with the receiver.

19.6 References

The paper "JavaUI: Effects for controlling UI object access" [GDEG13] (ECOOP 2013, `https://homes.cs.washington.edu/~mernst/pubs/gui-thread-ecoop2013-abstract.html`) 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.

Chapter 20

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 newton 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. The assignment of an unqualified value to `meters`, as in `meters = 99`, will be flagged as an error by the Units Checker. Utility class `UnitsTools` provides constants that you can multiply with an unqualified integer to get values of the corresponding unit; for example, `meters = 99 * UnitsTools.m` is legal, or just `meters = 99 * m` if the file contains `import static org.checkerframework.checker.units.util.UnitsTools.*;`. To use the `UnitsTools` class, the `checker-util.jar` file must be on the classpath at run time.

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.

20.1 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`
`@Force`
`@Length`
`@Luminance`
`@Mass`
`@Speed`
`@Substance`

@Temperature
@Time
@Volume

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

1. For `@Acceleration`: meters per second squared `@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`: amperes `@A`
5. For `@Force`: newtons `@N` and the derived unit kilonewton `@kN`
6. For `@Length`: meters `@m` and the derived units millimeters `@mm` and kilometers `@km`
7. For `@Luminance`: candelas `@cd`
8. For `@Mass`: kilograms `@kg` and the derived units grams `@g` and metric tons `@t`
9. For `@Speed`: meters per second `@mPERs` and kilometers per hour `@kmPERh`
10. For `@Substance`: moles `@mol`
11. For `@Temperature`: kelvins `@K` and the derived unit degrees Celsius `@C`
12. For `@Time`: seconds `@s` and the derived units minutes `@min` and hours `@h`
13. For `@Volume`: the derived units cubic millimeters `@mm3`, cubic meters `@m3`, and cubic kilometers `@km3`

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 a 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. Static imports allow 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. To use the `UnitsTools` class, the `checker-util.jar` file must be on the classpath at run time.

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 32.2. For an example of its use, see the `@PolyUnit` Javadoc.

20.2 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)
@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})
@SubtypeOf({Time.class})
@UnitsMultiple(quantity=s.class, prefix=Prefix.nano)
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 37.5.1.

To take full advantage of the additional unit qualifier, you need to take 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.

20.3 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 by 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.)

20.4 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. Before you run the Units Checker with `javac`, they must be compiled and on the same path (the classpath or processorpath) as the Checker Framework. 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, if the Checker Framework is on the classpath rather than the processorpath:

  ```
  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.

20.5 Suppressing warnings

One example of when you need to suppress warnings is when you initialize a variable of a unit type with 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`.

20.6 References

- The GNU Units tool provides a comprehensive list of units:
  `https://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 21

Signedness Checker

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

Recall that a computer represents a number as a sequence of bits. Signedness indicates how to interpret the most significant bit. For example, the bits `10000010` ordinarily represent the value -126, but when interpreted as unsigned, those bits represent the value 130. The bits `01111110` represent the value 126 in signed and in unsigned interpretation. The range of signed byte values is -128 to 127. The range of unsigned byte values is 0 to 255.

Signedness is only applicable to the integral types `byte`, `short`, `int`, and `long` and their boxed variants `Byte`, `Short`, `Integer`, and `Long`. `char` and `Character` are always unsigned. Floating-point types `float`, `double`, `Float`, and `Double` are always signed.

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 non-negative, but just because a variable's value is non-negative 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. To indicate the range of possible values for a variable, use the `@NonNegative` annotation of the Index Checker (see Chapter 11, page 98) or the `@IntRange` annotation of the Constant Value Checker (see Chapter 24, page 161).

The Signedness Checker trusts a `@NonNegative` or `@Positive` annotation on a `byte`, `short`, `int`, or `long` expression. To verify these annotations, also run the Index Checker. A `char` is always non-negative, so these annotations do not affect a `char` expression; a `char` expression has type `@SignedPositive` only when range analysis determines that its value is between 0 and 127.

Additional details appear in the paper "Preventing signedness errors in numerical computations in Java" [Mac16] (FSE 2016).

To run the Signedness Checker, run javac with `-processor org.checkerframework.checker.signedness.SignednessChecker`

21.1 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.

`@SignedPositive` indicates that a value is known at compile time to be in the non-negative signed range, so it has the same interpretation as signed or unsigned and may be used with either interpretation. (Equivalently, the most significant bit is guaranteed to be 0.) Programmers should usually write `@Signed` or `@Unsigned` instead.

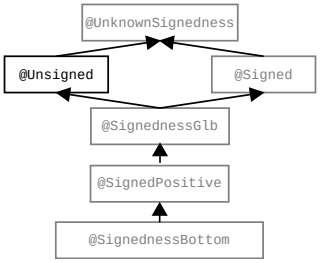


Figure 21.1: The type qualifier hierarchy of the signedness annotations. Qualifiers in gray are used internally by the type system but should never be written by a programmer.

`@SignednessGlb` indicates that a value may be interpreted as unsigned or signed. It covers the same cases as `@SignedPositive`, plus manifest literals, to prevent the programmer from having to annotate them all explicitly. *Programmers should rarely write this annotation, except on fields whose value is a negative manifest literal.*

`@PolySigned` indicates qualifier polymorphism. When two formal parameter types are annotated with `@PolySigned`, the two arguments at a call site must have the same signedness type annotation. (This differs from the standard rule for polymorphic qualifiers.) For a description of qualifier polymorphism, see Section 32.2.

`@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.

21.1.1 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 `@SignednessGlb`.
- All `char` and `Character` expressions are `@Unsigned`; this cannot be changed.
- All `char` and `Character` variables are `@Unsigned`; this cannot be changed.
- All other expressions default to `@Signed`.

21.2 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 (`<<`).

There are some special cases where these operations are permitted; see Section 21.2.2.

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.

21.2.1 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:

```
public class SignednessManualExample {

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

    @Unsigned int u1 = 2147483646; // unsigned: 2^32 - 2, signed: -2
    @Unsigned int u2 = 2147483647; // unsigned: 2^32 - 1, signed: -1

    void m() {
        int w = s1 / s2; // OK: result is 2, which is correct for -2 / -1
        int x = u1 / u2; // ERROR: result is 2, which is incorrect for (2^32 - 2) / (2^32 - 1)
    }

    int s3 = -1;
    int s4 = 5;

    @Unsigned int u3 = 2147483647; // unsigned: 2^32 - 1, signed: -1
    @Unsigned int u4 = 5;

    void m2() {
        int y = s3 % s4; // OK: result is -1, which is correct for -1 % 5
        int z = u3 % u4; // ERROR: result is -1, which is incorrect for (2^32 - 1) % 5 = 2
    }
}
```

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

21.2.2 Permitted shifts

As exceptions to the rules given above, the Signedness Checker permits certain right shifts that are immediately followed by a cast or masking operation.

For example, right shift by 8 then mask by 0xFF evaluates to the same value whether the argument is interpreted as signed or unsigned. Thus, the Signedness Checker permits both `((myInt >> 8) & 0xFF)` and `((myInt >>> 8) & 0xFF)`, regardless of the qualifier on the type of `myInt`.

Likewise, right shift by 8 then cast to byte evaluates to the same value whether the argument is interpreted as signed or unsigned, so the Signedness Checker permits both `(byte) (myInt >> 8)` and `(byte) (myInt >>> 8)`, regardless of the type of `myInt`.

21.3 Utility routines for manipulating unsigned values

Class `SignednessUtil` provides static utility methods for working with unsigned values. They are properly annotated with `@Unsigned` where appropriate, so using them may reduce the number of annotations that you need to write. To use the `SignednessUtil` class, the `checker-util.jar` file must be on the classpath at run time.

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

jdk.jfr.Unsigned	⇒ org.checkerframework.checker.signedness.qual.Unsigned

Figure 21.2: Correspondence between other signedness annotations and the Checker Framework's annotations.

21.4 Local type refinement

Local type refinement/inference (Section 33.7) may be surprising for the Signedness type system. Ordinarily, an expression with unsigned type may not participate in a division, as shown in Sections 21.2 and 21.2.1. However, if a constant is assigned to a variable that was declared with `@Unsigned` type, then — just like the constant — the variable may be treated as either signed or unsigned, due to local type refinement (Section 33.7). For example, it can participate in division.

```
void useLocalVariables() {

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

    @Unsigned int u1 = 2147483646; // unsigned: 2^32 - 2, signed: -2
    @Unsigned int u2 = 2147483647; // unsigned: 2^32 - 1, signed: -1

    int w = s1 / s2; // OK: result is 2, which is correct for -2 / -1
    int x = u1 / u2; // OK; computation over constants, interpreted as signed; result is signed
}
```

To prevent local type refinement, use a cast:

```
    @Unsigned int u1 = (@Unsigned int) 2147483646;
```

Note that type-checking produces a different result for `int x = u1 / u2;` here than in the similar example in Section 21.2.1. In Section 21.2.1, the method is reading fields, and all it knows is the declared type of the field. In Section 21.4, the method is reading a local variable, and dataflow (that is, flow-sensitive type refinement) refines the types of local variables.

21.5 Instantiating polymorphism

When calling a method with formal parameters annotated as `@PolySigned`, all arguments for `@PolySigned` formal parameters must have comparable types. (One way to do this is for all types to be the same.) This is different than the usual rules for polymorphic qualifiers. If you violate this rule, then the Signedness Checker's error messages can be obscure, because they are about `@SignednessBottom`. You can fix the signedness error messages by casting the arguments.

21.6 Other signedness annotations

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

If your code is already annotated with a different 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 Signedness Checker, as described in Figure 21.2.

Chapter 22

Modifiability Checker

In the Java Collections Framework, an *optional* method is one that a class may or may not support. If a method is unsupported, calling it throws `UnsupportedOperationException`. The Modifiability Checker verifies that optional methods related to modifying collections are only called if they are supported, thus eliminating the possibility of an `UnsupportedOperationException` at run time. (It is possible to write a checker that warns about other unsupported methods, but the Modifiability Checker focuses on those issued when modifying collections.)

An example of an optional method is `add`. Some subclasses of `Collection` support `add`, and others (such as the result of calling `Collections.unmodifiableCollection`) do not support `add`.

If the Modifiability Checker issues no warning, then your program is guaranteed not to throw `UnsupportedOperationException` due to invoking an optional method (such as `add` or `remove`) on an unmodifiable collection. (For brevity, this chapter generally uses just "collection" to mean "collection, map, or iterator".)

In addition to preventing UOEs, the Modifiability Checker ensures that collections that the user intends to be unmodifiable have an unmodifiable run-time implementation.

To run the Modifiability Checker:

```
javac -processor org.checkerframework.checker.modifiability.ModifiabilityChecker MyFile.java
```

22.1 Unmodifiable collections

An *unmodifiable collection* is one that throws `UnsupportedOperationException` if a mutating method is called. The JDK provides many such collections, for example, the results of:

- `List.of()`, `List.copyOf()`
- `Set.of()`, `Set.copyOf()`
- `Map.of()`, `Map.copyOf()`
- `Collections.emptyList()`, `Collections.emptySet()`, `Collections.emptyMap()`
- `Collections.unmodifiableList()`, `Collections.unmodifiableSet()`, `Collections.unmodifiableMap()`

A *modifiable collection* supports calling at least one optional mutating method. In general, classes support groups of methods which correspond to four different ways a collection can be modified. A collection may support modifying a collection through one or more of the capabilities **Grow**, **Sequential Grow**, **Shrink**, and **Replace**.

- **Grow**: The collection can grow by adding new elements (`add`, `addAll`, `offer`, ...) at arbitrary locations. Not to be confused with sequential grow.
- **Sequential Grow**: The sequenced collection can grow by adding new elements at the beginning or end (`addFirst`, `addLast`, `offerFirst`, `offerLast`, `push`, `putFirst`, `putLast`, ...).
  The Sequential Grow capability is needed for collections, such as `TreeSet`, that support grow operations (like `add` and `addAll`) but do not support sequential grow operations (like `addFirst` and `addLast`). The declaration of `TreeSet` is annotated as `@Growable @SeqUngrowable`.

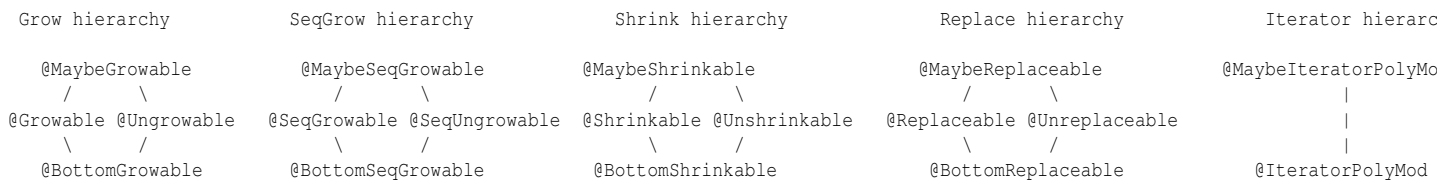


Figure 22.1: Independent modifiability qualifier hierarchies. Each annotated type carries one qualifier from each hierarchy.

"Sequential Grow" is shortened to "SeqGrow" in the rest of the manual.

- **Shrink**: The collection can shrink by removing existing elements (`remove`, `removeAll`, `clear`, `retainAll`, `poll`, `pop`, ...).
- **Replace**: The collection can replace existing elements with new values at specific locations/keys (`List.set`, `Map.replace`, `Map.replaceAll`, `ListIterator.set`, `Collections.sort`, `Map.Entry.setValue`, ...).

Some collections can also be modified through their iterator; see Section 22.4.

The following table summarizes the main method families that may be called for each capability. It is not a list of every overload. Read-only methods such as `get`, `contains`, `size`, `isEmpty`, `iterator`, and `entrySet` do not require a modifiability capability.

Capability required	Method families
`@Growable`	`Collection.add`, `addAll`, `List.add`, `List.addAll`, `Queue.add`, `Queue.offer`, `BlockingQueue.put`, timed `offer`, `TransferQueue.transfer`, `tryTransfer`, `Map.putIfAbsent`, `Map.computeIfAbsent`, `ListIterator.add`, legacy `Vector` insertion methods, and `Collections.addAll`
`@SeqGrowable`	`SequencedCollection.addFirst`, `addLast`, `Deque.addFirst`, `addLast`, `offerFirst`, `offerLast`, `push`, `BlockingDeque.putFirst`, `putLast`, timed `offerFirst`, timed `offerLast`, `SequencedMap.putFirst`, `putLast`
`@Shrinkable`	`Collection.remove`, `removeAll`, `retainAll`, `removeIf`, `clear`, `List.remove`, `Queue.remove`, `poll`, `BlockingQueue.take`, timed `poll`, `Deque.pop`, `removeFirst`, `removeLast`, `pollFirst`, `pollLast`, occurrence-removal methods, `Map.remove`, conditional `remove`, `Map.clear`, `NavigableMap.pollFirstEntry`, `pollLastEntry`, `Iterator.remove`, and legacy `Vector` removal methods
`@Replaceable`	`List.set`, `List.replaceAll`, `List.sort`, `Map.replace`, `Map.replaceAll`, `Map.Entry.setValue`, `ListIterator.set`, and mutating `Collections` algorithms such as `sort`, `reverse`, `shuffle`, `swap`, `fill`, `copy`, `rotate`, and `replaceAll`, plus legacy `Vector` replacement methods
Combined capabilities	`Map.put` and `putAll` require `@Growable @Replaceable`; `Map.computeIfPresent` requires `@Shrinkable @Replaceable`; `Map.compute` and `merge` require `@Growable @Shrinkable @Replaceable`; `BlockingQueue.drainTo` requires a `@Shrinkable` receiver and a `@Growable` target collection

An overriding method must preserve positive modifiability receiver requirements exactly. For example, if a superclass method has a `@Growable`, `@SeqGrowable`, `@Shrinkable`, or `@Replaceable` receiver, then the overriding method must require that same positive receiver capability; it may not drop or weaken the requirement.

22.2 Modifiability annotations

The Modifiability Checker uses **multiple independent 4-element hierarchies**, one per capability. Every collection type carries exactly one qualifier from each hierarchy. The checker also uses an additional 2-element hierarchy to express whether a collection's iterator preserves the collection's capabilities: whether `remove()` can be called on the iterator, and, for a `ListIterator`, whether `add()` and `set()` can be called.

Figure 22.1 shows the hierarchies. Each hierarchy is type-checked individually. An assignment `T x = y` is valid only if the compile-time type of `y` is a subtype of (or equal to) `T` in every hierarchy simultaneously. For example, `@Growable @MaybeSeqGrowable @MaybeShrinkable @MaybeReplaceable List` is a subtype of `@MaybeGrowable @MaybeSeqGrowable @MaybeShrinkable @MaybeReplaceable List` (because `@Growable` $\leq$ `@MaybeGrowable`), but it is *not* a subtype of `@Ungrowable @MaybeSeqGrowable @MaybeShrinkable @MaybeReplaceable List` (because `@Growable` and `@Ungrowable` are incomparable siblings in the Grow hierarchy).

22.2.1 Alias annotations

For convenience, the Modifiability Checker provides five alias annotations. They are not part of any hierarchy; the checker expands each one into its constituent qualifiers when it appears in source code.

`@Modifiable` Alias for `@Growable @SeqGrowable @Shrinkable @Replaceable` (but see Section 22.3 for exceptions). Calling any mutating operation on this collection will not throw `UnsupportedOperationException`.

`@Unmodifiable` Alias for `@Ungrowable @SeqUngrowable @Unshrinkable @Unreplaceable` (but see Section 22.3 for exceptions). The collection is definitely unmodifiable: calling any mutating operation always throws `UnsupportedOperationException`. The checker issues a warning at every invocation of a grow, seq-grow, shrink, or replace operation.

`@MaybeModifiable` Alias for `@MaybeGrowable @MaybeSeqGrowable @MaybeShrinkable @MaybeReplaceable`. The checker cannot determine the modifiability of the collection. The checker conservatively issues a warning at every invocation of a grow, seq-grow, shrink, or replace operation.

`@UnmodifiableParam` Syntactic sugar for `@MaybeModifiable`. It may only be written within a formal parameter type to indicate that the method does not modify the parameter. This annotation (and `@MaybeModifiable`) enable the programmer to distinguish between a data structure's capability to perform a modification and the programmer's intent about whether a modification should occur. `@UnmodifiableParam` may appear nested within a parameter type, such as in `List<@UnmodifiableParam List<String>>`.
`@UnmodifiableParam` represents *reference unmodifiability*: no modification may happen through the reference, but an alias may modify the value. By contrast, `@Unmodifiable` is object unmodifiability: no alias may modify the value.

`@PolyModifiable` Alias for `@PolyGrowable @PolySeqGrowable @PolyShrinkable @PolyReplaceable` (see Section 22.2.3).

Examples using alias annotations:

```
@Modifiable List<String> mod = new ArrayList<>();  // all four capabilities
@Unmodifiable List<String> unmod = List.of("a");   // no capabilities
@MaybeModifiable List<String> unknown = ...;  // capabilities are unknown at compile time
void foo(@UnmodifiableParam List<String> list) { ... }  // foo does not modify list
```

22.2.2 Qualifiers

The Grow hierarchy

`@MaybeGrowable` The top qualifier in the Grow hierarchy. The checker cannot determine whether this collection supports grow operations. Calling grow operations such as `add`, `addAll`, etc. on this collection *may* throw `UnsupportedOperationException`. This is the default qualifier for unannotated types in the Grow hierarchy.

`@Growable` Calling grow operations such as `add`, `addAll`, etc. on this collection *will not* throw `UnsupportedOperationException`.

`@Ungrowable` Calling grow operations such as `add`, `addAll`, etc. on this collection *will* throw `UnsupportedOperationException`.

`@BottomGrowable` The bottom qualifier in the Grow hierarchy. Programmers should rarely write it.

The SeqGrow hierarchy

@MaybeSeqGrowable The top qualifier in the SeqGrow hierarchy. The checker cannot determine whether this collection or map supports sequenced grow operations. Calling sequenced grow operations such as `addFirst`, `addLast`, `putFirst`, and `putLast` on this collection or map *may* throw `UnsupportedOperationException`. This is the default qualifier for unannotated types in the SeqGrow hierarchy.

@SeqGrowable Sequenced grow operations, such as `addFirst` and `addLast` for collections or `putFirst` and `putLast` for maps, *will not* throw `UnsupportedOperationException`.

@SeqUngrowable Sequenced grow operations, such as `addFirst` and `addLast` for collections or `putFirst` and `putLast` for maps, *will* throw `UnsupportedOperationException`. For example, `TreeSet` and `ConcurrentSkipListSet` support ordinary `add` but throw `UnsupportedOperationException` for explicit positional insertion with `addFirst` and `addLast`. Similarly, `TreeMap` and `ConcurrentSkipListMap` support ordinary `put` but throw `UnsupportedOperationException` for explicit positional updates with `putFirst` and `putLast`.

@BottomSeqGrowable The bottom qualifier in the SeqGrow hierarchy. Programmers should rarely write it.

The Shrink hierarchy

@MaybeShrinkable The top qualifier in the Shrink hierarchy. The checker cannot determine whether this collection supports shrink operations. Calling shrink operations such as `remove`, `clear`, etc. on this collection *may* throw `UnsupportedOperationException`. This is the default qualifier for unannotated types in the Shrink hierarchy.

@Shrinkable Calling shrink operations such as `remove`, `clear`, etc. on this collection *will not* throw `UnsupportedOperationException`.

@Unshrinkable Calling shrink operations such as `remove`, `clear`, etc. on this collection *will* throw `UnsupportedOperationException`.

@BottomShrinkable The bottom qualifier in the Shrink hierarchy. Programmers should rarely write it.

The Replace hierarchy

@MaybeReplaceable The top qualifier in the Replace hierarchy. The checker cannot determine whether this collection supports replace operations. Calling replace operations such as `set`, `replaceAll`, etc. on this collection *may* throw `UnsupportedOperationException`. This is the default qualifier for unannotated types in the Replace hierarchy.

@Replaceable Calling replace operations such as `set`, `replaceAll`, etc. on this collection *will not* throw `UnsupportedOperationException`.

@Unreplaceable Calling replace operations such as `set`, `replaceAll`, etc. on this collection *will* throw `UnsupportedOperationException`.

@BottomReplaceable The bottom qualifier in the Replace hierarchy. Programmers should rarely write it.

The Iterator hierarchy

@MaybeIteratorPolyMod The top qualifier. The checker cannot determine whether this collection's `iterator()` is `@PolyShrinkable`. This is the default qualifier for unannotated types.

@IteratorPolyMod This collection's `iterator()` is `@PolyShrinkable`. That is, if collection `c` is `@Shrinkable`, then `c.iterator()` is also `@Shrinkable`.

It might seem that it is sufficient to simply annotate `iterator()` directly, as in

```
class List {
  @MaybeShrinkable Iterator iterator() { ... }
}
class ArrayList {
  @PolyShrinkable Iterator iterator(@PolyShrinkable ArrayList this) { ... }
}
```

```
class CopyOnWriteArrayList {
  @Unshrinkable Iterator iterator(@PolyShrinkable CopyOnWriteArrayList this) { ... }
}
```

That approach would be sound but imprecise. In practice, many expressions of static type `List` evaluate to `ArrayList`s or other classes with `@PolyShrinkable iterator()` methods. The `@IteratorPolyMod` annotation is similar to writing the hypothetical annotation `@RuntimeType("ArrayList")`: it indicates a property that is more precise than the declared type.

22.2.3 Polymorphic qualifiers

A polymorphic qualifier (Section 32.2) specifies that a method's return type has the same qualifier (in some hierarchy) as a formal parameter type (possibly the receiver).

`@PolyGrowable` Polymorphic qualifier for the Grow hierarchy. The return type's growability matches the growability of whichever formal parameter is annotated with `@PolyGrowable`.

`@PolySeqGrowable` Polymorphic qualifier for the SeqGrow hierarchy. The return type's seq-growability matches the seq-growability of whichever formal parameter is annotated with `@PolySeqGrowable`.

`@PolyShrinkable` Polymorphic qualifier for the Shrink hierarchy. The return type's shrinkability qualifier matches the shrinkability of whichever formal parameter is annotated with `@PolyShrinkable`. Useful for methods such as `Map.keySet()` that preserve shrinkability while returning an `@Ungrowable` view.

`@PolyReplaceable` Polymorphic qualifier for the Replace hierarchy. The return type's replaceability matches the replaceability of whichever formal parameter is annotated with `@PolyReplaceable`.

`@PolyIteratorPolyMod` Polymorphic qualifier for the Iterator hierarchy. The return type's iterator-preservation qualifier matches the iterator-preservation qualifier of whichever formal parameter is annotated with `@PolyIteratorPolyMod`.

`@PolyModifiable` Alias for `@PolyGrowable @PolySeqGrowable @PolyShrinkable @PolyReplaceable`. Preserves all four capabilities simultaneously. Use on methods such as `Collections.synchronizedList` that do not change modifiability.

22.2.4 Modifiability method annotations

The Modifiability Checker supports method annotations that specify method behavior.

`@PreservesModifiability` Indicates that if the argument to the method is modifiable, then the returned collection is also modifiable. However, if the argument is unmodifiable, then the returned collection may or may not be unmodifiable. This is useful for methods that may return their argument, or may return a new modifiable collection. For example, if the argument is `@Growable`, then the returned collection is also `@Growable`. But if the argument is `@Ungrowable`, then the return collection is `@MaybeGrowable`. More generally, if the argument is `@Growable`, `@Shrinkable`, `@Replaceable`, `@SeqGrowable`, or `@IteratorPolyMod`, then the Modifiability Checker treats the return value as having that same capability. If the argument has any other qualifier, the return type is the top qualifier (`@MaybeGrowable`, `@MaybeSeqGrowable`, `@MaybeShrinkable`, `@MaybeReplaceable`, or `@MaybeIteratorPolyMod`) in the corresponding hierarchy.

This annotation may only be written on non-void, one-argument method declarations. The formal parameter may not be a varargs parameter, because the first argument of a call to a varargs method is an element of the varargs array rather than the sole formal parameter.

`@ThrowsUnsupportedOperation` Indicates that the method's implementation always throws `UnsupportedOperationException`. Write it on a skeletal implementation, such as one in `java.util.AbstractList`, that a subclass is expected to override.

A subclass that inherits the implementation, without overriding it, does not support the operation, even if the subclass declares that it does. The Modifiability Checker issues an error for such a subclass; see Section 22.2.5. Without the annotation, it could not, because the body of an inherited method is compiled separately and the checker sees only its signature.

The Modifiability Checker verifies the annotation on every method that it compiles: the body must be exactly `throw new UnsupportedOperationException(...)`.
Do not write the annotation on a method that throws `UnsupportedOperationException` only because some other method does, such as `AbstractList.add(E)`, whose body is `add(size(), e)`. A subclass that overrides the other method makes such a method work.

22.2.5 Inherited implementations

The Modifiability Checker requires the constructors of a class to agree with the class's method bodies: if the constructors are `@Ungrowable`, then every method with a `@Growable` receiver must throw `UnsupportedOperationException`, and if the constructors are `@Growable`, then no such method may.

A class does not escape the requirement by inheriting a method rather than declaring it. For example, this class does not support `add()`, because `AbstractList` implements `add(int, E)` by throwing `UnsupportedOperationException`:

```
// error: the inherited implementation of add(int, E) throws
class GrowableList extends AbstractList<String> {
  @Growable GrowableList() {}
  ...
}
```

The checker knows that an inherited implementation throws `UnsupportedOperationException` if the method is annotated `@ThrowsUnsupportedOperation`, or if the method's class declares the negative qualifier on its constructors, in which case the checker already verified that the method throws. The checker cannot tell for any other inherited method, so it says nothing about one.

22.3 `@Modifiable` and `@Unmodifiable` for types without grow, seq-grow, shrink, and/or replace methods

Ordinarily, the declaration `@Modifiable MyClass x;` is equivalent to `@Growable @SeqGrowable @Shrinkable @Replaceable MyClass x;`, and `@Unmodifiable MyClass x;` is equivalent to `@Ungrowable @SeqUngrowable @Unshrinkable @Unreplaceable MyClass x;`.

However, some types lack methods for one or more mutation capabilities. In such cases, the Modifiability Checker weakens the structurally unavailable component to the corresponding top qualifier. For example, `Iterator` has `remove` but not `add` or `set`, so `@Modifiable Iterator` expands to `@MaybeGrowable @MaybeSeqGrowable @Shrinkable @MaybeReplaceable Iterator`.

22.3.1 Alternative design: use negative capability rather than top qualifier

An alternative design would make `Modifiable Iterator` expand to `@Ungrowable @Shrinkable @Unreplaceable Iterator`. That would be incorrect, because not every `Iterator` is `@Ungrowable` and `@Unreplaceable`. For example, the static type `Iterator` does not declare `add` or `set` (they are structurally unavailable), but its subtype `ListIterator` and does declare those methods. Therefore `@Modifiable Iterator` should not mean that every iterator is incapable of grow and replace; it means only that those capabilities are not known from the static type `Iterator`.

Similarly, `Queue` does not declare replacement methods, but a variable with declared type `Queue` may contain a `LinkedList`, which does support replacement operations.

Using the top qualifier for structurally unavailable methods ensures that code like this type-checks:

```
@Replaceable LinkedList list = ...;
Queue q = list;
@Replaceable LinkedList list2 = (LinkedList) q; // still @Replaceable
```

22.3.2 Expansion rules for `@Modifiable` and `@Unmodifiable`

The following rules define this weakening. A capability is weakened to the corresponding `@Maybe*` qualifier when the type satisfies that capability's condition in the table below. For every other type, `@Modifiable` expands to the capability's positive qualifier, such as `@Growable`, and `@Unmodifiable` expands to the capability's negative qualifier, such as `@Ungrowable`.

Weakening conditions for `@Modifiable`/`@Unmodifiable`. "\" means set difference. A type includes its subtypes.

	Expands to `@Maybe*`
Grow	`Map.Entry` or (`Iterator`\`ListIterator`)
SeqGrow	not `SequencedCollection`, `SequencedMap`, or `Deque`
Shrink	`Map.Entry`
Replace	exact `Collection`, any `Set`, (`Queue`\`LinkedList`), or (`Iterator`\`ListIterator`)

These rules describe alias expansion, not the full specification of every JDK type. The JDK annotations may still give a more precise capability for a particular declaration. For instance, `SortedSet` is a `SequencedCollection` and `SortedMap` is a `SequencedMap`, so their seq-grow component is structurally relevant and is not weakened to `@MaybeSeqGrowable`. Their JDK declarations are annotated `@SeqUngrowable`, because their positional insertion methods such as `addFirst`, `addLast`, `putFirst`, and `putLast` always throw `UnsupportedOperationException`.

22.3.3 Example expansions for `@Modifiable` and `@Unmodifiable`

The following are representative canonical expansions.

```
      @Modifiable Collection   =      @Growable @MaybeSeqGrowable      @Shrinkable @MaybeReplaceable Collection
    @Unmodifiable Collection   =    @Ungrowable @MaybeSeqGrowable    @Unshrinkable @MaybeReplaceable Collection
      @Modifiable Queue        =      @Growable @MaybeSeqGrowable      @Shrinkable @MaybeReplaceable Queue
    @Unmodifiable Queue        =    @Ungrowable @MaybeSeqGrowable    @Unshrinkable @MaybeReplaceable Queue
      @Modifiable Deque        =      @Growable      @SeqGrowable      @Shrinkable @MaybeReplaceable Deque
    @Unmodifiable Deque        =    @Ungrowable    @SeqUngrowable    @Unshrinkable @MaybeReplaceable Deque
      @Modifiable Map          =      @Growable @MaybeSeqGrowable      @Shrinkable      @Replaceable Map
    @Unmodifiable Map          =    @Ungrowable @MaybeSeqGrowable    @Unshrinkable    @Unreplaceable Map
      @Modifiable SequencedMap =      @Growable      @SeqGrowable      @Shrinkable      @Replaceable SequencedMap
    @Unmodifiable SequencedMap =    @Ungrowable    @SeqUngrowable    @Unshrinkable    @Unreplaceable SequencedMap
      @Modifiable Set          =      @Growable @MaybeSeqGrowable      @Shrinkable @MaybeReplaceable Set
    @Unmodifiable Set          =    @Ungrowable @MaybeSeqGrowable    @Unshrinkable @MaybeReplaceable Set
      @Modifiable Iterator     = @MaybeGrowable @MaybeSeqGrowable      @Shrinkable @MaybeReplaceable Iterator
    @Unmodifiable Iterator     = @MaybeGrowable @MaybeSeqGrowable    @Unshrinkable @MaybeReplaceable Iterator
      @Modifiable ListIterator =      @Growable @MaybeSeqGrowable      @Shrinkable      @Replaceable ListIterator
    @Unmodifiable ListIterator =    @Ungrowable @MaybeSeqGrowable    @Unshrinkable    @Unreplaceable ListIterator
   Map.@Modifiable Entry       = @MaybeGrowable @MaybeSeqGrowable @MaybeShrinkable      @Replaceable Map.Entry
 Map.@Unmodifiable Entry       = @MaybeGrowable @MaybeSeqGrowable @MaybeShrinkable    @Unreplaceable Map.Entry
```

22.3.4 Expansion rules for `@PolyModifiable`

`@PolyModifiable` is weakened in only one case: on `Map.Entry`, its grow, seq-grow, and shrink components are the corresponding `@Maybe*` qualifiers, because no `Map.Entry` has methods for those capabilities. Every other type keeps all four polymorphic qualifiers, even a type such as `Iterator` or `Queue` whose `@Modifiable` expansion is weakened, because a variable of such a type may hold a value of a subtype, such as `ListIterator` or `LinkedList`, that does have the methods.

22.4 Behavior of the `iterator()` method

By default, the signature of `Collection.iterator()` is

```
@MaybeShrinkable Iterator iterator()
```

However, for some subclasses of `Collection`, the signature of `iterator()` is

```
@PolyShrinkable Iterator iterator(@PolyShrinkable MySubclassOfCollection this)
```

A collection uses the second signature if the collection is `@IteratorPolyMod`. The collection uses the first signature if the collection is `@MaybeIteratorPolyMod`.

Here are examples of its use:

```
@IteratorPolyMod @Shrinkable List<String> list = ...;
@Shrinkable Iterator<String> it = list.iterator();
it.remove();  // OK

@IteratorPolyMod @Unshrinkable List<String> list2 = ...;
@Unshrinkable Iterator<String> it2 = list2.iterator();
it2.remove();  // Error: it2 is @Unshrinkable

@Shrinkable List<String> list3 = ...;
@MaybeShrinkable Iterator<String> it3 = list3.iterator();
it3.remove();  // Error: list3 does not have @IteratorPolyMod
```

22.5 Examples of Modifiability Checker warnings

```
import java.util.*;
import org.checkerframework.checker.modifiability.qual.*;

class Demo {

  void basicUsage() {
    // @Modifiable is an alias for @Growable @SeqGrowable @Shrinkable @Replaceable
    @Modifiable List<String> mod = new ArrayList<>();
    mod.add("a");     // OK: @Growable guarantees add() works
    mod.addFirst("z"); // OK: @SeqGrowable guarantees addFirst() works
    mod.remove("a");  // OK: @Shrinkable guarantees remove() works
    mod.set(0, "b");  // OK: @Replaceable guarantees set() works

    // @Unmodifiable is an alias for @Ungrowable @SeqUngrowable @Unshrinkable @Unreplaceable
    @Unmodifiable List<String> unmod = List.of("a", "b");
    unmod.add("c");   // Error: add() requires @Growable, got @Ungrowable
    unmod.addFirst("d"); // Error: addFirst() requires @SeqGrowable, got @SeqUngrowable
    unmod.get(0);     // OK: read-only access needs no capability
  }

  void fineGrainedPermissions(@Growable List<String> g,
                              @Shrinkable List<String> s,
                              @Replaceable List<String> r) {
    // Growable list allows adding elements
    g.add("a");       // OK
    g.remove("a");    // Error: remove() requires @Shrinkable
    g.set(0, "b");    // Error: set() requires @Replaceable

    // Shrinkable list allows removing elements
    s.remove("a");    // OK
    s.add("a");       // Error: add() requires @Growable
```

```
    // Replaceable list allows updating elements
    r.set(0, "b");    // OK
    r.add("b");       // Error: add() requires @Growable
  }

  void combinedPermissions(@Growable @Replaceable Map<String, String> map) {
    // Map.put requires both Grow and Replace capabilities
    map.put("key", "value");  // OK

    // Map.remove requires Shrink capability
    map.remove("key");        // Error: @MaybeShrinkable (default) !<: @Shrinkable
  }

  // @PolyModifiable preserves all four capabilities
  @PolyModifiable List<String> wrap(@PolyModifiable List<String> list) {
    return list;
  }

  void testPoly(@Modifiable List<String> mod, @Growable @Shrinkable List<String> gs) {
    @Modifiable List<String> m = wrap(mod);            // OK
    @Growable @Shrinkable List<String> gs2 = wrap(gs); // OK
    // :: error: [assignment]
    @Modifiable List<String> bad = wrap(gs);  // Error: gs has @MaybeSeqGrowable and @MaybeReplaceable
  }
}
```

Chapter 23

Purity Checker

A *side-effect-free* method is one that has no externally-visible side effects: it changes no object that exists when it is called, except in ways that no caller can observe, such as updating a private cache. A *deterministic* method returns the same value each time it is called on the same arguments; equality is with respect to ==, not `.equals()`. A *pure* method is both side-effect-free and deterministic. The Purity Checker can check and infer all three types of methods, and "purity" refers to all concepts.

All checkers utilize purity annotations to aid type refinement (Section 33.7). By default, purity annotations are trusted, not checked.

To check purity annotations, pass `-AcheckPurityAnnotations` to any checker. The Purity Checker is conservative, so you are likely to need to suppress some warnings. To run *only* the Purity Checker (this is uncommon), supply the following command-line option to javac: `-processor org.checkerframework.framework.util.PurityChecker`. That implies `-AcheckPurityAnnotations`, so you do not need to supply both options.

To infer purity annotations, supply the command-line option `-AsuggestPureMethods`. The Checker Framework will suggest methods that can be marked as `@SideEffectFree`, `@Deterministic`, or `@Pure`. In addition, such suggestions are output by `-Ainfer` and when using whole-program inference.

23.1 Purity annotations

`@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`.

23.2 Lambdas, method references, and functional interfaces

Creating a lambda, or declaring a local or anonymous class, is not itself a side effect. A lambda expression evaluates to an object but does not run the code in the lambda's body. A side-effect-free method may therefore create a lambda whose body has side effects:

```
@SideEffectFree
Runnable makeRunnable() {
  return () -> count++;  // OK: creating the lambda has no side effect
}
```

The effects of the body occur where the functional method is invoked, and each such invocation is checked on its own.

Likewise, evaluating a method reference does not run the referenced method. However, the qualifier expression of a method reference *`expr`*`::m` is evaluated where the method reference appears, so its effects are effects of the enclosing method:

```
@SideEffectFree
Supplier<String> makeSupplier() {
  return sideEffect()::toString;  // error: sideEffect() is called here
}
```

The body of a lambda is checked against the purity annotations on the functional interface method that the lambda implements, not against those on the method that creates the lambda. For example:

```
@FunctionalInterface
interface PureSupplier {
  @SideEffectFree String get();
}

PureSupplier s1 = () -> "hello";                  // OK
PureSupplier s2 = () -> { count++; return ""; };  // error: the body has a side effect
```

Similarly, the method that a method reference refers to must be annotated at least as pure as the functional interface method:

```
@SideEffectFree String name() { ... }
String describe() { ... }

PureSupplier s3 = this::name;      // OK
PureSupplier s4 = this::describe;  // error: describe() is not @SideEffectFree
```

The functional interface method's purity annotations include those that it inherits from methods that it overrides (Section 23.4). If the functional interface method has no purity annotation — for example, because it is declared in an unannotated library and no stub file annotates it — then neither a lambda's body nor a method reference is checked against it.

The errors shown in this section are reported only when purity annotations are checked: by the Purity Checker, or by another checker when you pass `-AcheckPurityAnnotations`. With `-Ainfer`, the body of a lambda is not checked; instead, it is used to infer the purity of the functional interface method.

23.2.1 Functional-interface parameters of methods with purity annotations

A method annotated with a purity annotation may call the functional methods of its own functional-interface parameters. A functional-interface parameter is one whose type, or the interface that declares the type's functional method, is annotated with `@FunctionalInterface`, as `Function`, `Runnable`, and the other interfaces in `java.util.function` are. A parameter of a type such as `Iterable`, `AutoCloseable`, or `Comparable` is not one, even though the type has a single abstract method. A call to the functional method of such a parameter is treated as having the purity that the method promises:

- In a `@SideEffectFree` method, the call is treated as side-effect-free.
- In a `@Deterministic` method, the call is treated as deterministic.
- In a `@Pure` method, the call is treated as pure.

```
@SideEffectFree
int m(Function<String, Integer> f, String s) {
  return f.apply(s);  // permitted: apply is treated as side-effect-free
}
```

The assumption applies only when the parameter is effectively final and the functional method is called directly on the parameter. It does not apply to a call through a local variable or field that aliases the parameter, nor to a method of the parameter's type other than its functional method, such as `Function.andThen`:

```
@SideEffectFree
int m2(Function<String, Integer> f, String s) {
  Function<String, Integer> g = f;
  return g.apply(s);  // error: g is not a parameter
}
```

A call may reach the method through any method that it overrides, and such a call checks its arguments only against the overridden method's promise. So the method may assume only the purity that it and every method it overrides promise for that parameter. An overridden method that makes no promise, or whose parameter at that position is not a functional interface (for example, its type is a type variable), guarantees nothing, and the assumption does not apply.

At each call to such a method, every argument passed to a functional-interface parameter must have the purity that the method promises (side-effect-free for a `@SideEffectsOnly` method). The requirement is the same for every functional-interface parameter, and it applies even if the method's body never calls that parameter's functional method. Only a parameter whose declared type is a functional interface is checked: the method may assume nothing about an argument to a parameter whose type is a type variable, nor about the elements of a varargs array. How the argument is checked depends on its form:

- A method reference is checked via the referenced method, which must have the required purity annotations.
- A lambda is checked via its body, which must have the required purity.
- Any other argument is checked via the functional method of the argument's declared type, which must have the required purity annotations.

```
m(String::length, "hi");   // OK: String.length() is side-effect-free
m(s -> s.length(), "hi");  // OK: the lambda body is side-effect-free
m(s -> count++, "hi");     // error: the lambda body has a side effect

void caller(Function<String, Integer> g) {
  m(g, "hi");              // error: [purity.functional.argument] Function.apply is not @SideEffectFree
}
```

If the functional method returns `void`, a `@Pure` method requires only that the argument be side-effect-free, because a side-effect-free call that returns no value is deterministic. Otherwise a `@Pure` method requires the argument to be deterministic too, even if the method calls it only on an error path. For example, `Objects.requireNonNull(T, Supplier<String>)` is `@Pure`, so the following is an error, because `String.format` is not deterministic:

```
Objects.requireNonNull(x, () -> String.format("bad %s", y));  // error: [purity.call]
```

The assumption affects purity checking, and the dataflow analysis of a lambda in the method body, but not the dataflow analysis of the body itself. A direct call to a functional-interface parameter in the body still discards refinements of fields, as a call to any method that is not annotated as side-effect-free does:

```
@SideEffectFree
int m(Function<String, Integer> f, String s) {
  if (this.x != null) {
    f.apply(s);
    return this.x.hashCode();  // error: [dereference.of.nullable]
  }
  return 0;
}
```

23.3 Purity annotations are trusted

By default, purity annotations are trusted by every checker built with the Checker Framework. Purity annotations on called methods affect type-checking of client code, so an incorrect purity annotation can make type-checking unsound.

You can make a mistake by writing `@SideEffectFree` on the declaration of a method that is not side-effect-free, or by writing `@Deterministic` on the declaration of a method that is not deterministic.

Other than in the Purity Checker, checking of purity annotations (`-AcheckPurityAnnotations`) 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.

23.4 Overriding methods inherit specifications from overridden methods

A purity annotation is inherited: if a method in a superclass or superinterface has a purity annotation, then every overriding definition has that annotation too, even if the annotation is not written on the overriding definition. An overriding definition may strengthen the specification by writing a stronger annotation, but it cannot weaken it.

For example, the `Object` class is annotated as:

```
class Object {
  ...
  @Pure int hashCode() { ... }
}
```

(where `@Pure` means both `@SideEffectFree` and `@Deterministic`). Therefore, every definition of `hashCode`, including those in your program, is implicitly annotated as `@Pure` and its body is checked accordingly:

```
MyClass.java:1465: error: [purity.assign.field] field assignment not allowed in deterministic side-effect-free method
    this.hash = h;
        ^
```

You can fix the definition by making its body pure. Alternately, you can suppress the warning; see Chapter 34, page 222.

23.5 Suppressing warnings

The command-line options `-AassumeSideEffectFree`, `-AassumeDeterministic`, and `-AassumePure` make the Checker Framework unsoundly assume that *every* called method is side-effect-free, is deterministic, or is both, respectively.

The command-line option `-AassumePureGetters` makes the Checker Framework unsoundly assume that every getter method is side-effect-free and deterministic. For the purposes of `-AassumePureGetters`, a getter method is defined as an instance method with no formal parameters, whose name starts with "get", "is", "not", or "has" followed by an uppercase letter.

These options 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, these options can mask real errors. They are most appropriate when one of the following is true:

- You are starting out annotating a project.
- You are using the Checker Framework to find bugs but not to give a guarantee that no more errors exist of the given type.
- You are working with an unannotated library that makes the given assumptions, in which case (say) using `-AassumePureGetters` is easier than writing stub files for all the library's getter methods.

Chapter 24

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.
- Alternatively, you can run just the Constant Value Checker by supplying the following command-line option to javac: `-processor org.checkerframework.common.value.ValueChecker`

24.1 Annotations

The Constant Value Checker uses type annotations to indicate the value of an expression (Section 24.1.1), and it uses method annotations to indicate methods that the Constant Value Checker can execute at compile time (Section 24.2.2).

24.1.1 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`, `@StringVal`, `@MatchesRegex`, `@DoesNotMatchRegex`, and `@EnumVal`. Additional type annotations for arrays and strings are `@ArrayLen`, `@ArrayLenRange`, and `@MinLen`. A polymorphic qualifier (`@PolyValue`) is also supported (see Section 32.2). In addition, there are separate checkers for `@ClassVal` and `@MethodVal` annotations (see Section 26.2).

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 have more than 10 different values, the Constant Value Checker gives up and its type becomes `@IntRange` for integral types, `@ArrayLenRange` for array types, `@MatchesRegex`, `@DoesNotMatchRegex`, `@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`, `@MatchesRegex`, and `@DoesNotMatchRegex` annotations may be applied to char arrays. Although byte arrays are often converted to/from strings, these annotations may not be applied to them. This is because the conversion depends on the platform's character set.

The `@MatchesRegex` and `@DoesNotMatchRegex` annotations use the standard Java regular expression syntax. `@MatchesRegex(A)` is only a subtype of `@MatchesRegex(B)` if the set of regular expressions `A` is a subset of the set of

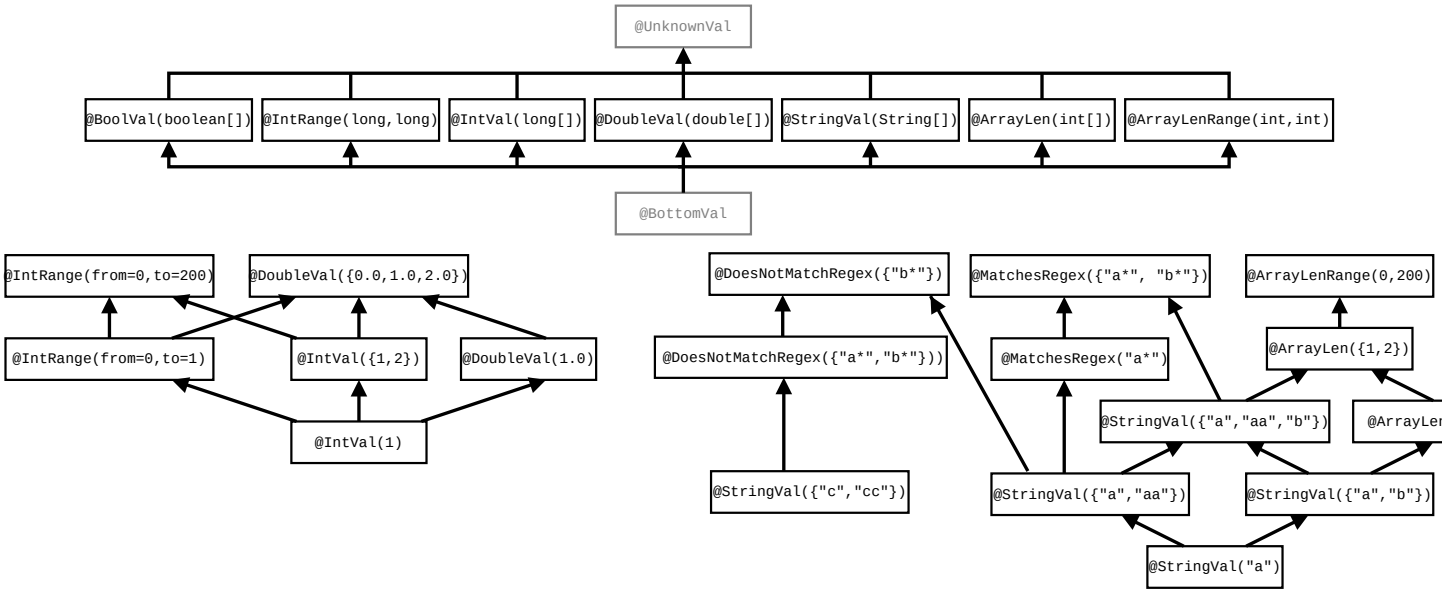


Figure 24.1: At the top, the type qualifier hierarchy of the Constant Value Checker annotations. The first four qualifiers are applicable to primitives and their wrappers; the next to `String`s (it can also be written as `@EnumVal` for enumeration constants), and the final two to arrays. Qualifiers in 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.

regular expressions `B`. An `@StringVal` annotation is a subtype of a `@MatchesRegex` annotation if each string matches at least one of the regular expressions. `@DoesNotMatchRegex(A)` is only a subtype of `@DoesNotMatchRegex(B)` if the set of regular expressions `A` is a superset of the set of regular expressions `B`. Matching is done via the java.lang.String#matches method, which matches against the entire string (it does not look for a matching substring).

The `@EnumVal` annotation's argument is the names of the enum constants that the type might evaluate to. (Java syntax does not allow the enum constants themselves to be arguments.) `@EnumVal` is treated identically to `@StringVal` by the checker internally, so `@StringVal` may appear in error messages related to enums.

`@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` annotation and an `@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 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 24.1 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 cast to `double`, an `@IntVal` annotation is a subtype of a `@DoubleVal` annotation with the same values.

Figure 24.2 illustrates how the Constant Value Checker infers type annotations (using flow-sensitive type qualifier refinement, Section 33.7).

24.2 Other constant value annotations

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

If your code is already annotated with a different constant value or range annotation, the Checker Framework can

```
public void flowSensitivityExample(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 24.2: The Constant Value Checker infers different types for a variable on different lines of the program.

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

Figure 24.3: Correspondence between other constant value and range annotations and the Checker Framework's annotations.

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 24.3. If the other annotation is a declaration annotation, it may be moved; see Section 40.6.10.

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.

24.2.1 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 (at compile time) operators that do not throw exceptions (e.g., `+`, `-`, `<<`, `!=`).

24.2.2 `@StaticallyExecutable` methods and the classpath

The Constant Value Checker statically executes (at compile time) methods annotated with `@StaticallyExecutable`.

The static execution feature has some requirements:

- A `@StaticallyExecutable` method must be `@Pure` (side-effect-free and deterministic).

```
@StaticallyExecutable @Pure
public static int myAdd(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 = myAdd(a, b);  // c has type:  @IntVal({9}) int
}
```

Figure 24.4: The `@StaticallyExecutable` annotation enables constant propagation through method calls.

- The Constant Value Checker must have an estimate for all the arguments at a call site.
  This means that `@StaticallyExecutable` is not applicable to user-written instance methods. It is only applicable to instance methods whose receiver is a compile-time constant, such as a primitive wrapper or an array.
- The `@StaticallyExecutable` method and any method it calls must be on the same path (the classpath or the processorpath) as the Checker Framework. This is because the Constant Value Checker reflectively calls these methods at compile time.
  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 or processorpath, whichever is appropriate. For example, the command-line arguments to the Checker Framework might include:

  ```
  -processor org.checkerframework.common.value.ValueChecker
  -classpath $CLASSPATH:MY_PROJECT/build/
  ```

  or

  ```
  -processor org.checkerframework.common.value.ValueChecker
  -processorpath ${CHECKERFRAMEWORK}/checker/build/libs/checker-4.2.1-SNAPSHOT.jar:MY_PROJECT/build/
  ```

24.3 Warnings

If the option `-AreportEvalWarns` 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 resolved to a specific value annotation. Here are some examples:

- `[class.find.failed] Failed to find class named Test.`
  The checker could not find the class specified for resolving a `@StaticallyExecutable` method. Typically this means that the path that contains the Checker Framework (the classpath or the processorpath) lacks the given classfile.
- `[method.find.failed] Failed to find a method named foo with argument types [@IntVal(3) int].`
  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 classfile 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 (at compile time) 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`.

24.4 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 24.5 gives an example of behavior with and without the `-AignoreRangeOverflow` command-line option.

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

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.

24.5 Strings can be null in concatenations

By default, the Constant Value Checker is sound with respect to string concatenation and nullness. It assumes that, in a string concatenation, every non-primitive argument might be null, except for String literals and compile-time constants. It ignores Nullness Checker annotations. (This behavior is conservative but sound.)

Consider a variable declared as `@StringVal("a", "b") String x;`. At run time, `x` evaluates to one of the values `"a"`, `"b"`, or `null`. Therefore, the type of "`x + "c"`" is `@StringVal("ac", "bc", "nullc") String`.

The `-AnonNullStringsConcatenation` command-line option makes the Constant Value Checker unsoundly assume that no arguments in a string concatenation are null. With the command-line argument, the type of "`x + "c"`" is `@StringVal("ac", "bc") String`.

Chapter 25

Returns Receiver Checker

The Returns Receiver Checker enables documenting and checking that a method returns its receiver (i.e., the `this` parameter).

There are two ways to run the Returns Receiver Checker.

- Typically, it is automatically run by another checker.
  If the code being checked does not use fluent APIs, you can pass the `-AdisableReturnsReceiver` command-line option. This disables the Returns Receiver Checker and makes the other checker run faster.
- Alternatively, you can run just the Returns Receiver Checker by supplying the following command-line option to javac: `-processor org.checkerframework.common.returnsreceiver.ReturnsReceiverChecker`

25.1 Annotations

The qualifier `@This` on the return type of a method indicates that the method returns its receiver. Methods that return their receiver are common in so-called "fluent" APIs. Here is an example:

```
class MyBuilder {
  @This MyBuilder setName(String name) {
    this.name = name;
    return this;
  }
}
```

A `@This` annotation can only be written on a return type, on a receiver type, or in a downcast.

As is standard, the Returns Receiver Checker has a top qualifier, `@UnknownThis`, and a bottom qualifier, `@BottomThis`. Programmers rarely need to write these annotations.

Here are additional details. `@This` is a polymorphic qualifier rather than a regular type qualifier (see Section 32.2). Conceptually, a receiver type always has a `@This` qualifier. When a method return type also has a `@This` qualifier, the presence of the polymorphic annotation on both the method's return and receiver type forces their type qualifiers to be *equal*. Hence, the method will only pass the type-checker if it returns its receiver argument, achieving the desired checking.

25.2 AutoValue and Lombok support

The AutoValue and Lombok projects both support automatic generation of builder classes, which enable flexible object construction. For code using these two frameworks, the Returns Receiver Checker automatically adds `@This` annotations to setter methods in builder classes. All the `@This` annotations in Figures 25.1 and 25.2 are automatically added by the Returns Receiver Checker.

```
@AutoValue
abstract class Animal {
  abstract String name();
  abstract int numberOfLegs();
  static Builder builder() {
    return new AutoValue_Animal.Builder();
  }
  @AutoValue.Builder
  abstract static class Builder {
    abstract Builder setName(String value);       // @This is automatically added here
    abstract Builder setNumberOfLegs(int value);  // @This is automatically added here
    abstract Animal build();
  }
}
```

Figure 25.1: User-written code that uses the `@AutoValue.Builder` annotation. Given this code, (1) AutoValue automatically generates a concrete subclass of `Animal.Builder`, see Figure 25.2, and (2) the Returns Receiver Checker automatically adds `@This` annotations on setters in both user-written and automatically-generated code.

```
class AutoValue_Animal {
  static final class Builder extends Animal.Builder {
    private String name;
    private Integer numberOfLegs;
    @This Animal.Builder setName(String name) {
      this.name = name;
      return this;
    }
    @This Animal.Builder setNumberOfLegs(int numberOfLegs) {
      this.numberOfLegs = numberOfLegs;
      return this;
    }
    @Override
    Animal build() {
      return new AutoValue_Animal(this.name, this.numberOfLegs);
    }
  }
}
```

Figure 25.2: Code generated by AutoValue for the example of Figure 25.1, including the `@This` annotations added by the Returns Receiver Checker.

Chapter 26

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 run time. When type-checking the call, the Checker Framework uses the library annotations for the possibly-invoked methods, rather than the imprecise one for `Method.invoke`.

If the estimate of invoked methods is small, the checker issues fewer false positive warnings. If the estimate of invoked methods is large, the resulting types may be 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 classpath 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, which may be useful for debugging.

Section 26.1 gives an example of reflection resolution. Then, Section 26.2 describes the MethodVal and ClassVal Checkers, which reflection resolution uses internally. The paper "Static analysis of implicit control flow: Resolving Java reflection and Android intents" [BJM+15] (ASE 2015, `https://homes.cs.washington.edu/~mernst/pubs/implicit-control-flow-ase2015-abstract.html`) gives further details.

26.1 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.

26.2 MethodVal and ClassVal Checkers

The implementation of reflection resolution internally uses the ClassVal Checker (Section 26.2.1) and the MethodVal Checker (Section 26.2.2). They are similar to the Constant Value Checker (Chapter 24) 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 26.2.3).

26.2.1 ClassVal Checker

The ClassVal Checker defines the following annotations:

`@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.
Each argument is a "fully-qualified binary name" (`@FqBinaryName`): a primitive or binary name (JLS §13.1), possibly followed by array brackets.

`@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.
Each argument is a "fully-qualified binary name" (`@FqBinaryName`): a primitive or binary name (JLS §13.1), possibly followed by array brackets.

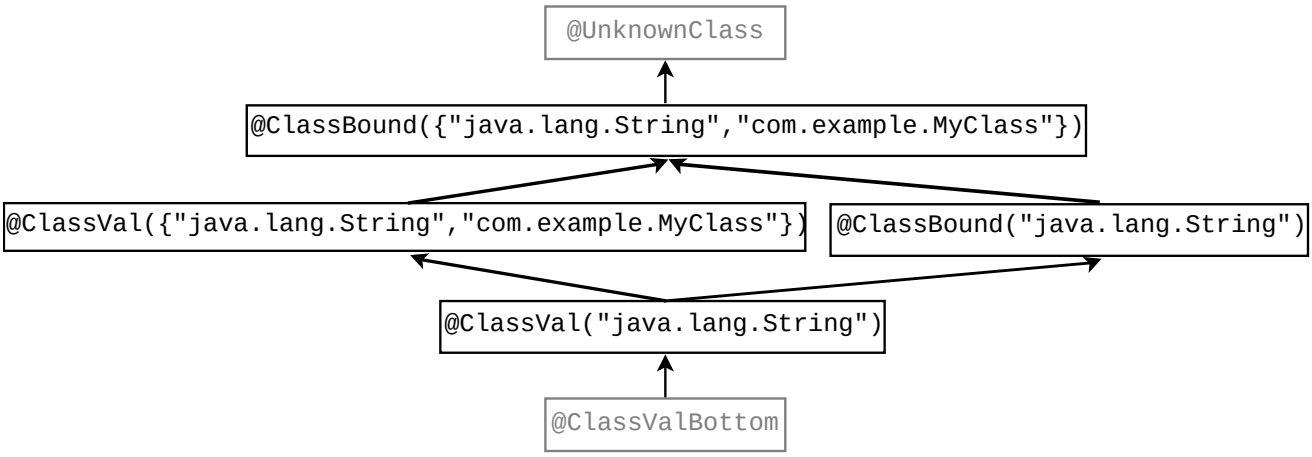


Figure 26.1: Partial type hierarchy for the ClassVal type system. The type qualifiers in gray (`@UnknownClass` and `@ClassValBottom`) should never be written in source code; they are used internally by the type system.

`@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 26.1 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.

26.2.2 MethodVal Checker

The MethodVal Checker defines the following annotations:

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

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

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

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

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

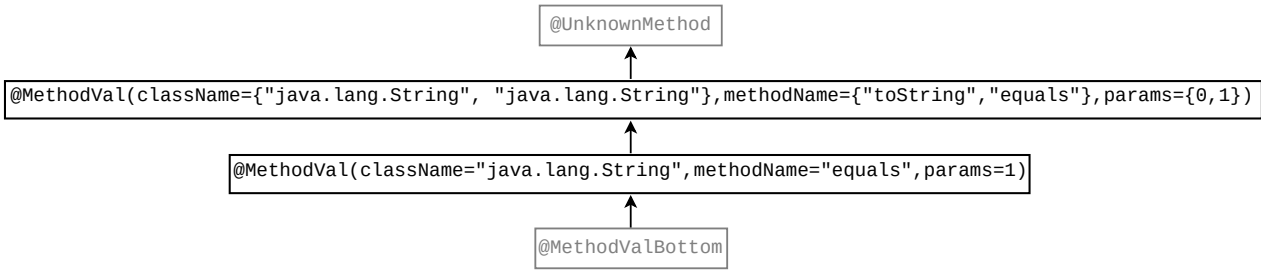


Figure 26.2: Partial type hierarchy for the MethodVal type system. The type qualifiers in gray (`@UnknownMethod` and `@MethodValBottom`) should never be written in source code; they are used internally by the type system.

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

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

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

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

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

Figure 26.3: Example inference rules for @ClassVal, @ClassBound, and @MethodVal. Additional rules exist for expressions with similar semantics but that call methods with different names or signatures.

Subtyping rules

Figure 26.2 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

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

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

26.2.3 MethodVal and ClassVal inference

The developer rarely has to write `@ClassVal` or `@MethodVal` annotations, because the Checker Framework infers them according to Figure 26.3. 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.

Chapter 27

Initialized Fields Checker

The Initialized Fields Checker warns if a constructor does not initialize a field.

27.1 Running the Initialized Fields Checker

An example invocation is

```
javac -processor org.checkerframework.common.initializedfields.InitializedFieldsChecker MyFile.java
```

If you run it together with other checkers, then it issues warnings only if the default value assigned by Java (0, false, or null) is not consistent with the field's annotation, for the other checkers. An example invocation is

```
javac -processor ValueChecker,InitializedFieldsChecker MyFile.java
```

27.2 Motivation: uninitialized fields

Without the Initialized Fields Checker, every type system is unsound with respect to fields that are never set. (Exception: The Nullness Checker (Chapter 3, page 31) is sound. Also, a type system is sound if every annotation is consistent with 0, false, and null.) Consider the following code:

```
import org.checkerframework.checker.index.qual.Positive;

class MyClass {
  @Positive int x;
  MyClass() {
    // empty body
  }

  @Positive int getX() {
    return x;
  }
}
```

Method `getX` is incorrect because it returns 0, which is not positive. However, the code type-checks because there is never an assignment to `x` whose right-hand side is not positive. If you run the Index Checker together with the Initialized Fields Checker, then the code correctly does not type-check.

Remaining unsoundness

Even with the Initialized Fields Checker, every type system (except the Nullness Checker, Chapter 3, page 31) is unsound with respect to partially-initialized fields. Consider the following code:

```
import org.checkerframework.checker.index.qual.Positive;

class MyClass {
  @Positive int x;
  MyClass() {
    foo();
    x = 1;
  }

  @Positive int foo() {
    // ... use x, expecting it to be positive ...
  }
}
```

Within method `foo`, `x` can have the value 0 even though the type of `x` is `@Positive int`.

27.3 Example

As an example, consider the following code:

```
import org.checkerframework.checker.index.qual.Positive;

class MyClass {

  @Positive int x;
  @Positive int y;
  int z;

  // Warning: field y is not initialized
  MyClass() {
    x = 1;
  }
}
```

When run by itself, the Initialized Fields Checker warns that fields `y` and `z` are not set.

When run together with the Index Checker, the Initialized Fields Checker warns that field `y` is not set. It does not warn about field `z`, because its default value (0) is consistent with its annotations.

27.4 Annotations

The Initialized Fields type system uses the following type annotations:

`@InitializedFields` indicates which fields have definitely been initialized so far.
`@InitializedFieldsBottom` is the type of `null`. Programmers rarely write this type.
`@PolyInitializedFields` is a qualifier that is polymorphic over field initialization. For a description of qualifier polymorphism, see Section 32.2.

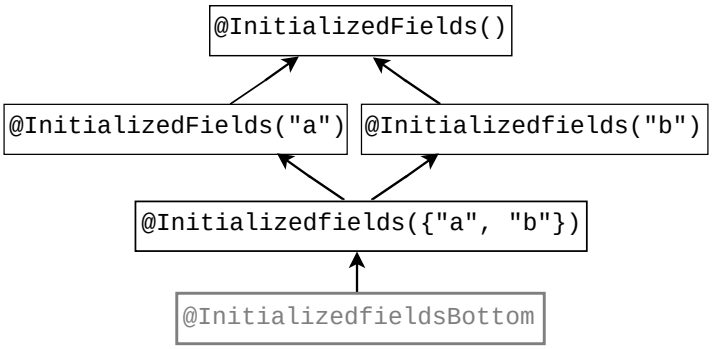


Figure 27.1: The type qualifier hierarchy of the Initialized Fields Checker. `@InitializedFieldsBottom` is rarely written by a programmer.

Figure 27.1 shows the subtyping relationships among the type qualifiers.
There is also a method declaration annotation:

`@EnsuresInitializedFields` indicates which fields the method sets. Use this for helper methods that are called from a constructor.

27.5 Comparison to the Initialization Checker

The Initialized Fields Checker is a lightweight version of the Initialization Checker (Section 3.8). Here is a comparison between them.

	Initialization Checker	Initialized Fields Checker
superclasses	tracks initialization of supertype fields	checks one class at a time
partial initialization	changes the types of fields that are not initialized	unsound treatment of partially-initialized objects (*)
type systems	works only with the Nullness Checker (**)	works for any type system
disabling	always runs with the Nullness Checker	can be enabled/disabled per run

* See Section 27.2 for an example.

** The Initialization Checker could be made to work with any type system, but doing so would require changing the implementation of both the type system and the Initialization Checker.

Chapter 28

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 31.20) 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.

28.1 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 create 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`):

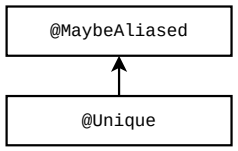


Figure 28.1: Type hierarchy for the Aliasing type system. These qualifiers are applicable to any reference (non-primitive) type.

@NonLeaked identifies a formal parameter that is neither 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 constructor 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.

28.2 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 28.4.)

```
@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 mightDoAnything(Object o) { ... }

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

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

28.3 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 makesAlias() {
        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>`.

28.4 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 33.7.

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 therefore 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 therefore have type @MaybeAliased.
}
```

Figure 28.2: Example of the 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, these are only method/constructor calls with a `@Unique` return type. A variable/field is not fresh because it can produce 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 28.2 gives an example of the Aliasing Checker's type refinement rules.

Chapter 29

Must Call Checker

The Must Call Checker conservatively over-approximates the set of methods that an object should call before it is de-allocated. The checker does not enforce any rules other than subtyping; in particular, it does not enforce that the methods are called before objects are de-allocated. The Must Call Checker is intended to be run as a subchecker of another checker. The primary client of the Must Call Checker is the Resource Leak Checker (Chapter 8, page 73), which enforces that every method in a must-call obligation for an expression is called before that expression is de-allocated.

For example, consider a `java.io.OutputStream`. This `OutputStream` might have an obligation to call the `close()` method, to release an underlying file resource. The type of this `OutputStream` is `@MustCall({"close"})`. Or, the `OutputStream` might not have such an obligation, if the underlying resource is a byte array. The type of this `OutputStream` is `@MustCall({})`. For an arbitrary `OutputStream`, the Must Call Checker over-approximates the methods that it should call by assigning it the type `@MustCall({"close"}) OutputStream`, which can be read as "an OutputStream that might need to call `close()` (but no other methods) before it is de-allocated".

If you are running the Resource Leak Checker (Chapter 8, page 73), then there is no need to run the Must Call Checker, because the Resource Leak Checker does so automatically. Running both may cause warning suppressions for the Must Call Checker not to work.

29.1 Must Call annotations

The Must Call Checker supports these type qualifiers:

`@MustCall(String[] value)` gives a superset of the methods that must be called before the expression's value is de-allocated. The annotation is not refined after a method is called; an expression of type `@MustCall("foo")` might or might not have already had `foo()` called on it. The default type qualifier for an unannotated type is `@MustCall({})`.

`@MustCallUnknown` represents a value with an unknown or infinite set of must-call obligations. It is used internally by the type system but should never be written by a programmer.

`@PolyMustCall` is the polymorphic annotation for the Must Call type system. For a description of polymorphic qualifiers, see Section 32.2.

`@InheritableMustCall(String[] value)` is an alias of `@MustCall` that can only be written on a class declaration. It applies both to the class declaration on which it is written and also to all subclasses. This frees the user of the need to write the `@MustCall` annotation on every subclass. See Section 29.2 for details.

Here are some facts about the type qualifier hierarchy, which is shown in Figure 29.1. Any expression of type `@MustCall({}) Object` also has type `@MustCall({"foo"}) Object`. The type `@MustCall({"foo"}) Object` contains objects that need to call `foo` and objects that need to call nothing, but the type does not contain an object that needs to call `bar` (or both `foo` and `bar`). `@MustCall({"foo", "bar"}) Object` represents all objects that need to call `foo`, `bar`, both, or neither; it cannot represent an object that needs to call some other method.

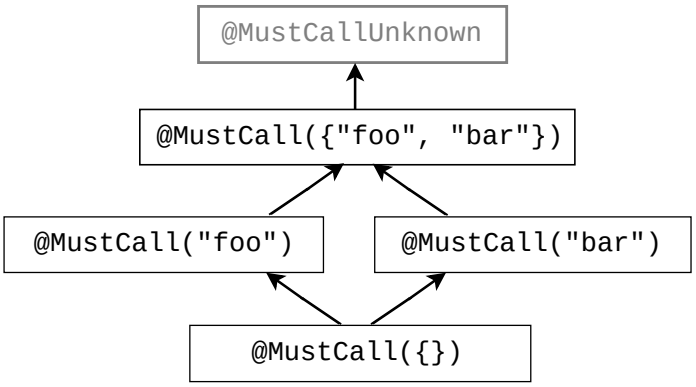


Figure 29.1: Part of the Must Call Checker's type qualifier hierarchy. The full hierarchy contains one `@MustCall` annotation for every combination of methods. Qualifiers in gray are used internally by the type system but should never be written by a programmer.

29.2 Writing `@MustCall`/`@InheritableMustCall` on a class

As explained in Section 33.3, a type annotation on a type declaration means that every use of the type has that annotation by default. If a class `A`'s declaration has a `@MustCall` annotation, then `A`'s subclasses usually should too. To do so, a programmer can either write a `@MustCall` annotation on every subclass, or can write `@InheritableMustCall` on only `A`, which will cause the checker to treat every subclass as if it has an identical `@MustCall` annotation. The latter is the preferred style.

For example, given the following class annotation:

```
package java.net;

import org.checkerframework.checker.mustcall.qual.InheritableMustCall;

@InheritableMustCall({"close"}) class Socket { }
```

any use of `Socket` or a subclass of `Socket` defaults to `@MustCall({"close"}) Socket`.

The `@InheritableMustCall` annotation is necessary because type annotations cannot be made inheritable. `@InheritableMustCall` is a declaration annotation.

29.3 Relationship to lightweight ownership

The Must Call Checker interacts with the `@Owning` and `@NotOwning` annotations of the Resource Leak Checker (Chapter 8, page 73). In particular, any location that the Resource Leak Checker would consider "non-owning" (e.g., unannotated parameters and `@NotOwning` returns) is treated as `@MustCall({})` by the Must Call Checker. This can lead to surprising results when running the Must Call Checker alone (which is not recommended). For example, consider the following program:

```
@MustCall({"toString"}) String foo(@MustCall({"hashCode"}) String arg) {
  return arg;
}
```

The Must Call Checker will not issue a warning about this code, because `arg` is non-owning (and therefore treated as `@MustCall({})`).

29.4 Assumptions about reflection

The Must Call Checker is unsound with respect to reflection: it assumes that objects instantiated reflectively do not have must-call obligations (i.e., that their type is `@MustCall({})`). If the checker is run with `-AresolveReflection`, then this assumption applies only to types that cannot be resolved.

29.5 Type parameter bounds often need to be annotated

In an unannotated program, there may be mismatches between defaulted type qualifiers that lead to type-checking errors. That is, the defaulted annotations at two different locations may be different and in conflict. A specific example is in code that contains a mix of explicit upper bounds with an `extends` clause and implicit upper bounds without an `extends` clause.

For example, consider the following example (from plume-util) of an interface with explicit upper bounds and a client with implicit upper bounds:

```
interface Partitioner<ELEMENT extends @Nullable Object, CLASS extends @Nullable Object> {}

class MultiRandSelector<T> {
  private Partitioner<T, T> eq;
}
```

The code contains no `@MustCall` annotations. Running the Must Call Checker on this code produces an error at each use of `T` in `MultiRandSelector`:

```
error: [type.argument] incompatible type argument for type parameter ELEMENT of Partitioner.
       private Partitioner<T, T> eq;
                          ^
  found   : T extends @MustCallUnknown Object
  required: @MustCall Object
error: [type.argument] incompatible type argument for type parameter CLASS of Partitioner.
       private Partitioner<T, T> eq;
                             ^
  found   : T extends @MustCallUnknown Object
  required: @MustCall Object
```

The defaulted Must Call annotations differ. `Partitioner` has an explicit bound, which uses the usual default of `@MustCall({})`. `MultiRandSelector` has an implicit bound, which defaults to top (that is, `@MustCallUnknown`), as explained in Section 33.5.3.

In many cases, including this one, you can eliminate the false positive warning without writing any `@MustCall` annotations. You can either:

- add an explicit bound to `MultiRandSelector`, changing its declaration to `class MultiRandSelector<T extends Object>`, or
- remove the explicit bound from `Partitioner`, changing its declaration to `interface Partitioner<ELEMENT, CLASS>`.

Chapter 30

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 37). This includes type introduction rules via the `@QualifierForLiterals` meta-annotation, and other features such as type refinement (Section 33.7) and qualifier polymorphism (Section 32.2).

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. For more details (including a definition of "typestate"), see Section 31.30. See Section 40.7.1 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 37.

30.1 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. You must provide one or both of the two following command-line arguments:

- 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 ...
  ```
- 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, if the Checker Framework is on the classpath rather than the processorpath:
  ```
  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
  ```

If the Checker Framework is on the processorpath, place the annotations on the processorpath instead of on the classpath.

30.1.1 Compiling your qualifiers and your project

The annotations listed in `-Aquals` or `-AqualDirs` must be accessible to the compiler during compilation. Before you run the Subtyping Checker with `javac`, they must be compiled and on the same path (the classpath or processorpath) as the Checker Framework. It is not sufficient to supply their source files on the command line.

30.1.2 Suppressing warnings from the Subtyping Checker

When suppressing a warning issued by the Subtyping Checker, as the "checker name" you may use the unqualified, uncapitalized name of any of the annotations passed to `-Aquals`. (See Section 34.1.1 for details about the `@SuppressWarnings` syntax.) As a matter of style, you should choose one of the annotations as the "checker name" part of the `@SuppressWarnings` string and use it consistently; this avoids confusion and makes it easier to search for uses.

30.2 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 these steps.

(This example also appears in the Checker Framework Tutorial, in more detail.)

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

```
package myPackage.qual;

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

/**
 * Denotes that the representation of an object is encrypted.
 */
@SubtypeOf(PossiblyUnencrypted.class)
@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 37.5.1.
Don't forget to compile these classes:

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

The resulting `.class` files should be on the same path (the classpath or processor path) as the Checker Framework.

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
      -Aquals=myPackage.qual.Encrypted,myPackage.qual.PossiblyUnencrypted YourProgram.jav

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
```

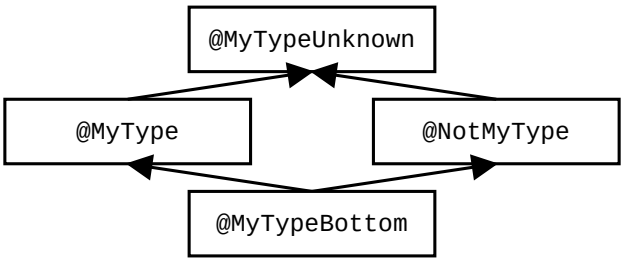


Figure 30.1: Type system for a type alias or typedef type system. The type system designer may choose to omit some of these types, but this is the general case. The type system designer's choice of defaults affects the interpretation of unannotated code, which affects the guarantees given for unannotated code.

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

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

30.3 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 20, page 140), the Fake Enum Checker (Chapter 9, page 83), 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 when changing the Java type. Therefore, it is best to avoid the pseudo-typedef antipattern.

The Units Checker (Chapter 20, page 140) is useful for the particular case of units of measurement, such as kilometers versus miles.

The Fake Enum Checker (Chapter 9, page 83) 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 the same effect 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 30.1.

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 type-check:

```
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 32.1.2), 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 31

Third-party checkers

The Checker Framework has been used to build other checkers that are not distributed together with the framework. This chapter gives an *incomplete* list of them. (If you know of others, or if you want this chapter to reference your checker, please send us a link and a short description.) Many of the publications in Section 41.7 provide implementations, which are not necessarily listed here yet.

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

This list is in reverse chronological order; newer tools appear first and older ones appear last.

31.1 Determinism checker

The Determinism Checker (`https://github.com/t-rasmud/checker-framework/tree/nondet-checker`) ensures that a program is deterministic across executions. A deterministic program is easier to test and to debug (for example, by comparing executions).

The Determinism Checker focuses on sequential programs. It detects nondeterminism due to the iteration order of a hash table (or on any other property of hash codes), default formatting (such as Java's `Object.toString()`, which includes a memory address), `random`, date-and-time functions, and accessing system properties such as the file system or environment variables.

The paper "Verifying determinism in sequential programs" [MWME21] describes the Determinism Checker.

31.2 Constant Value Inference (Interval Inference)

Interval analysis estimates the run-time values of numerical expressions in the source code by computing a lower bound and an upper bound. The Constant Value Inference (`https://github.com/d367wang/value-inference/tree/artifact`) project is a whole-program inference approach for integral range analysis (interval analysis).

The thesis "Interval Type Inference: Improvements and Evaluations" [Wan21] describes Constant Value Inference.

31.3 Crypto Checker

The Crypto Checker (`https://github.com/vehiloco/crypto-checker`) can help you find whether your program uses any weak or unsupported crypto algorithms or unsupported algorithm providers. If the Crypto Checker issues no warnings for a given program, then you have a guarantee that your program at run time will never have these issues. This is similar to the earlier AWS Crypto Policy Compliance Checker (Section 31.4).

The paper "Ensuring correct cryptographic algorithm and provider usage at compile time" [XCD21] (FTfJP 2021) and the thesis "Light-weight verification of cryptographic API usage" [Xin20] describe the Crypto Checker.

31.4 AWS crypto policy compliance checker

The AWS crypto policy compliance checker (`https://github.com/awslabs/aws-crypto-policy-compliance-checker`) checks that no weak cipher algorithms are used with the Java crypto API.

The paper "Continuous Compliance" [KSTE20] (ASE 2020) describes several custom Checker Framework checkers in the context of a compliance regime at Amazon Web Services.

31.5 AWS KMS compliance checker

The AWS KMS compliance checker (`https://github.com/awslabs/aws-kms-compliance-checker`) extends the Constant Value Checker (see Chapter 24, page 161) to enforce that calls to Amazon Web Services' Key Management Service only request 256-bit (or longer) data keys. This checker can be used to help enforce a compliance requirement (such as from SOC or PCI-DSS) that customer data is always encrypted with 256-bit keys.

The KMS compliance checker is available in Maven Central. To use it in `build.gradle`, add the following dependency:

```
compile("software.amazon.checkerframework:aws-kms-compliance-checker:1.0.2")
```

Other build systems are similar.

The paper "Continuous Compliance" [KSTE20] (ASE 2020) describes several custom Checker Framework checkers in the context of a compliance regime at Amazon Web Services.

31.6 PUnits units of measurement

The PUnits system (`https://github.com/opprop/units-inference`) can check the correctness of a program or annotate a program with units of measurement. It is described in the paper "Precise inference of expressive units of measurement types" (OOPSLA 2020) [XLD20].

31.7 JaTyC typestate checker

JaTyC, Java Typestate Checker (`https://github.com/jdmota/java-typestate-checker`), ensures that methods are called in the correct order. The sequences of method calls allowed are specified in a protocol file which is associated with a Java class by adding a `@Typestate` annotation to the class. It is described in "Behavioural Types for Memory and Method Safety in a Core Object-Oriented Language" [BFG+20], and the implementation is an extension of Mungo (`https://www.dcs.gla.ac.uk/research/mungo/index.html`).

31.8 NullAway

NullAway (`https://github.com/uber/NullAway`) is a fast type-checker for only nullness properties. It is built on top of Error Prone and the Checker Framework's Dataflow Framework. See Section 40.10.3 for a comparison between NullAway and the Nullness Checker. NullAway is described in the paper "NullAway: Practical type-based null safety for Java" [BCS19].

31.9 Nullness Rawness Checker

The Nullness Rawness Checker is a variant of the Nullness Checker that uses a different type system for initialization. It was distributed with the Checker Framework through release 2.9.0 (dated 3 July 2019). If you wish to use it, install Checker Framework version 2.9.0.

The paper "Inference of field initialization" [SE11] describes inference for the Rawness Initialization Checker.

31.10 UI Thread Checker for ReactiveX

The Rx Thread & Effect Checker (`https://github.com/uber-research/RxThreadEffectChecker`) enforces UI Thread safety properties for stream-based Android applications. The paper "Safe Stream-Based Programming with Refinement Types" [SCSC18] describes the Rx Thread & Effect Checker.

31.11 Practical Immutability For Classes And Objects (PICO)

Practical Immutability For Classes And Objects (PICO) is a type system that supports class-level and object-level immutability, based on the Checker Framework. The implementation is available at `https://github.com/opprop/immutability`, and a docker image is available at `https://hub.docker.com/r/lnsun/pico`.

The theses "Context Sensitive Typechecking And Inference: Ownership And Immutability" [Ta18] and "An Immutability Type System for Classes and Objects: Improvements, Experiments, and Comparisons" [Sun21] describe PICO.

31.12 Read Checker and Cast Checker for ensuring that EOF is recognized

The `InputStream.read()` and `Reader.read()` methods read a byte (or character) from a stream. The methods return an `int`, using -1 to indicate the end of the stream. It is an error to cast the value to a byte (or character) without first checking the value against -1. This is rule CERT-FIO08-J. The Read Checker (`https://github.com/opprop/ReadChecker`) and/or Cast Checker (`https://github.com/opprop/cast-checker`) enforces it.

The paper "Don't miss the end: Preventing unsafe end-of-file comparisons" [CD18] and the thesis "Pluggable Properties for Program Understanding: Ontic Type Checking and Inference" [Che18] describe the Read Checker.

31.13 Ontology type system

The Ontology type system (`https://github.com/opprop/ontology`) groups related types into coarser concepts. For example, the SEQUENCE concept includes Java's `Array`, `List`, and their subtypes. Other examples are VELOCITY and FORCE.

The thesis "Pluggable Properties for Program Understanding: Ontic Type Checking and Inference" [Che18] describes the Ontic type system.

31.14 Glacier: Class immutability

Glacier (`https://glacier.coblenz.us/`) 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 changeable through any reference to the object.

Glacier is described by the paper "Glacier: Transitive class immutability for Java" (ICSE 2017) [CNA+17] and the PhD thesis "User-Centered Design of Principled Programming Languages" [Cob20].

31.15 SQL checker that supports multiple dialects

jOOQ is a database API that lets you build typesafe SQL queries. jOOQ version 3.9.0 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/jsr-308-and-the-checker-framework-add-even-mo`

31.16 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 2016-06-01 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.

IGJ and OIGJ are described in the papers "Object and reference immutability using Java generics" [ZPA+07] (ESEC/FSE 2007) and "Ownership and immutability in generic Java" [ZPL+10] (OOPSLA 2010). The Javari type system is described in "Javari: Adding reference immutability to Java" [TE05] (OOPSLA 2005); for inference, see "Inference of reference immutability" [QTE08] (ECOOP 2008) and "Parameter reference immutability: Formal definition, inference tool, and comparison" [AQKE09] (J.ASE 2009). The paper "Practical pluggable types for Java" [PAC+08] (ISSTA 2008) describes case studies in which the Javari and IGJ Checkers found previously-unknown errors in real software.

31.17 JCrypt: computation over encrypted data

JCrypt (`https://github.com/proganalysis/type-inference`) is a static program analysis which transforms a Java program into an equivalent one, so that it performs computation over encrypted data and preserves data confidentiality. It is described in the paper "JCrypt: Towards computation over encrypted data" [DMD16a].

31.18 DroidInfer: information flow

DroidInfer (`https://github.com/proganalysis/type-inference`) determines the Android library sources (e.g., location access, phone state) and sinks (e.g., Internet access) used by a program, and determines whether there is impermissible information flow between them. It is described in the paper "Scalable and Precise Taint Analysis for Android" (ISSTA 2015) [HDMD15].

31.19 Error Prone linter

Error Prone (`https://errorprone.info/`) is a linter or bug detector for Java. It reports violations of style rules. It is built on top of the Checker Framework's Dataflow Framework. See Section 40.10.1 for a comparison between Error Prone and the Nullness Checker.

31.20 SPARTA information flow type-checker for Android

SPARTA (`https://checkerframework.org/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 paper "Collaborative verification of information flow for a high-assurance app store" [EJM+14] (CCS 2014) describes the SPARTA toolset and information flow type-checker.

31.21 SFlow type system for information flow

SFlow (`https://github.com/proganalysis/type-inference`) is a context-sensitive type system for secure information flow. It contains two variants, SFlow/Integrity and SFlow/Confidentiality. SFlowInfer is its worst-case-cubic inference analysis.

The paper "Type-based taint analysis for Java web applications" (FASE 2014) and the thesis "An Inference And Checking Framework For Context-Sensitive Pluggable Types" (2014) describe SFlow and SFlowInfer.

31.22 CheckLT taint checker

CheckLT (https://checklt.github.io/) uses taint tracking to detect illegal information flows, such as unsanitized data that could result in a SQL injection attack.

31.23 EnerJ checker

EnerJ (https://sampa.cs.washington.edu/research/approximation/enerj.html) is 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. More details appear in the paper "EnerJ: Approximate Data Types for Safe and General Low-Power Computation" [SDF+11] (PLDI 2011).

31.24 ReIm immutability

A checker and inference tool for ReIm [HMDE12, Hua14], an immutability type system, is available at https://github.com/proganalysis/type-inference.

31.25 SFlow x ReIm for information flow and reference immutability

The "SFlow × ReIm" (https://github.com/proganalysis/type-inference) system combines information flow and reference immutability. It is described in the paper "Composing polymorphic information flow systems with reference immutability" [MH13].

31.26 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/ and https://github.com/opprop/universe.

The paper "Tunable static inference for Generic Universe Types" [DEM11] (ECOOP 2011) 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).

31.27 Safety-Critical Java checker

A checker for Safety-Critical Java (SCJ, JSR 302) [TPV10, TPNV11] is available at https://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/.

31.28 Thread locality checker

Loci is a checker for thread locality. For more details, see the paper "Loci: Simple thread-locality for Java" [WPM+09] (ECOOP 2009) and the thesis "The Design, Implementation and Evaluation of a Pluggable Type Checker for Thread-Locality in Java" [She11].

31.29 Units and dimensions checker

A checker for units and dimensions was available at https://www.lexspoon.org/expannots/.

Unlike the Units Checker that is distributed with the Checker Framework (see Chapter 20), 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.

31.30 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.

Accumulation analysis (Chapter 38, page 277) is a special case of typestate analysis. One instantiation of it is the Called Methods Checker (Chapter 7, page 68), which can check any property of the form "call method A before method B". It also ensures that builders are used correctly.

JaTyC (Section 31.7) is a Java typestate checker.

31.30.1 Comparison to flow-sensitive type refinement

The Checker Framework's flow-sensitive type refinement (Section 33.7) 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 33.10) also have similarities with typestate analysis and can occasionally substitute for it. For brevity, this discussion omits them.)

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

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

Chapter 32

Generics and polymorphism

Section 32.1 describes support for Java generics (also known as "parametric polymorphism"). Section 32.2 describes polymorphism over type qualifiers for methods. Section 32.3 describes polymorphism over type qualifiers for classes.

32.1 Generics (parametric polymorphism or type polymorphism)

The Checker Framework fully supports type-qualified Java generic types and methods (also known as "parametric polymorphism"). When instantiating a generic type, clients supply the qualifier along with the type argument, as in `List<@NonNull String>`. When using a type variable `T` within the implementation of a generic type, typically no type qualifier is written (see Section 32.1.4); rather, the instantiation of the type parameter is restricted (see Section 32.1.2).

32.1.1 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 arguments and the type qualifiers. By default, these inferred types are ignored in subtyping checks. If you supply the command-line option `-AignoreRawTypeArguments=false`, you will see errors from raw types.

32.1.2 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`.

Syntax for upper and lower bounds

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. Write the upper bound on the `extends` clause, and write the lower bound on the type variable.

```
class MyList<@LowerBound T extends @UpperBound Object> { ... }
```

You may omit either the upper or the lower bound, and the Checker Framework will use a default.

For a discussion of wildcards, see Section 32.1.5.

For a concrete example, consider the type system of the Regex Checker (see Figure 14.1, page 114) 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 `MyRegexes<T>` must be subtypes of `@Regex(1) String` and supertypes of `@Regex(5) void`. Since `void` cannot be used to instantiate a generic class, `MyRegexes` 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 subtype of @NonNull
```

Defaults

A generic type parameter or wildcard is written as `class MyClass<`*`@LowerBound`* `T extends` *`@UpperBound`* `JavaUpperBound>` or as `MyClass<`*`@UpperBound`* `? super` *`@LowerBound`* `JavaLowerBound>`, where "*`@LowerBound`*" and "*`@UpperBound`*" are type qualifiers.

For lower bounds: If no type annotation is written in front of `?`, then the lower bound defaults to `@`*`BottomType`* `void`.

For upper bounds:

- If the `extends` clause is omitted, then the upper bound defaults to `@`*`TopType`* `Object`.
- 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 33.5.3).

The upper-bound rules mean that even though in Java the following two declarations are equivalent:

```
class MyClass<T>
class MyClass<T extends Object>
```

they specify different type qualifiers on the upper bound, if the type system's default annotation is not its top annotation.

The Nullness type system is an example.

```
class MyClass<T>                 ==  class MyClass<T extends @Nullable Object>
class MyClass<T extends Object>  ==  class MyClass<T extends @NonNull Object>
```

The rationale for this choice is:

- The "`<T>`" in `MyClass<T>` means "fully unconstrained", and the rules maintain that, without the need for a programmer to change existing code.
- The "`Object`" in `MyClass<T extends Object>` is treated exactly like every other occurrence of `Object` in the program — it would be confusing for different occurrences of `Object` to mean different annotated types.

Here are some style guidelines:

- Use "`<T>`" when there are no constraints on the type qualifiers. This is short and is what already appears in source code.
- Whenever you write an `extends` clause, write an explicit type annotation on it. For example, for the Nullness Checker, write `class MyClass<T>` rather than `class MyClass<T extends @Nullable Object>`, and write `class MyClass<T extends @NonNull Object>` rather than `class MyClass<T extends Object>`.

For further discussion, see Section 40.7.2.

32.1.3 Overriding a generic method

When a method overrides a generic method, the type parameter bounds of the overriding method must contain the type parameter bounds of the overridden method. That is, the upper bound of the overriding method must be a supertype of the upper bound of the overridden method, and the lower bound of the overriding method must be a subtype of the lower bound of the overridden method. The reason is that a client may call the overridden method with any type argument that is within the bounds of the overridden method, and the overriding method is what runs, so the overriding method must permit every such type argument.

```
class Super {
  <T extends @Nullable Object> T pick(T p) { ... }
}

class Sub extends Super {
  @Override
  // error: the bounds do not contain those of the overridden method
  <@Nullable T extends @Nullable Object> T pick(T p) { ... }
}
```

The declaration in `Sub` is illegal because a client may call `pick` through a reference of type `Super`, with the type argument `@NonNull String` that `Sub.pick` does not permit:

```
void client(Super s) {
  @NonNull String result = s.<@NonNull String>pick("x");
}
```

It is legal for the overriding method to state weaker bounds than the overridden method, because that permits more type arguments rather than fewer.

The bounds are not checked when the overridden method is unchecked code that receives conservative defaults (Section 36.4.1), because no overriding method could contain the conservative bounds. The Map Key Checker does not check lower bounds, because its default for lower bounds differs between the JDK and other code (Section 4.3.1).

32.1.4 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 sometimes useful, but more often the implementation of a generic type just uses the type variable `T`, whose instantiation is restricted (see Section 32.1.2).

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.

32.1.5 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 one bound. You can specify the upper bound with `<? extends SomeType>`, in which case the lower bound is implicitly the bottom type `void`. You can specify the lower bound with `<? super OtherType>`, in which case the upper bound is implicitly the top type `Object`. The Checker Framework is more flexible: it lets you similarly write annotations on both the upper and lower bound.

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 33.5.5).

32.1.6 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 a 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", above,
@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`.

32.1.7 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 clients read values of that type from the class but never pass values of that type to the class — 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`. A client can read elements but not write them, 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 supports 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.

32.1.8 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.

32.1.9 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) { ... }
```

32.2 Qualifier polymorphism for methods

Type qualifier polymorphism permits a single method to have multiple different qualified type signatures.

Here is where a polymorphic qualifier (e.g., `@PolyNull`) can be used:

- Polymorphic qualifiers are most often used in method signatures. See the examples below in Section 32.2.1.
- Polymorphic qualifiers can also be written in method bodies (implementations).
- If you can use generics, you typically do not need to use a polymorphic qualifier. Do not write a polymorphic qualifier on a type variable declaration.
- Polymorphic qualifiers may not be used on a class declaration. To apply qualifier polymorphism to classes, use a class qualifier parameter; see Section 32.3.
- A polymorphic qualifier may be used on a field declaration only in a class with a class qualifier parameter; see Section 32.3.2.

- If a class has a class qualifier parameter, then a polymorphic qualifier written on a method in the class has a slightly different meaning, see Section 32.3.1.

Section 37.5.2 explains how to define a polymorphic qualifier.

32.2.1 Using polymorphic qualifiers in a method signature

A method whose signature has a polymorphic qualifier (such as `@PolyNull`) 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.

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.

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.

32.2.2 Relationship to subtyping and generics

Qualifier polymorphism has the same purpose and plays the same role as Java's generics. You use them for similar reasons, 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.

32.2.3 Using multiple polymorphic qualifiers in a method signature

Usually, it does not make sense to write only a single instance of a polymorphic qualifier in a method definition: if you write one instance of (say) `@PolyNull`, then you should use at least two. 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. (Section 32.2.4 describes some exceptions to this rule: times when it makes sense to write a single polymorphic qualifier in a signature.)

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());
}
```

32.2.4 Using a single polymorphic qualifier in a method signature

As explained in Section 32.2.3, 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 a return type

It is unusual, but permitted, to write just one polymorphic qualifier, on a return type.

Similarly, 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()`.

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) { ... }
```

Another example is a method that writes bytes to a file. It accepts an array of signed or unsigned bytes, and it behaves identically for both:

```
void write(@PolySigned byte[] b) { ... }
```

These examples use arrays, but there are similar examples that use collections.

Don't use a single polymorphic qualifier on a formal parameter type

There is no point in writing just one polymorphic qualifier in a method signature, as the main type qualifier on a formal parameter type. Consider this signature:

```
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)
```

However, it does make sense to write a single polymorphic qualifier *within* a formal parameter type, as in

```
void m2a(List<@PolyNull MySubClass> strings)
```

which is similar to

```
void m2b(List<? extends @Nullable MySubClass> strings)
```

Method `m2b()` can be called with (for example) `List<@Nullable MySubClass>` and `List<@NonNull MySubClass>`, which are not legal arguments to method `m2a()`.

32.3 Class qualifier parameters

Class qualifier parameters permit you to supply a type qualifier (only, without a Java base type) to any class (generic or not).

When a generic class represents a collection, a user can write a type qualifier on the type argument, as in `List<@Tainted Character>` versus `List<@Untainted Character>`. When a non-generic class represents a collection with a hard-coded type (as `StringBuffer` hard-codes `Character`), you can use a class qualifier parameter to distinguish `StringBuffer`s that contain different types of characters.

To add a qualifier parameter to a class, annotate its declaration with `@HasQualifierParameter` and supply the class of the top qualifier as its argument.

```
@HasQualifierParameter(Tainted.class)
class StringBuffer { ... }
```

A qualifier on a use of `StringBuffer` is treated as appearing both on the `StringBuffer` and on its conceptual type argument. That is:
`@Tainted StringBuffer` ≈ `@Tainted Collection<@Tainted Character>`
`@Untainted StringBuffer` ≈ `@Untainted Collection<@Untainted Character>`

If two types have different qualifier arguments, they have no subtyping relationship. (This is "invariant subtyping", also used by Java for generic classes.) In particular, `@Untainted StringBuffer` is not a subtype of `@Tainted StringBuffer`; an attempt to cast between them, in either direction, will yield an `invariant.cast.unsafe` error.

`@HasQualifierParameter` is inherited. If type *T* has a `@HasQualifierParameter` annotation, then its subtypes are automatically treated as having the same annotation.

Within a class with a qualifier parameter, the default qualifier for uses of that class is the polymorphic qualifier.

32.3.1 Resolving polymorphism when the receiver type has a polymorphic qualifier

A class qualifier parameter changes the rules for instantiating polymorphic qualifiers (Section 32.2). If the receiver type has a qualifier parameter and is annotated with a polymorphic qualifier, then at a call site all polymorphic annotations are instantiated to the same qualifier as the type of the receiver expression of the method call. Otherwise, use the rules of Section 32.2.

For example, consider

```
@HasQualifierParameter(Tainted.class)
class Buffer {
  void append(@PolyTainted Buffer this, @PolyTainted String s) { ... }
}
```

Because `@PolyTainted` applies to a type (`Buffer`) with a qualifier parameter, all uses of `@PolyTainted` are instantiated to the qualifiers on the type of the receiver expression at call sites to `append`. For example,

```
@Untainted Buffer untaintedBuffer = ...;
@Tainted String taintedString = ...;
untaintedBuffer.append(taintedString); // error: [argument]
```

The above `append` call is illegal because the `@PolyTainted` qualifier is instantiated to `@Untainted` and the type of the argument is `@Tainted`, which is a supertype of `@Untainted`. If the type of `untaintedBuffer` were `@Tainted` then the call would be legal.

32.3.2 Using class qualifier parameters in the type of a field

To express that the type of a field should have the same qualifier as the class qualifier parameter, annotate the field type with the polymorphic qualifier for the type system.

```
@HasQualifierParameter(Tainted.class)
class Buffer {
  @PolyTainted String field;
}
```

At a field access where the declared type of the field has a polymorphic qualifier, that polymorphic qualifier is instantiated to the qualifier on the type of the receiver of the field access (or in the case of type variables, the qualifier on the upper bound). That is, the qualifier on `myBuffer.field` is the same as that on `myBuffer`.

32.3.3 Local variable defaults for types with qualifier parameters

Local variables default to the top type (see Section 33.5.3). Type refinement determines whether a variable can be treated as a suitable subtype, and annotations on local variables are rarely needed as a result. However, since qualifier parameters add invariant subtyping, type refinement is no longer valid. For example, suppose in the following code that `StringBuffer` is annotated with `@HasQualifierParameter(Tainted.class)`.

```
    void method(@Untainted StringBuffer buffer) {
        StringBuffer local = buffer;
        executeSql(local.toString());
    }

    void executeSql(@Untainted String code) {
        // ...
    }
```

Normally, the framework would determine that `local` has type `@Untainted StringBuffer` and the call to `executeSql` would be valid. However, since by default `local` has type `@Tainted StringBuffer`, and `@Untainted StringBuffer` is not a subtype, no type refinement would be performed, leading to an error. Fixing this would require manually annotating `local` as an `@Untainted StringBuffer`, increasing the annotation burden on programmers.

For this reason, local variables with types that have a qualifier parameter use different defaulting rules. When a local variable has an initializer, the type of that initializer is used as the default type of that variable if no other annotations are written. For example, in the above code, the type of `local` would be `@Untainted StringBuffer`. This eliminates the need for type refinement.

32.3.4 Qualifier parameters by default

If many classes in a project should have `@HasQualifierParameter`, it's possible to enable it on all classes in a package by default. Writing `@HasQualifierParameter` on a package is equivalent to writing `@HasQualifierParameter` on each class in that package and all subpackages with the same arguments.

For example, writing this annotation enables `@HasQualifierParameter` for all classes in `mypackage`.

```
@HasQualifierParameter(Tainted.class)
package mypackage;
```

When using `@HasQualifierParameter` on a package, it's possible to disable it for a specific class using `@NoQualifierParameter`. Writing this on a class indicates that it has no class qualifier parameter, and `@HasQualifierParameter` will not be enabled by default. Like `@HasQualifierParameter`, it takes one or more top annotations. It is illegal to explicitly write both `@HasQualifierParameter` and `@NoQualifierParameter` on the same class for the same hierarchy.

32.3.5 Types with qualifier parameters as type arguments

Types with qualifier parameters are only allowed as type arguments to type parameters whose upper bound has a qualifier parameter. If they were allowed as type arguments for any type parameter, then unsound casts would be permitted. For example:

```
@HasQualifierParameter(Tainted.class)
interface Buffer {
    void append(@PolyTainted String s);
}

public class ClassQPTypeVarTest {
    <T> @Tainted T cast(T param) {
        return param;
    }

    void bug(@Untainted Buffer b, @Tainted String s) {
        cast(b).append(s);  // error
    }
}
```

Chapter 33

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.

33.1 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 = strings[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.

33.2 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 14, page 113), 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 `@PartialRegex 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"}`).

33.3 Upper bound of qualifiers on uses of a given type (annotations on a class declaration)

The examples in this section use the type qualifier hierarchy `@A :> @B :> @C`.

A qualifier on a use of a certain type must be a subtype or equal to the upper bound for that type. The upper bound of qualifiers used on a given type is specified by annotating the type declaration with some qualifier — that is, by writing an annotation on a class declaration.

```
@C class MyClass {}
```

This means that `@B MyClass` is an invalid type. (Annotations on class declarations may also specify default annotations for uses of the type; see Section 33.5.1.)

If it is not possible to annotate the class's definition (e.g., for primitives and some library classes), the type-system designer can specify an upper bound by using the meta-annotation `@UpperBoundFor`.

If no annotation is present on a type declaration and if no `@UpperBoundFor` mentions the type, then the bound is top. This can be changed by overriding `AnnotatedTypeFactory#getTypeDeclarationBounds`.

There are two exceptions.

- An expression can have a supertype of the upper bound; that is, some expression could have type `@B MyClass`. This type is not written explicitly, but results from viewpoint adaptation.
- Using usual CLIMB-to-top rules (Section 33.5.3), local variables of type MyClass default to `@A MyClass`. It is legal for `@A MyClass` to be the type of a local variable. For consistency, users are allowed to write such a type on a local variable declaration.

Due to existing type rules, an expression of type `@A MyClass` can only be used in limited ways.

- Since every field, formal parameter, and return type of type MyClass (or lower) is annotated as `@B` (or lower), it cannot be assigned to a field, passed to a method, or returned from a method.
- It can be used in a context that requires `@A Object` (or whatever the least supertype is of MyClass for which the `@A` qualifier is permitted). Examples include being tested against `null` or (for most type systems) being passed to polymorphic routines such as `System.out.println` or `System.identityHashCode`.

These operations might refine its type. If a user wishes to annotate a method that does type refinement, its formal parameter must be of illegal type `@A MyClass`, which requires a warning suppression.

If the framework were to forbid expressions and local variables from having types inconsistent with the class annotation, then important APIs and common coding paradigms would no longer type-check.

Consider the annotation

```
@NonNull class Optional { ... }
```

and the client code

```
Map<String, Optional> m;
String key = ...;
Optional value = m.get(key);
if (value != null) {
  ...
}
```

The type of `m.get(key)` is `@Nullable Optional`, which is an illegal type. However, this is a very common paradigm. Programmers should not need to rewrite the code to test `m.containsKey(key)` nor suppress a warning in this safe code.

33.4 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, programmers rarely write qualifiers in method bodies (except on type arguments and array component types).

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. If there is no explicit qualifier on a type, then a default qualifier is applied; see Section 33.5. Defaulted qualifiers are treated by checkers exactly as if the programmer had written them explicitly.
3. 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 33.7.

33.5 Default qualifier for unannotated types

An unannotated Java type is treated as if it had a default annotation. Both the type system designer and an end-user programmer can control the defaulting. Defaulting never applies to uses of type variables, even if they do not have an explicit type annotation. Most of this section is about defaults for source code that is read by the compiler. When the compiler reads a `.class` file, different defaulting rules apply. See Section 33.5.6 for these rules.

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

1. The qualifier specified via `@DefaultQualifierForUse` on the declaration of `MyClass`. (Section 33.5.1)
2. If no `@NoDefaultQualifierForUse` is written on the declaration of `MyClass`, the qualifier explicitly written on the declaration of `MyClass`. (Section 33.5.1)
3. The qualifier with a meta-annotation `@DefaultFor(types = MyClass.class)`. (Section 33.5.1)
4. The qualifier with a meta-annotation `@DefaultFor(typeKinds = KIND)`, where `KIND` is the `TypeKind` of `MyClass`. (Section 33.5.1)
5. The qualifier with a meta-annotation `@DefaultFor(names = REGEX)`, where `REGEX` matches the name of the variable being defined (if any). For return types, the name of the method is used. (Section 33.5.1)
6. The qualifier in the innermost user-written `@DefaultQualifier` for the location of the use of `MyClass`. (Section 33.5.2)
7. The qualifier in the meta-annotation `@DefaultFor` for the location of the use of `MyClass`. These are defaults specified by the type system designer (Section 37.5.4); this is usually CLIMB-to-top (Section 33.5.3).
8. The qualifier with the meta-annotation `@DefaultQualifierInHierarchy`.

If the unannotated type is the type of a local variable, then the first 5 rules are skipped and only rules 6 and 7 apply. If rule 7 applies, it makes the type of local variables top so they can be refined.

33.5.1 Default for use of a type

The type declaration annotation `@DefaultQualifierForUse` indicates that the specified qualifier should be added to all unannotated uses of the type.

For example:

```
@DefaultQualifierForUse(B.class)
class MyClass {}
```

This means any unannotated use of `MyClass` is treated as `@B MyClass` by the checker. (Except for locals, which can be refined.)

Similarly, the meta-annotation `@DefaultFor` can be used to specify defaults for uses of types, using the `types` element, or type kinds, using the `typeKinds` element.

Interaction between qualifier bounds and `DefaultQualifierForUse`:

- If a type declaration is annotated with a qualifier bound, but not a `@DefaultQualifierForUse`, then the qualifier bound is added to all unannotated uses of that type (except locals). For example, `@C class MyClass {}` is equivalent to

  ```
  @DefaultQualifierForUse(C.class)
  @C class MyClass {}
  ```

- If the qualifier bound should not be added to all unannotated uses, then `@NoDefaultQualifierForUse` should be written on the declaration:

  ```
  @NoDefaultQualifierForUse
  @C class MyClass {}
  ```

  This means that unannotated uses of MyClass are defaulted normally.
- If neither `@DefaultQualifierForUse` nor a qualifier bound is present on a type declaration, that is equivalent to writing `@NoDefaultQualifierForUse`.

33.5.2 Controlling defaults in source code

The end-user programmer specifies a default qualifier by writing the `@DefaultQualifier(`*`ClassName`*`, [`*`locations`*`])` 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.DefaultQualifier;
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
      ...
    @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"
    ...
  }
}
```

You may write multiple `@DefaultQualifier` annotations at a single location.

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

33.5.3 Defaulting rules and CLIMB-to-top

Each type system defines a default qualifier (see Section 37.5.4). For example, the default qualifier for the Nullness Checker is `@NonNull`. When a user writes an unqualified type such as `Date`, the Nullness Checker interprets it as `@NonNull Date`.

The type system applies that default qualifier to most but not all type uses. 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 the default for the CLIMB locations: **C**asts, **L**ocals, and (some) **Im**plicit **B**ounds. For example, when the user writes an unqualified 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). (Casts are treated a bit specially; see below.)

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), see Section 33.5.6.

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 33.7) 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 default to the top type because they might catch an exception thrown anywhere in the program.
  An alternate design for exception parameters would be to default exception parameters to some other type T (instead of the top type); then the Checker Framework would need to issue a warning at every `throw` statement whose argument might not be a subtype of T. A checker can implement this alternate design by overriding a few methods. The alternative is not appropriate for all type systems. The alternative is unsound for deep type systems because the JDK's annotations are trusted rather than checked. A deep type system is one where the type of a field can determine the type of its containing instance, such as tainting. Example: a user passes a secret regex to the JDK, and the JDK throws a format exception that includes the regex. This could be caught by a `catch` clause in the program whose exception parameter is not annotated as secret. As another example, the user passes a secret integer and the JDK throws an ArithmeticException that reveals the value.
- Cast types are defaulted to the same type as their argument expression. This has the same effect as if they were given the top type and then flow-sensitively refined to the type of their argument. However, note that programmer-written type qualifiers are *not* refined, so writing the top annotation is not the same as writing no annotation.
- Implicit upper bounds are defaulted to top to allow them to be instantiated in any way.
  Suppose a user declares a class as `class C<T> { ... }`. The Checker Framework assumes that the user intended to allow any instantiation of the class. The Checker Framework interprets the declaration 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>`. This is a bit unfortunate, but it is the least bad option. The more confusing alternative would be 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 32.1.2.

A `@DefaultQualifier` that specifies a CLIMB-to-top location takes precedence over the CLIMB-to-top rule.
Here is how the Nullness Checker overrides part of the CLIMB-to-top rule:

```
@DefaultQualifierInHierarchy
@DefaultFor({ TypeUseLocation.EXCEPTION_PARAMETER })
public @interface NonNull {}

public @interface Nullable {}
```

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

33.5.4 Inherited defaults

When overriding a method, programmers must fully specify types in the overriding method, which duplicates information on the overridden method. By contrast, declaration annotations that are meta-annotated with `@InheritedAnnotation` are inherited by overriding methods.

An example for type annotations is that when defining an `equals()` method, programmers must write the type annotation `@Nullable`:

```
public boolean equals(@Nullable Object obj) {
    ...
}
```

An alternate design would be for every annotation on a superclass member to be automatically inherited by subclasses that override it.

The alternate design would reduce annotation effort.

However, the alternate design would reduce program comprehensibility. 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. For declaration annotations, no computation is necessary; that is why they may be inherited. Computation is necessary for type annotations 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, the Checker Framework would need to issue an error if no such annotations existed.

Because a program is read more often than it is edited/annotated, the Checker Framework does not currently support the alternate design. In the future, this feature may be added.

33.5.5 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` 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 that 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 that the upper bound of the wildcard `? extends Object` is defaulted to `@NonNull` using the CLIMB-to-top rule (see Section 33.5.3). 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`.

33.5.6 Default qualifiers for `.class` files (library defaults)

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. The rules are the same regardless of whether the `.class` file is from your own codebase or from a library, including the JDK.

If the `.class` file was compiled *using* the checker (that is, 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.)

The rest of this section discusses the situation when the `.class` file was *not* compiled using the checker (that is, the checker was not run during the compiler execution that created the `.class` file). The `.class` file contains only the type qualifiers that the programmer wrote explicitly — possibly none. (Furthermore, there is no guarantee that these qualifiers are correct, since they have not been checked.) Each checker decides what qualifier to use where the programmer did not write an annotation.

- With `-AuseConservativeDefaultsForUncheckedCode=-bytecode`: (This is also the behavior if `-AuseConservativeDefaults` is not supplied or is supplied without "bytecode" in its argument.)
  The source code defaulting rules are used.
  For example, an unannotated method

  ```
  String concatenate(String p1, String p2)
  ```

  in a classfile would be interpreted as

```
@Default String concatenate(@Default String p1, @Default String p2)
```

for an appropriate value of `@Default`. (The default might be different for return types and parameter types, depending on how the checker is written.)
These defaulting rules are unsafe: the default might be different than what the library author intended, so the Checker Framework may not warn about some misuses of the library. (This includes both misuse by passing in a bad value and misuse by using a returned value incorrectly.) However, this unsafe default can be useful because it permits you to focus on errors within the codebase before you have annotated the libraries it uses.

- With `-AuseConservativeDefaultsForUncheckedCode=bytecode`:
  - For method parameters and lower bounds, use the bottom qualifier (see Section 37.5.7).
  - For method return values, fields, and upper bounds, use the top qualifier (see Section 37.5.7).

  For example, an unannotated method

  ```
  String concatenate(String p1, String p2)
  ```

  in a classfile would be interpreted as

  ```
  @MyTopQualifier String concatenate(@MyBottomQualifier String p1, @MyBottomQualifier String p2)
  ```

  where `@MyTopQualifier` is the top qualifier in the hierarchy and `@MyBottomQualifier` is the bottom qualifier in the hierarchy.
  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 36) or alternately suppress the warnings (see Chapter 34).
  There is one case in which these choices are not conservative. Fields default to the top qualifier, so any assignment into a field is permitted. If an unannotated library has a mutable public field (which would be poor code style), the Checker Framework won't warn about incorrect assignments to the field.

33.6 Annotations on constructors

33.6.1 Annotations on constructor declarations

An annotation on the "return type" of a constructor declaration indicates what the constructor creates. For example,

```
@B class MyClass {
  @C MyClass() {}
}
```

means that invoking that constructor creates a `@C MyClass`.

The Checker Framework cannot verify that the constructor really creates such an object, because the Checker Framework does not know the type-system-specific semantics of the `@C` annotation. Therefore, if the constructor result type is different than the top annotation in the hierarchy, the Checker Framework will issue a warning. The programmer should check the annotation manually, then suppress the warning.

Defaults

If a constructor declaration is unannotated, it defaults to the same type as that of its enclosing class (rather than the default qualifier in the hierarchy). For example, the Tainting Checker (Chapter 12) has `@Tainted` as its default qualifier. Consider the following class:

```
@Untainted class MyClass {
  MyClass() {}
}
```

The constructor declaration is equivalent to `@Untainted MyClass() {}`.

The Checker Framework produces the same error messages for explicitly-written and defaulted annotations.

33.6.2 Annotations on constructor invocations

The type of a method call expression `x.myMethod(y, z)` is determined by the return type of the declaration of `myMethod`. There is no way to write an annotation on the call to change its type. However, it is possible to write a cast: `(@Anno SomeType) x.myMethod(y, z)`. The Checker Framework will issue a warning that it cannot verify that the downcast is correct. The programmer should manually determine that the annotation is correct and then suppress the warning.

A constructor invocation `new MyClass()` is also a call, so its semantics are similar. The type of the expression is determined by the annotation on the result type of the constructor declaration. It is possible to write a cast `(@Anno MyClass) new MyClass()`. The syntax `new @Anno MyClass()` is shorthand for the cast. For either syntax, the Checker Framework will issue a warning that it cannot verify that the cast is correct. The programmer may suppress the warning if the code is correct.

33.7 Type refinement (flow-sensitive type qualifier inference)

A checker can sometimes deduce that an expression's type is more specific than — that is, a subtype of — its declared or defaulted type (Section 33.5). This is called "flow-sensitive type refinement" or "local type inference".

Due to local type refinement, a programmer typically does not write any qualifiers on local variables within a method body (except on type arguments and array component types). However, the programmer must write type annotations for method signatures (arguments and return values) and fields, unless the default annotations are correct. Local type refinement does not change the source code to insert the inferred annotations on local variables.

33.7.1 Type refinement examples

Here is an example for the Nullness Checker (Chapter 3, page 31). `myVar` is declared as `@Nullable String`, but it is treated as `@NonNull String` within the body of the `if` test.

```
@Nullable String myVar;
...                    // myVar has type @Nullable String here.
myVar.hashCode();      // warning: possible dereference of null.
...
if (myVar != null) {
  ...                  // myVar has type @NonNull String here.
  myVar.hashCode();    // no warning.
}
```

Here is another example. Note that the same expression may yield a warning or not depending on its context (that is, depending on the current type refinement).

```
@Nullable String myVar;
...                    // myVar has type @Nullable String
myVar = "hello";
...                    // myVar has type @NonNull String
myVar.hashCode();      // no warning
...
myVar = myMap.get(someKey);
...                    // myVar has type @Nullable String
myVar.hashCode();      // warning: possible dereference of null
```

Type refinement applies to every checker, including new checkers that you write. Here is an example for the Regex Checker (Chapter 14, page 113):

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

33.7.2 Type refinement behavior

The checker treats a variable or expression as a subtype of its declared type:

- starting at the time that it is assigned a value, a method establishes a postcondition (e.g., as expressed by `@EnsuresNonNull` or `@EnsuresQualifierIf`), or a run-time check is performed (e.g., via an assertion or `if` statement).
- until its value might change (e.g., via an assignment, or because a method call might have a side effect).

The checker never treats a variable as a supertype of its declared type. For example, an expression with declared type `@NonNull` is never treated as possibly-null, and such an assignment is always illegal.

The functionality has a variety of names: type refinement, flow-sensitive type qualifier inference, local type inference, and sometimes just "flow".

33.7.3 Which types are refined

You generally do not need to annotate the top-level type of a local variable. You do need to annotate its type arguments or array element types. (Type refinement does not change them, because doing so would not produce a subtype, as explained in Section 32.1.7 and Section 33.1.) Type refinement works within a method, so you still need to annotate method signatures (parameter and return type) and field types.

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 41.2) that includes a small piece of Java code that reproduces the problem.

Fields and type refinement

Type refinement 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 or any other expression. 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() ...
  }
}
```

A method call may invalidate inferences about field types; see Section 33.7.5.

33.7.4 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 31)
- Map Key Checker to track which values are keys in a map (see Chapter 4, page 54)
- Optional Checker for errors in using the `Optional` type (see Chapter 5, page 59)
- Lock Checker for concurrency and lock errors (see Chapter 10, page 87)
- Index Checker for array accesses (see Chapter 11, page 98)
- Resource Leak Checker for ensuring that resources are disposed of properly (see Chapter 8, page 73)
- SQL Quotes Checker to mitigate SQL injection attacks (see Chapter 13, page 111)
- Regex Checker to prevent use of syntactically invalid regular expressions (see Chapter 14, page 113)
- Format String Checker to ensure that format strings have the right number and type of `%` directives (see Chapter 15, page 116)
- Internationalization Format String Checker to ensure that i18n format strings have the right number and type of `{}` directives (see Chapter 16, page 123)

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 62)
- Property File Checker to ensure that valid keys are used for property files and resource bundles (see Chapter 17, page 129)
- Internationalization Checker to ensure that code is properly internationalized (see Section 17.2, page 130)
- Signature String Checker to ensure that the string representation of a type is properly used, for example in `Class.forName` (see Chapter 18, page 132)
- Constant Value Checker to determine whether an expression's value can be known at compile time (see Chapter 24, page 161)
- Modifiability Checker to warn about a possible `UnsupportedOperationException` at run time when modifying a collection (see Chapter 22, page 148)

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

- Initialization Checker to ensure all fields are set in the constructor (see Section 3.8, page 44)
- Called Methods Checker for the builder pattern (see Chapter 7, page 68)
- Fake Enum Checker to allow type-safe fake enum patterns and type aliases or typedefs (see Chapter 9, page 83)
- Tainting Checker for trust and security errors (see Chapter 12, page 108)
- GUI Effect Checker to ensure that non-GUI threads do not access the UI, which would crash the application (see Chapter 19, page 135)

- Units Checker to ensure operations are performed on correct units of measurement (see Chapter 20, page 140)
- Signedness Checker to ensure unsigned and signed values are not mixed (see Chapter 21, page 144)
- Purity Checker to identify whether methods have side effects (see Chapter 23, page 157)
- Initialized Fields Checker to ensure all fields are set in the constructor (see Chapter 27, page 173)
- Aliasing Checker to identify whether expressions have aliases (see Chapter 28, page 176)
- Must Call Checker to over-approximate the methods that should be called on an object before it is de-allocated (see Chapter 29, page 180)
- Subtyping Checker for customized checking without writing any code (see Chapter 30, page 183)

33.7.5 Side effects, determinism, purity, and type refinement

Calling a method typically causes the checker to discard its knowledge of the refined type, because the method might assign a field. The `@SideEffectFree` annotation indicates that the method has no side effects, so calling it does not invalidate any dataflow facts.

Calling a method twice might have different results, so facts known about one call cannot be relied upon at another call. The `@Deterministic` annotation indicates that the method returns the same result every time it is called on the same arguments.

`@Pure` means both `@SideEffectFree` and `@Deterministic`. The `@TerminatesExecution` annotation indicates that a given method never returns. This can enable type refinement to be more precise.

Chapter 23 gives more information about these annotations. This section explains how to use them to improve type refinement.

Side effects

Consider the following declarations and uses:

```
@Nullable Object myField;

int computeValue() { ... }

void m() {
  ...
  if (myField != null) {
                      // The type of myField is now "@NonNull Object".
    int result = computeValue();
                      // The type of myField is now "@Nullable Object",
                      // because computeValue might have set myField to null.
    myField.toString(); // Warning: possible null pointer exception.
  }
}
```

There are three ways to express that `computeValue` does not set `myField` to `null`, and thus to prevent the Nullness Checker from issuing a warning about the call `myField.toString()`.

1. If `computeValue` has no side effects, declare the method as `@SideEffectFree`:

   ```
   @SideEffectFree
   int computeValue() { ... }
   ```

   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.
2. If no method resets `myField` to `null` after it has been initialized to a non-null value (even if a method has some other side effect), declare the field as `@MonotonicNonNull`:

   ```
   @MonotonicNonNull Object myField;
   ```

3. If `computeValue` sets `myField` to a non-null value (or maintains it as a non-null value), declare the method as `@EnsuresNonNull`:

   ```
   @EnsuresNonNull("myField")
   int computeValue() { ... }
   ```

   If `computeValue` maintains `myField` as a non-null value, even if it might have other side effects and even if other methods might set `myField` to `null`, declare it as

   ```
   @RequiresNonNull("myField")
   @EnsuresNonNull("myField")
   int computeValue() { ... }
   ```

There are two other ways to suppress the warning:

- Put the expression in a local variable before the invocation. A method call never affects the value of a local variable.
- If the warning is about a formal parameter (including the receiver) to a method `foo()`, then instead of declaring `foo()` as `@SideEffectFree`, you can supply the `-AinvocationPreservesArgumentNullness` command-line argument; see Section 3.1.1.

Deterministic methods

Consider the following declaration and uses:

```
@Nullable Object getField(Object arg) { ... }

void m() {
  ...
  if (x.getField(y) != null) {
    x.getField(y).toString(); // warning: possible null pointer exception
  }
}
```

The Nullness Checker issues a warning regarding the `toString()` call, because its receiver `x.getField(y)` might be `null`, according to the `@Nullable` return type in the declaration of `getField`. The Nullness Checker cannot assume that `getField` returns non-null on the second call, just based on the fact that it returned non-null on the first call.

To indicate that a method returns the same value each time it is called on the same arguments, use the `@Deterministic` annotation. Actually, it is necessary to use `@Pure`, which means both `@Deterministic` and `@SideEffectFree`, because otherwise the first call might change a value that the method depends on.

If you change the declaration of `getField` to

```
@Pure
@Nullable Object getField(Object arg) { ... }
```

then the Nullness Checker issues no warnings. Because `getField` is `@SideEffectFree`, the values of `x` and `y` are the same at both invocations. Because `getField` is `@Deterministic`, the two invocations of `x.getField(y)` have the same value. Therefore, `x.getField(y)` is non-null within the `then` branch of the `if` statement.

33.7.6 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`, `-enableassertions`, `-da`, or `-disableassertions` 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 (that is, Java `assert` statements) are enabled, as if Java is run with the `-ea` or `-enableassertions` command-line argument. If you supply the `-AassumeAssertionsAreDisabled` command-line option, then the Checker Framework assumes assertions are disabled, as if Java is run with the `-da` or `-disableassertions` command-line argument. 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 34.2.

33.7.7 The `var` keyword

The initial type of a variable declared with `var` is exactly the type of the initializer expression at the declaration.

```
var list1 = new ArrayList<@Tainted String>(); // type of list1 is ArrayList<@Tainted String>
var list2 = new ArrayList<@Untainted String>(); // type of list2 is ArrayList<@Untainted String>
```

Flow-sensitive type refinement applies to these variables as usual. The base type qualifier may change at a later program point due to a subsequent assignment, but the qualifiers on type parameters and array contents remain exactly the same.

```
var list1 = new ArrayList<@Tainted String>(); // type of list1 is ArrayList<@Tainted String>
var list2 = new ArrayList<@Untainted String>(); // type of list2 is ArrayList<@Untainted String>
var list3 = list1; // type of list3 is ArrayList<@Tainted String>
list2 = list1; // assignment error
```

33.8 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`
- `@SideEffectsOnly`
- `@RequiresNonNull`
- `@EnsuresNonNull`
- `@EnsuresNonNullIf`
- `@RequiresPresent`
- `@EnsuresPresent`
- `@EnsuresPresentIf`
- `@KeyFor`
- `@EnsuresKeyFor`
- `@EnsuresKeyForIf`
- `@I18nFormatFor`
- `@EnsuresLockHeld`
- `@EnsuresLockHeldIf`
- `@GuardedBy`
- `@Holding`

The set of permitted expressions is a subset of all Java expressions, with a few extensions. The extensions are formal parameters like "`#1`" and (for some type systems) "`<self>`".

- `this`, the receiver object. 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 static methods. For a field, `this` is the field's receiver (sometimes called its container). For a local variable, it is the method's receiver.
- `super`, the receiver object as seen from the superclass. This can be used to refer to fields shadowed in the subclass (although shadowing fields is discouraged in Java).
- `<self>`, 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, e.g., `#2`. It is 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.
  The formal parameter syntax `#1` is less natural in source code than writing the formal parameter name. This syntax is necessary for separate compilation, because no formal parameter name information is available in a `.class` file. Suppose an annotated method `m` has already been compiled into a `.class` file, perhaps by a compilation that did not use the Checker Framework. When a client of `m` is later compiled, it cannot interpret a formal parameter name, but it can interpret a number.
  Within a method *body*, you may use the formal parameter name. The formal parameter name never works within a method *signature* or for a contract (pre- or post-condition) annotation; in those locations, an identifier is interpreted as a field name (not a formal parameter).
- a local variable, e.g., `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, e.g., `System.out`. Write the class name and the variable.
- 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]`.
- an array creation. For example: `new int[10]`, `new String[] {"a", "b"}`.
- literals: string, integer, char, long, float, double, 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")`.
  Currently, the Checker Framework cannot prove all contracts about method calls, so you may need to suppress some warnings.
  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.
- a binary expression, e.g., `x + y` or `#1 - 1`. These are used by the Index Checker, for example.
- a class name expression within another expression, e.g., "String" in `String.class` or "pkg.MyClass" in `pkg.MyClass.staticField`. The class name must be fully-qualified unless it can be referenced by its simple name without an `import` statement at the location where the annotation appears. For example, an annotation in class `C` can use the simple name of a class in `java.lang` or in the same package as `C`.

33.8.1 Limitations

If you mention a class in a Java expression, then that class must be available to the compiler. For example, if you write `@EnsuresNonNull("package1.package2.MyClass.myField.myOtherField")`, then you should pass `MyClass.java` to the compiler. Otherwise, the compiler does not know which components of the dotted name are packages, classes, and field names. In that case, you might get confusing error messages such as "Invalid 'package1.package2' because could not find class package1 in package package2".

It is not possible to write a 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.

33.9 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 subtypes of (or equal to) the annotations for those fields in 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.

33.10 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 field `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 33.10.1), 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 Section 31.30.

33.10.1 `@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 34

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 2.4.5 to correct the underlying problem.

This chapter describes what to do if the methodology of Section 2.4.5 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. (Changing the code to fix a bug is another way to prevent the Checker Framework from issuing a warning, but it is not what this chapter is about.)

You may wish to suppress checker warnings because of unannotated libraries or unannotated 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 34.1), or
- the `@AssumeAssertion` string in an `assert` message (Section 34.2).

You can suppress warnings throughout the codebase by using the following mechanisms:

- the `-AsuppressWarnings` command-line option (Section 34.3),
- the `-AskipUses` and `-AonlyUses` command-line options (Section 34.4),
- the `-AskipDefs` and `-AonlyDefs` command-line options (Section 34.5),
- the `-AskipFiles` and `-AonlyFiles` command-line options (Section 34.6),
- the `-AuseConservativeDefaultsForUncheckedCode=source` command-line option (Section 36.4),
- the `-Alint` command-line option, which enables/disables optional checks (Section 34.7),
- changing the specification of a method (Section 34.8), or
- not running the annotation processor (Section 34.9).

Some type-checkers can suppress warnings via

- checker-specific mechanisms (Section 34.10).

The rest of this chapter explains these mechanisms in turn.

You can use the `-AwarnUnneededSuppressions` command-line option to issue a warning for each `@SuppressWarnings` that does not suppress any warnings issued by the current checker.

34.1 `@SuppressWarnings` annotation

You can suppress specific errors and warnings by use of the `@SuppressWarnings` annotation, for example `@SuppressWarnings("interning")` or `@SuppressWarnings("nullness")`. Section 34.1.1 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 within that program element. Section 34.1.2 discusses where the annotation may be written in source code.

Section 34.1.3 gives best practices for writing `@SuppressWarnings` annotations.

34.1.1 `@SuppressWarnings` syntax

The `@SuppressWarnings` annotation takes a string argument, in one of the following forms: "*checkername*`:`*messagekey*", "*checkername*", or "*messagekey*".

The argument *checkername* is the checker name, without "Checker". It is lowercase by default, though a checker can choose a different casing. 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 30.1.) Sometimes, a checker honors multiple *checkername* arguments; use the `-AshowSuppressWarningsStrings` command-line option to see them.

The argument *messagekey* is the message key for the error. Each warning message from the compiler gives the most specific suppression string 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);
          ^
```

You are allowed to use any substring of a message key, so long as the substring extends at each end to a period or an end of the key. For example, to suppress a warning with message key `"lock.expression.possibly.not.final"`, you could use `@SuppressWarnings("lock.expression.possibly.not.final")`, `@SuppressWarnings("lock.expression")`, `@SuppressWarnings("not.final")`, or other variants. We recommend using the longest possible message key; a short message might suppress more warnings than you expect.

The checkername `"allcheckers"` means all checkers. Using this is not recommended, except for messages common to all checkers such as purity-related messages when using `-AcheckPurityAnnotations`. The Checker Framework never issues an `unneeded.suppression` warning about a `@SuppressWarnings` whose checkername is `"allcheckers"`.

The special messagekey "`all`" means to suppress all warnings.

If the *checkername* part is omitted, the `@SuppressWarnings` applies to all checkers. If the *messagekey* part is omitted, the `@SuppressWarnings` applies to all messages (it suppresses all warnings from the given checker).

With the `-ArequirePrefixInWarningSuppressions` command-line option, the Checker Framework only suppresses warnings when the string is in the "*checkername*" or "*checkername*`:`*messagekey*" format, as in `@SuppressWarnings("nullness")` or `@SuppressWarnings("nullness:assignment")`. For example, `@SuppressWarnings("assig` and `@SuppressWarnings("all")` have no effect (they are ignored) when `-ArequirePrefixInWarningSuppressions` is used. You can use `@SuppressWarnings("allcheckers")` to suppress all Checker Framework warnings.

34.1.2 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.

`@SuppressWarnings` cannot be used on **statements, expressions, or types**. To work around this, you can

- extract an expression into a local variable and suppress a warning on its declaration,
- write `assert` plus `@AssumeAssertion` between statements; it can affect arbitrary expressions (see Section 34.2), or
- use a type-system-specific method call, such as `NullnessUtil.castNonNull`.

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. Or, you can use `@AssumeAssertion` on an `assert` statement; see Section 34.2.

As an example, consider suppressing a warning at an assignment that you know is safe. This example uses the Tainting Checker (Chapter 12). 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;
```

Java does not permit annotations (such as `@SuppressWarnings`) on assignments (or on other statements or expressions), so it would have been *illegal* to write

```
@Untainted String myvar;
...
@SuppressWarnings("tainting:cast.unsafe") // expr is untainted because ...
myvar = expr;
```

34.1.3 Good practices when suppressing warnings

Suppress warnings in the smallest possible scope

Prefer `@SuppressWarnings` on a local variable declaration to one on a method, and prefer one on a method to one on a class. `@SuppressWarnings` on a local variable declaration applies only to the declaration (including its initializer if any), not to all uses of the variable.

You may be able to suppress a warning about a use of an expression by writing `@AssumeAssertion` for the expression, before the use. See Section 34.2.

Another way to reduce the scope of a `@SuppressWarnings` is to extract the expression into a new local variable and place a `@SuppressWarnings` annotation on the variable declaration. See Section 34.1.2.

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.

Any of the following would have suppressed the warning in Section 34.1.2:

```
@SuppressWarnings("tainting")              // suppresses all tainting-related warnings
@SuppressWarnings("cast")                  // suppresses warnings from all checkers about casts
@SuppressWarnings("unsafe")                // suppresses warnings from all checkers about unsafe code
@SuppressWarnings("cast.unsafe")           // suppresses warnings from all checkers about unsafe casts
@SuppressWarnings("tainting:cast")         // suppresses tainting warnings about casts
@SuppressWarnings("tainting:unsafe")       // suppresses tainting warnings about unsafe code
@SuppressWarnings("tainting:cast.unsafe")  // suppresses tainting warnings about unsafe casts
```

The last one is the most specific, and therefore is the best style.

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.)

A suppression message like "a.f is not null" is not useful. The fact that you are suppressing the warning means that you believe that `a.f` is not null. The message should explain *why* you believe that; for example, "a.f was checked above and no subsequent side effect can affect it".

Here are some terse examples from libraries in plume-lib:

```
@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
@SuppressWarnings("purity") // side effect to local state of type BitSet
```

A particularly good (and concise) justification is to reference an issue in the issue tracker, as in these two from Daikon:

```
@SuppressWarnings("flowexpr.parse.error") // https://tinyurl.com/cfissue/862
@SuppressWarnings("keyfor") // https://tinyurl.com/cfissue/877
```

When you encounter a false positive warning, please upvote the corresponding issue on the issue tracker, or create a new issue. This helps the Checker Framework maintainers to prioritize their work. If you reference the issue URL in your warning suppression, then you can later follow the URL to check whether an issue has been fixed (though the `-AwarnUnneededSuppressions` command-line option also serves this purpose).

34.2 `@AssumeAssertion` string in an `assert` message

Sometimes, it is too disruptive to refactor your code to create a location where `@SuppressWarnings` can be written. You can instead suppress a warning by writing an assertion whose message contains the string `@AssumeAssertion(`*`checkername`*`)`.

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 refinement (Section 33.7), if the expression appeared in a conditional test. Each type system has its own rules about what type refinement it performs.

The value in parentheses is a checker name (typically lowercase), exactly as in the `@SuppressWarnings` annotation (Section 34.1.1). Any subcheckers will also assume that the assertion is true (e.g., the Map Key Checker will assume that the assertion in the example above cannot fail, when it runs as a subchecker of the Nullness Checker).

The same good practices apply as for `@SuppressWarnings` annotations, such as writing a comment justifying why the assumption is safe (Section 34.1.3).

The `-AassumeAssertionsAreEnabled` and `-AassumeAssertionsAreDisabled` command-line options (Section 33.7.6) 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.

34.2.1 Suppressing warnings and defensive programming

This section explains the distinction between two different uses for assertions: debugging a program (also known as defensive programming) versus specifying a program. The examples use nullness annotations, but the concepts apply to any checker.

The purpose of assertions is to aid debugging by throwing an exception when a program does not work correctly. Sometimes, programmers use assertions for a different purpose: documenting how the program works. By default, the Checker Framework assumes that each assertion is used for its primary purpose of debugging: the assertion might fail at run time, and the programmer wishes to be informed at compile time about such possible run-time errors.

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.

A non-standard use for assertions is to document facts that a programmer has independently verified to be true. 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 (see Section 34.2). Only do so if you are sure that the assertion always succeeds at run time.

Methods such as `Objects.requireNonNull`, JUnit's `Assert.assertNotNull`, and Guava's `verifyNotNull` and `checkNotNull` are similar to assertions. Just as for assertions, their intended use is as debugging aids, they might fail at run time, and the Checker Framework warns if that might happen. Some programmers may use assert methods as documentation of facts that the programmer has verified in some other manner. If you know that a particular codebase always uses an assertion method not for defensive programming but to indicate facts that are guaranteed to be true (that is, these assertions cannot fail at run time), then there are two approaches to avoid false positive warnings: write specifications or suppress warnings; see below for an explanation of each approach.

The method `NullnessUtil.castNonNull` is not an assertion method. It is a warning suppression method.

Note that some libraries have an imprecise/incorrect specification of their assertion methods. For example, Guava's `Verify.verifyNotNull` is imprecisely/incorrectly specified to have a `@Nullable` formal parameter. In a correct execution, `null` never flows there, so its type can and should be annotated as `@NonNull`. That annotation allows the Nullness Checker to warn about programs that crash due to passing `null` to `verifyNotNull`. (A comment in Guava's `Preconditions.java` agrees with this reasoning: "the first parameter to `checkNotNull` *should* be annotated to require it to be non-null." The comment goes on to say "I had hoped to take a principled stand on this" (that is, write the annotation `@NonNull`), but Guava annotates it as `@Nullable` to accommodate misuses within the Google codebase.)

By default, the Checker Framework uses Guava's annotations as written. If you wish to take the principled stand in order to prevent *all* `NullPointerException`s, you can use a different version of Guava with a small change to your build file. Where the build file refers to Maven Central's `guava` artifact, change the group name from "com.google.guava" to "org.checkerframework.annotatedlib". (The code is identical; the only difference is annotations.)

Option 1: Write specifications based on uses of assertion methods Suppose you are annotating a codebase that already contains precondition checks, such as:

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

Because `key` is non-null in every correct execution, its type should be `@NonNull` in `myGet`'s signature. (`@NonNull` is the default, so in this case there is nothing to write.) The checker will not issue a warning about the `checkNotNull` call, but will issue a warning at incorrect calls to `myGet`.

Option 2: Suppress warnings at uses of assertion methods This section explains how to suppress warnings at all uses of an assertion method. As with any warning suppression, you will compromise the checker's guarantee that your code is correct and will not fail at run time.

- If the method is defined in your source code, annotate its definition just as `NullnessUtil.castNonNull` is annotated; see its Javadoc or 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 the method is defined in an external library, write a stub file that changes the method's annotations, or use `-AskipUses` to make the Checker Framework ignore all calls to an entire class.
  As a special case, if you want the Nullness Checker to prevent most null pointer exceptions in your code, but to *permit* null pointer exceptions at nullness assertion methods, you can pass `-Astubs=permit-nullness-assertion-exception.astu`

34.3 `-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 strings, 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.

34.4 `-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`, `-AonlyDefs`, `-AskipFiles`, or `-AonlyFiles` command-line option; see Section 34.5). You can also use these options to affect entire packages or directories/folders.

Set the `-AskipUses` command-line option to a regular expression that matches fully-qualified 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 fully-qualified class names (not file names) for which warnings and errors should be emitted; warnings about uses of all other classes will be suppressed. The regular expressions are unanchored, unless you anchor them yourself with "`^`" and/or "`$`".

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. Often, `-AskipUses` is more useful.

34.5 `-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 Section 34.4.) You can also use these options to affect entire packages or directories/folders.

Set the `-AskipDefs` command-line option to a regular expression that matches fully-qualified 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 fully-qualified class names (not file names) whose definitions should be type-checked. (This is somewhat similar to NullAway's `-XepOpt:NullAway:AnnotatedPackages` command-line argument.) (For file names, use `-AskipFiles` and `-AonlyFiles` instead; see Section 34.6.) The regular expressions are unanchored, unless you anchor them yourself with "^" and/or "$".

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 these options 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.

34.6 `-AskipFiles` and `-AonlyFiles` command-line options

You can suppress all errors and warnings originating from classes that are located in a given file or directory/folder, or all those that are *not* in a given file or directory/folder.

Set the `-AskipFiles` command-line option to a regular expression that matches the path to files or directories/folders containing classes for which errors and warnings should be suppressed. Similarly, `-AonlyFiles` suppresses warnings everywhere *except* in the given files. The regular expressions are unanchored, unless you anchor them yourself with "^" and/or "$".

For example, if you use "`-AskipFiles=/build/generated/`" on the command line when invoking `javac`, then the definitions of classes located in any `/build/generated` directory will not be checked.

If you supply both `-AskipFiles` and `-AonlyFiles`, then `-AskipFiles` takes precedence.

A common scenario for using these options is when you want to exclude certain directories/folders (such as those containing generated or legacy code) from type-checking. Another scenario is when a file contains multiple top-level classes, only one of which has a name that corresponds to the file name.

34.7 `-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 (and possibly more, see the checker's documentation):

- `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 because of type refinement (Section 33.7).
- `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 2) and `dotequals` for the Interning Checker (see Section 6.4). 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 a check name 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`.

34.8 Change the specification of a method

To prevent a checker from issuing a warning at calls to a specific method, you can change the annotations on that method by writing a stub file (see Section 36.5).

Stub files are usually used to provide correct specifications for unspecified code.

Stub files can also be used to provide *incorrect* specifications, for the purpose of suppressing warnings. For example, suppose that you are running the Nullness Checker to prevent null pointer exceptions. Further suppose that for some reason you do not care if method `Objects.requireNonNull` crashes with a `NullPointerException`. You can supply a stub file containing:

```
package java.util;
class Objects {
  @EnsuresNonNull("#1")
  public static <T> @NonNull T requireNonNull(@Nullable T obj);
}
```

This particular stub file already exists. If you want the Nullness Checker to prevent most null pointer exceptions in your code, but to *permit* null pointer exceptions at nullness assertion methods, you can pass `-Astubs=permit-nullness-assertion-exc`

34.9 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.

34.10 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 20.5).

Chapter 35

Type inference

This chapter is about tools that infer annotations for your program's method signatures and fields, before you run a type-checker. To learn about local type inference within a method body, see Section 33.7.

A typical workflow (Section 2.4) is for a programmer to first write annotations on method signatures and fields, then run a type-checker. Type inference performs the first step automatically for you. This saves time for programmers who would otherwise have to understand the code, then write annotations manually.

Type inference outputs type qualifiers that are consistent with your program's source code. Your program still might not type-check if your program contains a bug or contains tricky code that is beyond the capabilities of the type-checker.

The qualifiers are output into an annotation file. They can be viewed and adjusted by the programmer, can be used by tools such as the type-checker, and can be inserted into the source code or the class file.

Inserting the inferred annotations into the program source code creates documentation in the form of type qualifiers, which can aid programmer understanding and may make type-checking warnings more comprehensible. Storing annotations in side-files is more desirable if the program's source code cannot be modified for some reason, if the type-checking is "one-off" (type-checking will be done once and its results will be evaluated, but it will not be done repeatedly), or if the set of annotations is extremely voluminous and would clutter the code.

Type inference is most effective when you run it on a program rather than on a library — unless you also run it on an extensive test suite for the library. See Section 35.7 for an explanation.

Type inference is costly: it takes several times longer than type-checking does. However, it only needs to be run once, after which you can use and possibly modify the results.

35.1 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.

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

For the Purity 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 33.7.5.

WPI, for any type system: "Whole program inference", or WPI, is distributed with the Checker Framework. See Section 35.2.

CFI, for any type system: "Checker Framework Inference", or CFI, is a type inference framework built on a variant of the Checker Framework. You need to slightly rewrite your type system to work with CFI. The CFI repository (`https://github.com/opprop/checker-framework-inference`) contains rewritten versions of some of the type systems that are distributed with the Checker Framework.

Cascade, for any type system: Cascade [VPEJ15] 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`. Cascade was last updated in November 2014, so it might or might not work for you.

Except for one of the nullness inference tools, all these type inference tools are static analyses. They analyze your program's source code, but they do not run your program.

35.2 Whole-program inference

Whole-program inference infers types for fields, method parameters, and method return types that do not have a user-written qualifier (for the given type system). The inferred type qualifiers are output into annotation files. 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.

There are four scripts that you can use to run whole-program inference. Each has advantages and disadvantages, discussed below:

- To run whole-program inference on a single project, with a few changes to its buildfile but no changes to its source code, use the `wpi2.sh` script (Section 35.3).
- To run whole-program inference on a single project without modifying either its buildfile or its source code, use the `wpi.sh` script (Section 35.4). This script can automatically understand many Ant, Maven, and Gradle build files, so it requires little manual configuration. However, it is based on a no-longer-maintained project, `dljc`, so it is not guaranteed to work.
- To run whole-program inference on many projects without modifying their source code (say, when running it on projects from GitHub), use the `wpi-many.sh` script (Section 35.5). This script can understand the same build files as `wpi.sh`.
- If you want to insert the inferred annotations directly into a single project's source code, use the `infer-and-annotate.sh` script (Section 35.6).

These type inference scripts appear in the `checker/bin/` directory. The remainder of this chapter describes them (Sections 35.3–35.6), then concludes with discussion that applies to all of them.

35.3 Running whole-program inference on a single project, with buildfile editing

You need to edit your buildfile, then run the `wpi2.sh` script.

35.3.1 Editing your buildfile

First, set up your buildfile to run the Checker Framework. If you use the Checker Framework Gradle plugin (`https://github.com/typetools/checker-framework-gradle-plugin`), then there is nothing to do.

Otherwise, add these command-line arguments:

- `-Ainfer=ajava`
- `-AinferOutputDirectory=.../whole-program-inference-new`
- `-Aajava=.../whole-program-inference-output`
- `-Awarns`

and remove or comment out `-Werror` and `-AinferOutputOriginal`. Here is a Gradle example:

```
checkerFramework {
  extraJavacArgs = [
    "-Ainfer=ajava",
    "-AinferOutputDirectory=" + project.rootDir + "/whole-program-inference-new",
    "-Aajava=" + project.rootDir + "/whole-program-inference-output",
    "-Awarns",
    // "-Werror", // remove or comment out any occurrence of "-Werror".
    // "-AinferOutputOriginal" // remove or comment out any occurrence.
    ...
  ]
  ...
}
```

The two directories must be named `whole-program-inference-new` and `whole-program-inference-output` and must be absolute paths. They must also be the same for every subproject of a multi-project build, which is why the example uses `project.rootDir` rather than `project.projectDir`.

Put the directories at the top level of your project. Do not put either directory under your build system's output directory, such as `build/` or `target/`. Your build system's `clean` task would delete the inference output, possibly in the middle of a build.

35.3.2 Running `wpi2.sh`

Run `wpi2.sh` from your project's top-level directory — the same directory that contains the `whole-program-inference-*` directories that you configured your buildfile to use.

The arguments to `wpi2.sh` are a command that runs the Checker Framework. Here is a typical invocation (if you use the Checker Framework Gradle plugin (`https://github.com/typetools/checker-framework-gradle-plugin`)).

```
$CHECKERFRAMEWORK/checker/bin/wpi2.sh ./gradlew -Pwpi2 clean assemble
```

The `clean` task is needed because `wpi2.sh` runs the Checker Framework repeatedly, and Gradle does nothing if the old (but still up-to-date) compilation output is present. If the given command does not recompile every source file, then `wpi2.sh` halts with an error.

The output of `wpi2.sh` is a top-level directory `whole-program-inference-output`. For each of your project's source files for which the Checker Framework inferred an annotation, it contains a file *`package/ClassName-checkername`*`.ajava`, where *checkername* is the fully-qualified name of the checker that ran. A source file for which nothing was inferred has no `.ajava` file. `wpi2.sh` uses three other top-level directories: `whole-program-inference-new`, which the compiler writes each iteration's results to and which `wpi2.sh` removes when inference converges; `whole-program-inference-diffs`, which contains one diff file per iteration (`iteration-1.diff`, `iteration-2.diff`, etc.) for diagnostic purposes; and `whole-program-inference-previous`, which exists only while each iteration's results replace the previous iteration's. You may wish to add `whole-program-inference-*` to your `.gitignore` file.

`wpi2.sh` exits with status 0 if inference converged, and with a nonzero status otherwise. In particular, it exits with status 1 if the Checker Framework inferred no annotations at all — most often because the buildfile is misconfigured, but also if your project is already fully annotated.

If `wpi2.sh` fails — for example, because the Checker Framework crashed or your buildfile is misconfigured — then a subsequent run of `wpi2.sh` continues from the annotations that it inferred so far in `whole-program-inference-output`. To start inference from scratch, remove `whole-program-inference-output` before running `wpi2.sh`.

`wpi2.sh` runs the command at most 10 times. Set the `WPI2_MAX_ITERATIONS` environment variable to an integer 2 or greater to change the bound.

35.4 Running whole-program inference on a single project, without buildfile editing

A typical invocation of `wpi.sh` is

```
$CHECKERFRAMEWORK/checker/bin/wpi.sh -- --checker nullness
```

The result is a set of log files placed in the `dljc-out/` folder of the target project. See Section 35.4.1 for an explanation of the log files generated by an invocation of `wpi.sh`. The inferred annotations appear in `.ajava` files in a temporary directory whose name appears in the `dljc-out/wpi-stdout.log` file; you can find their location by examining the `-Aajava` argument to the last `javac` command that was run. The annotation files generated in each round of inference appear in directories labeled `iteration0`, `iteration1`, etc.

The `wpi.sh` script is most useful when analyzing projects that follow the standard conventions of their build system for single-module projects. When analyzing a project that requires non-standard build system commands, use the `-c` and `-b` options to override the defaults used by `wpi.sh`. For example, suppose that you wanted to use `wpi.sh` to infer annotations for the Resource Leak Checker on Apache ZooKeeper's `zookeeper-server` module only. Without `wpi.sh`, one might use the following command: `mvn -B -projects zookeeper-server -also-make install -DskipTests`. To use this command via `wpi.sh`, use the following: `sh wpi.sh -b "-B -projects zookeeper-server -also-make" -c "install -DskipTests" - -checker resourceleak`.

The full syntax for invoking `wpi.sh` is

```
wpi.sh [-d PROJECTDIR] [-t TIMEOUT] [-c COMPILATION_TARGET] [-b EXTRA_BUILD_ARGS] [-g GRADLECACHEDIR] --
```

Arguments in square brackets are optional. Here is an explanation of the arguments:

-d PROJECTDIR The top-level directory of the project. It must contain an Ant, Gradle, or Maven buildfile. The default is the current working directory.

-t TIMEOUT The timeout for running the checker, in seconds.

-c COMPILATION_TARGET The name(s) of the build system target(s) used to compile the target project. The default is chosen based on the build system (e.g., `compileJava` for Gradle, `compile` for Maven, etc.). This argument is passed directly to the build system when compiling the target project, so it can also include command-line arguments that only apply to the compilation targets (and not other targets like `clean`); for general command-line arguments that apply to all targets, use the `-b` option instead. When running WPI on a single module of a multi-module project, you might want to use this option (possibly in combination with `-b`, below), depending on the setup of the target project. The argument may contain spaces and is re-tokenized by the shell.

-b EXTRA_BUILD_ARGS Extra arguments to pass to the build script invocation. This argument will be passed to compilation tasks, such as `ant compile`, `gradle compileJava`, or `mvn compile`. The main difference between this option and `-c` is that the values of the `-b` option are also passed to other, non-compilation build system commands, such as `ant clean`, `gradle clean`, or `mvn clean`. The argument may contain spaces and is re-tokenized by the shell.

-g GRADLECACHEDIR The directory to use for the `-g` option to Gradle (the Gradle home directory). This option is ignored if the target project does not build with Gradle. The default is `.gradle` relative to the target project (i.e., each target project has its own Gradle home). This default is motivated by Gradle issue #1319.

DLJC-ARGS Arguments that are passed directly to do-like-javac's `dljc` program without modification. One argument is required: `--checker`, which indicates what type-checker(s) to run (in the format described in Section 2.2.4). The documentation of do-like-javac (`https://github.com/kelloggm/do-like-javac`) describes the other commands that its WPI tool supports. Notably, to pass checker-specific arguments to invocations of `javac`, use the `--extraJavacArgs` argument to `dljc`. For example, to use the `-AignoreRangeOverflow` option for the Constant Value Checker (Chapter 24, page 161) when running inference, you would add `--extraJavacArgs='-AignoreRangeOverflow'` anywhere after the `--` argument to `wpi.sh`.

You may need to wait a few minutes for the command to complete.

35.4.1 Whole-program inference results

The results of invoking `wpi.sh` are log files stored in the `dljc-out/` folder of the target project. The `dljc-out/` folder contains:

- `typecheck.out`: The final error messages produced by `javac`, comprising the results of type-checking obtained with the latest iteration of WPI (i.e., using the most precise, consistent set of annotations).
- `build_output.txt`: Logs collected from compiling the target project without the Checker Framework, using the build file found at the top-level directory.
- `wpi-stdout.log`: The results of type-checking with each candidate set of annotations, concatenated together in the order in which the annotations were inferred. The final results (i.e., those obtained using the most precise, consistent set of annotations) will appear at the end of this file.
- `wpi-stdout-*`: These files are separate from the logs in `wpi-stdout.log`. They may be useful in the case where `dljc` is not working as expected.
- `toplevel.log`: The log of the command executed at the top-level of the target directory to invoke whole-program inference.
- `javac.json`: The list of all `.java` files for which inference was attempted by `wpi.sh`, and the options passed to `javac`.
- `stats.json`: Statistics from the invocation of `wpi.sh` on the target project, including build time, the number of built `.jar` files, the number of executable `.jar` files, the number of `javac` invocations, and the number of source files.

35.4.2 Requirements for whole-program inference scripts

The requirements to run `wpi.sh` and `wpi-many.sh` are the same:

- The project on which inference is run must contain an Ant, Gradle, or Maven buildfile that compiles the project.
- At least one of the `JAVA_HOME`, `JAVA8_HOME`, `JAVA11_HOME`, `JAVA17_HOME`, `JAVA21_HOME`, `JAVA24_HOME`, `JAVA25_HOME`, or `JAVA26_HOME` environment variables must be set.
- If set, the `JAVA_HOME` environment variable must point to a Java JDK.
- If set, the `JAVA8_HOME` environment variable must point to a Java 8 JDK.
- If set, the `JAVA11_HOME` environment variable must point to a Java 11 JDK.
- If set, the `JAVA17_HOME` environment variable must point to a Java 17 JDK.
- If set, the `JAVA21_HOME` environment variable must point to a Java 21 JDK.
- If set, the `JAVA24_HOME` environment variable must point to a Java 24 JDK.
- If set, the `JAVA25_HOME` environment variable must point to a Java 25 JDK.
- If set, the `JAVA26_HOME` environment variable must point to a Java 26 JDK.
- The `CHECKERFRAMEWORK` environment variable must point to a built copy of the Checker Framework.
- If set, the `DLJC` environment variable must point to a copy of the `dljc` script from do-like-javac (`https://github.com/kelloggm/do-like-javac`). (If this variable is not set, the WPI scripts will download this dependency automatically.)
- Other dependencies: ant, awk, curl, git, gradle, mvn, python3 (for dljc), wget.
  Python 2.7 modules: subprocess32.

35.5 Running whole-program inference on many projects

The requirements to run `wpi.sh` and `wpi-many.sh` are the same. See Section 35.4.2 for the list of requirements.

To run an experiment on many projects:

1. Use `query-github.sh` to search GitHub for candidate repositories. File `docs/examples/wpi-many/securerandom.query` is an example query, and file `docs/examples/wpi-many/securerandom.list` is the standard output created by running `query-github.sh securerandom.query 100`. If you do not want to use GitHub, construct a file yourself that matches the format of the file `securerandom.list`.

2. Use `wpi-many.sh` to run whole-program inference on multiple Ant, Gradle, or Maven projects. You provide it a file in which each line is "GitHub repository URL" "git hash" (with no quotes, and with whitespace between them).
    - If you are using a checker that is distributed with the Checker Framework, use `wpi-many.sh` directly.
    - If you are using a checker that is not distributed with the Checker Framework (also known as a "custom checker"), file `docs/examples/wpi-many/wpi-many-custom-checker-example.sh` is a no-arguments script that serves as an example of how to use `wpi-many.sh`.

    Log files are copied into a results directory. For a failed run, the log file indicates the reason that WPI could not be run to completion on the project. For a successful run, the log file indicates whether the project was verified (i.e., no errors were reported), or whether the checker issued warnings (which might be true positive or false positive warnings).
3. Use `wpi-summary.sh` to summarize the logs in the output results directory. Use its output to guide your analysis of the results of running `wpi-many.sh`: you should manually examine the log files for the projects that appear in the "results available" list it produces. This list is the list of every project that the script was able to successfully run WPI on. (This does not mean that the project type-checks without errors afterward, or even type-checks at all — just that the Checker Framework attempted to type-check the project and some output was produced.)
4. (Optional) Add annotations that WPI does not infer, to eliminate false positive warnings. There are two ways you can add annotations to a target program:
    - Fork the project and add the annotations directly to the project's source code, and add a dependency on `org.checkerframework:checker-qual` to the project's build system. This approach is the most difficult, but has the advantage that the checker will attempt to verify any annotations you add.
    - Add the annotations in a stub file, creating the `.astub` file as a copy of the `.java` file. When you run the checker, supply the `-AmergeStubsWithSource` and `-Astubs=...` command-line arguments.

A typical invocation is

```
wpi-many.sh -o outdir -i /path/to/repo.list -t 7200 -- --checker optional
```

The `wpi-many.sh` script takes the following command-line arguments. The `-o` and `-i` arguments are mandatory. An invocation should also include `-- [`*`DLJC-ARGS`*`]` at the end; ***DLJC-ARGS*** is documented in Section 35.4.

-o outdir Run the experiment in the *`outdir`* directory, and place the results in the *`outdir`*`-results` directory. Both will be created if they do not exist. The directory may be specified as an absolute or relative path.

-i infile Read the list of repositories to use from the file infile. The file must be specified as an absolute, not relative, path. Each line should have 2 elements, separated by whitespace:

1. The URL of the git repository on GitHub. The URL must be of the form https://github.com/username/repository . The script is reliant on the number of slashes, so excluding "https://" is an error.
2. The commit hash to use.

-t timeout The timeout for running the checker on each project, in seconds.

-g GRADLECACHEDIR The directory to use for the `-g` option to Gradle (the Gradle home directory). This option is ignored if the target project does not build with Gradle. The default is `.gradle` relative to the target project (i.e., each target project has its own Gradle home). This default is motivated by Gradle issue #1319.

-s If this flag is present, then projects that are not buildable — for which no supported build file is present or for which running the standard build commands fails — are skipped on future runs but are *not* deleted immediately (such projects are deleted immediately if this flag is not present). This flag is useful if you intend to run `wpi-many.sh` several times on the same set of repositories (for example, during checker development), to avoid re-downloading unusable projects.

To obtain the locations of the annotation files generated by an invocation of `wpi-many.sh`, you can run `wpi-annotation-paths.sh` on the result directory, e.g., *`outdir`*`-results`.

A typical invocation is

```
wpi-annotation-paths.sh outdir-results
```

Here, the results of `wpi-many.sh` are located in *`outdir`*`-results`.

35.6 Whole-program inference that inserts annotations into source code

To use this version of whole-program inference, make sure that `insert-annotations-to-source`, from the Annotation File Utilities project, is on your path (for example, its directory is 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
$CHECKERFRAMEWORK/checker/bin/infer-and-annotate.sh \
    "LockChecker,NullnessChecker" java/plume.jar:java/lib/junit-4.12.jar:$JAVA_HOME/lib/tools.jar \
    $(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 it tries 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.

35.7 Inference results depend on uses in your program or test suite

Type inference outputs the most specific type qualifiers that are consistent with all the source code it is given. (Section 35.7.1 explains when type inference ignores some code.) This may be different than the specification the programmer had in mind when writing the code. If the program uses a method or field 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 (or test suite) 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 inferred `@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`.

- Suppose that method `m2` has no body, because it is defined in an interface or abstract class. Type inference can still infer types for its signature, based on the overriding implementations. If all the methods that override `m2` return a non-null value, type inference will infer that `m2`'s return type has `@NonNull` type, even if some other overriding method is allowed to return `null`.

If the program contains erroneous calls, the inferred annotations may reflect those errors. Suppose you intend method `m3` to be called with non-null arguments, but your program contains an error and one of the calls to `m3` passes `null` as the argument. Then the tool will infer that `m3`'s parameter has `@Nullable` type.

If you run whole-program inference on a library that contains mutually recursive routines, and there are no non-recursive calls to the routines, then whole-program inference may run a long time and eventually produce incorrect results. In this case, write type annotations on the formal parameters of one of the routines.

Whole-program inference is a "forward analysis". It 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. It determines a method's return type based on code in the method body but not on uses of the method return value in client code.

35.7.1 Whole-program inference ignores some code

Whole-program inference ignores code within the scope of a `@SuppressWarnings` annotation with an appropriate key (Section 34.1). 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 above, whole-program inference generalizes from invocations of methods and assignments to fields. If a field is set via reflection (such as via injection), there are no explicit assignments to it for type inference to generalize from, and type inference will 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 assignments and pseudo-assignments where the right-hand side is the `null` literal.

35.7.2 Manually checking whole-program inference results

With any type inference tool, it is a good idea to manually examine the results. This can help you find bugs in your code or places where type inference inferred an overly-precise result. You can correct the inferred results manually, or you can add tests that pass additional values and then re-run inference.

When arguments or assignments are literals, whole-program inference commonly infers overly precise type annotations, such as `@Interned` and `@Regex` annotations when the analyzed code only uses a constant string.

When an annotation is inferred for a *use* of a type variable, you may wish to move the annotation to the corresponding upper bounds of the type variable *declaration*.

35.8 How whole-program inference works

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

Each script repeatedly runs the checker with an `-Ainfer=` command-line option to infer types for fields and method signatures. The output of this step is a `.jaif` (for `infer-and-annotate.sh`) or `.ajava` (for `wpi.sh`) file that records the inferred types. Each script adds the inferred annotations to the next run, so that the checker takes them into account (and checks them). `wpi.sh` does this by updating the set of `.ajava` files that are passed to the checker via the `-Aajava` command-line argument; `infer-and-annotate.sh` inserts the inferred annotations in the program using the Annotation File Utilities (`https://checkerframework.org/annotation-file-utilities/`).

On each iteration through the process, there may be new annotations in the `.jaif` or `.ajava` files, and some type-checking errors may be eliminated (though others might be introduced). The process halts when there are no more changes to the inference results, that is, the `.jaif` or `.ajava` files are unchanged between two runs.

When the type-checker is run on the program with the final annotations inserted, there might still be errors. This may be because the tool did not infer enough annotations, or because your program cannot type-check (either because it contains a defect, or because it contains subtle code that is beyond the capabilities of the type system). However, each of the inferred annotations is sound, and this reduces your manual effort in annotating the program.

The iterative process is required because type-checking is modular: it processes each class and each method 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.
- Revisiting an already-processed class may result in a better estimate.

35.9 Type inference compared to other whole-program analyses

There exist monolithic whole-program analyses that run without requiring any annotations in the source code. An advantage of such a tool is that the programmer never needs to write any type annotations.

Running a whole-program inference tool, then running a type-checker, has some benefits:

- The type qualifiers act as machine-checked documentation, which can aid programmer understanding.
- Error messages may be more comprehensible. With a monolithic whole-program analysis, error messages can be obscure, because the analysis has already inferred (possibly incorrect) types for a number of variables.
- Errors are localized. A change to one part of the program does not lead to an error message in a far-removed part of the program.
- Type-checking is modular, which can be faster than re-doing a whole-program analysis every time the program changes.

Chapter 36

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 find and 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 33.5.6 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, the `-AskipUses` or `-AonlyUses` command-line option can suppress all warnings related to an unannotated library, or to part of your codebase; see Sections 34.4–34.5.)

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 39, page 279). Now, when you compile the library, the resulting `.jar` file contains type annotations.
   When checking a client of the library, 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 36.2.
2. Write annotations in a "stub file", if you do not have access to the source code.
   Then, when checking a client of the library, 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 class or method).
   For instructions, see Section 36.5.

If you annotate a new library (either in its source code or in a stub file), please inform the Checker Framework developers so that your annotated library can be advertised to users of 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 36.1.4, you should annotate an entire class at a time. You may find type inference tools (Chapter 35, page 230) helpful when getting started, but you should always examine their results.

36.1 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.

36.1.1 Don't change the code

If you annotate a library that you maintain, you can refactor it to improve its design.

When you annotate a library that you do not maintain, you should only add annotations and, when necessary, documentation of those annotations. You can place the documentation in a Java comment (`//` or `/*...*/`) or in a `@CFComment` annotation.

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.

Any of these changes would increase the difference between upstream (the original version) and your annotated version. Unnecessary differences make it harder for others to understand what you have done, and they make it harder to pull changes from upstream into the annotated library.

36.1.2 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 or permits null. (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.

36.1.3 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 pull request 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.

36.1.4 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. Here are reasons for this rule:

- If you annotate just part of the file, then users may be surprised that calls to some methods type-check as expected whereas other methods do not (because they have not been annotated).
- Annotating one method or field at a time may lead to inconsistencies between different parts of the file. Different people may make different assumptions, might write annotations in a way that is locally convenient but globally inconsistent, or might not read all the documentation of the class to understand how it works.
- It is only a bit more effort to annotate an entire class than to annotate one method or field. In either case it is usually necessary to understand the entire class's design and implementation. Once you have done that, you might as well annotate the whole thing.
- If you fully annotate the file, it is possible to type-check the library to verify the annotations. (Even if you do not do this right now, it eases the task in the future.)

36.1.5 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. 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; you can supply `-Awarns` so that the compiler will still produce `.class` files.

36.2 Creating an annotated library

This section describes how to create an annotated library.

1. See the `org.checkerframework.annotatedlib` group in the Maven Central Repository to find out whether an annotated version of the library already exists.
   - If it exists, but you want to add annotations for a different checker:
     Clone its repository from `https://github.com/typetools/`, and tweak its buildfile to run an additional checker.
   - If it does not already exist:
     Fork the project. (First, verify that its license permits forking.)
     Add a line in its main README to indicate that this is an annotated version of the library. That line should also indicate how to obtain the corresponding upstream version (typically a git tag corresponding to a release), so that others can see exactly what edits you have made.
     Adjust the library's build process, such as an Ant, Gradle, or Maven buildfile (for guidance, see Chapter 39, page 279). Every time the build system runs the compiler, it should:
     - pass the `-AuseConservativeDefaultsForUncheckedCode=source,bytecode` command-line option and
     - run every pluggable type-checker for which any annotations exist, using `-processor` *`TypeSystem1,TypeSystem2,`*

     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.
2. Annotate some files.
   When you annotate a file, annotate the whole thing, not just a few of its methods. Add an `@AnnotatedFor({"`*`checkername`*`"})` annotation to each class declaration, 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 a pull request or 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.
- Optionally, encourage the library's maintainers to accept your annotations into its main version control repository. This suggestion will be most compelling if you have already reported bugs in the library that were revealed by pluggable type-checking. Once the annotations are in the main version control repository, they will be easier to maintain, the library will obtain the correctness guarantees of pluggable type-checking, and there will be no need to distribute 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 a pull request.

36.3 Creating an annotated JDK

When you create a new checker, you need to also supply annotations for parts of the JDK. You can do so either as stub files or in a copy of the JDK source code, as described in Section 37.10. Section 41.2.1 tells how to improve JDK annotations for an existing type system.

36.4 Compiling partially-annotated libraries

If you *completely* annotate a library, then you can compile it using a pluggable type-checker. You get a guarantee that the library contains no errors (that is, it is consistent with the specifications you wrote).

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 36.1.5.)

Here are two concerns you may have when partially annotating a library:

- Ignoring type-checking errors in unannotated parts of the library. Use the `-AskipDefs`, `-AonlyDefs`, `-AskipFiles`, or `-AonlyFiles` command-line arguments; see Section 34.5.
- Putting conservative annotations in unannotated parts of the library. The checker needs to use normal defaulting rules (Section 33.5.3) for code you have annotated and conservative defaulting rules (Section 33.5.6) for code you have not yet annotated. This section describes how to do this. You use `@AnnotatedFor` to indicate which classes you have annotated.

36.4.1 The `-AuseConservativeDefaultsForUncheckedCode=source,bytecode` command-line argument

When compiling a library that is not fully annotated, use command-line argument `-AuseConservativeDefaultsForUncheckedCode=source,bytecode`. This has the following effect. For classes with `@AnnotatedFor` for the current checker, the checker applies defaults normally. For classes without `@AnnotatedFor` for the current checker, the checker uses conservative defaults (see Section 33.5.6) for any type use with no explicit user-written annotation, *and* the checker issues no warnings.

For example, an unannotated method

```
String concatenate(String p1, String p2)
```

in a class *with* a relevant `@AnnotatedFor` is defaulted normally.

In a class *without* a relevant `@AnnotatedFor`, it is interpreted as

```
@MyTopQualifier String concatenate(@MyBottomQualifier String p1, @MyBottomQualifier String p2)
```

where `@MyTopQualifier` is the top qualifier in the hierarchy and `@MyBottomQualifier` is the bottom qualifier in the hierarchy. These choices are conservative. They are likely to cause many false-positive type-checking errors in client code, which will help you to know which library methods need annotations. You can then write those library annotations.

The `@AnnotatedFor` annotation, written on a class, indicates that the class has been annotated for certain type systems. You may write `@AnnotatedFor` on a class that contains no explicit annotations from the given type system, if the defaulting rules already give appropriate qualifiers for every type in the class.

For example, `@AnnotatedFor({"nullness", "regex"})` means that the programmer has written annotations for the Nullness and Regular Expression type systems. For any type system other than Nullness or Regex, that `@AnnotatedFor` annotation has no effect — the checker treats the class as unannotated for that type system. The argument to `@AnnotatedFor` is not an annotation name, but a checker name.

Whenever you compile a class using the Checker Framework, including when using the `-AuseConservativeDefaultsForUncheckedCode=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.

36.5 Using stub classes

You use a "stub file" to write annotations for a library, when you cannot edit and recompile the library. A checker uses the annotated signatures at compile time, instead of or in addition to annotations that appear in the library's `.class` files.

A stub file cannot override the types of a package-private class, method, or field.

A stub class is Java source code that is allowed to omit certain parts, such as method bodies. Section 36.5.4 describes the stub file format.

Section 36.3 explains how you should choose between creating stub classes and creating an annotated library. Section 36.5.5 describes how to create stub classes. Section 36.5.1 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.

There are two types of annotation files, which are files containing annotations that can be read by the Checker Framework. This section describes stub files, which are used by programmers and type system designers. Section 36.6 describes ajava files, which are used by tools, such as type inference.

36.5.1 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=path/to/String.astub:stubs MyFile.java MyOtherFile.java ...
```

Each stub path entry is a stub file (ending with `.astub`), directory, or `.jar` file; specifying a directory or `.jar` file 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 some of its own stub files, even without a `-Astubs` command-line argument; see Section 37.10.

User-supplied stub files override a checker's built-in stub files and the annotated JDK.

36.5.2 Multiple specifications for a method

There are three possible sources of information for a given element: source files, stub files, and bytecode files. Usually source files take precedence over the other two, and stub files take precedence over bytecode. In other words:

- If file `A.java` is being compiled, then by default any stub for class `A` is ignored. See below for how to change this behavior and respect both types of annotations.
- An unannotated type variable in a stub file is used instead of annotations on a type variable in bytecode. Use the `-AstubWarnIfRedundantWithBytecode` command-line option to get a warning whenever a stub file specification is redundant with bytecode annotations.
- If a stub file does not mention a method or field (and no source is available), its annotations are taken from bytecode.

If a method appears in more than one stub file (or twice in the same stub file), then, for each type hierarchy, the last annotation read is used.

The annotated JDK is read as a stub file. You can override JDK annotations by providing your own stub file. If your stub file contains a JDK method `m`, then no type annotations from the JDK's `m` are used. If the JDK's `m` contains a declaration annotation `@D`, it is used unless your stub file contains a different `@D` annotation for `m`.

The command-line option `-AmergeStubsWithSource` tells the checker to use both source files and stub files. The checker permits only values that are permitted by *both* the source and stub annotations. (This is called the greatest lower bound, or GLB, of the types.)

As an example of GLB, suppose the source annotation says the value is in the range [1..20] and the stub file says the value is in the range [11..30]. Since both sources of information are trusted, the value must be in the range [11..20]. Equivalently, the GLB of `@IntRange(from=1, to=20)` and `@IntRange(from=11, to=30)` is `@IntRange(from=11, to=20)`. As another example, the GLB of `@KeyFor({"map1", "map2"})` and `@KeyFor({"map2", "map3"})` is `@KeyFor({"map1", "map2", "map3"})` since the value is known to be a key for all three maps.

36.5.3 Stub methods in subclasses of the declaring class

Sometimes, a method's return type is different in a subclass than in a superclass, even though the subclass inherits (does not define) the method. An example, `SecureRandom.nextInt()`, is shown below.

To express that method `m` has a different type in subclass `Sub` than where `m` is defined, write a "fake override": write the method in class `Sub` in a stub file. The Checker Framework will use that type for `m` at calls where the receiver's declared type is `Sub` or lower.

In a fake override of `m`, the formal parameter Java types of the method must be identical to those where `m` is defined, but you can change its annotations. (The return type and the receiver type can be different.) You may write the name of a formal parameter type in fully-qualified, partially-qualified, or simple form, and you may omit its type arguments. If a fake override has no annotations on a given type use, that type use is defaulted according to the usual defaulting rules.

This feature currently only works for return types. That is, a fake override changes the return type at certain calls, but formal parameter type annotations on fake overrides have no effect. Type annotations on fake overrides of fields also have no effect. Fake overrides will be able to change formal parameter and field type annotations in a future release of the Checker Framework.

Example As an example, consider a type system that tracks whether a value is cryptographically secure. `@Secure` is a subtype of `@Insecure`. Although `SecureRandom` does not override `Random.nextInt` (it overrides the source of randomness instead), you are allowed to write the following stub file:

```
package java.util;
class Random {
  @Insecure int nextInt();
}

package java.security;
class SecureRandom extends Random {
  @Secure int nextInt();
}
```

Client code behaves as follows:

```
Random r = ...;
SecureRandom sr = ...;

@Secure int i1 = r.nextInt(); // error
@Secure int i2 = sr.nextInt(); // OK
```

36.5.4 Stub file format

Every Java file is a valid stub file. However, you can omit information that is not relevant to pluggable type-checking, as explained below; this makes the stub file smaller and easier for people to read and write. The stub file's extension must be `.astub`, not `.java`.

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 the library's `.class` files. If you are using a stub class, then often the library is unannotated, but you can use stub files to override library annotations.

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.

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.

Import statements: Imports may appear at the beginning of the file or after any package declaration. An import applies to the whole file. The only required import statements are the ones to import type annotations. Import statements for types are optional. If you omit an import statement for an annotation, occurrences of that annotation are silently ignored; this is a common error in stub files.

When you create a stub file, it is recommended that you validate it by copying the stub file to a file with the `.java` extension and running the type-checker. This will ensure that your annotations are syntactically valid, are written in the correct locations, and are properly imported. 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 a stub file for a class in the JDK, then you should use the `-Aignorejdkastub` command-line option.

36.5.5 Creating a stub file

If you have access to the Java source code

Every Java file is a stub file. If you have access to the Java file, copy file `A.java` to `A.astub`. You can 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.

This copy-and-annotate 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 the copy-and-annotate 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.

You can minimize source files to speed up parsing them. Use the `JavaStubifier` to convert, in-place, all `.java` files in given directories into minimal stub files.

```
mkdir project-stubs
cp -R project/src project-stubs
java -cp $CHECKERFRAMEWORK/checker/dist/checker.jar org.checkerframework.framework.stub.JavaStubifier pr
find project-stubs -type f -name "*.java" -exec rename 's/.java$/.astub/' {} \;
```

Supply it a list of directories to process. Replacement happens in-place, so make sure to process a copy of your sources.

You can now provide `project-stubs` as a stub directory using `-Astubs=project-stubs` as an additional command-line option.

If you wish to use a single stub file, rather than a stub directory, you can concatenate all the minimized `.java` files into a single `.astub` file:

```
find project-stubs/ -name "*.java" -type f | xargs cat > project.astub
```

If you do not have access to the Java source code

If you do not have access to the library source code, you can create a stub file by hand.

If you have access to the API documentation for the element you wish to annotate, it is often easiest to copy-paste names, types, and signatures from the API documentation. (The API documentation is a `.html` file that is sometimes colloquially called the "Javadoc".) This avoids common mistakes when writing stub files by hand, such as typos in the package name or in a method signature.

Here are steps to write a stub file by hand:

1. Create a file whose name ends in `.astub`, or use an existing one.
2. Write a package declaration, exactly as if you are writing a Java file for the target class. (If the file already contains a declaration for that package, re-use it by doing the rest of your work under it; don't write a duplicate package declaration in the file.) Typically, it is easiest to copy-paste the package name from API documentation or from an import statement in the code that you are type-checking, to avoid typos.
3. Write a class declaration header: `class` followed by the class name. The stub file format does not care whether the enclosing type is a class, interface, etc., so you can write "`class`" regardless of what kind of type it is.
   A common mistake at this stage is that the target method or field may be defined in a superclass of the static type of the class that is actually being used in the code you are trying to analyze. For example, suppose that you are trying to annotate the `drawImage(Image,int,int,int,int,ImageObserver)` method of an object whose static type is `java.awt.Graphics2D`. This method is actually defined in the abstract `Graphics` class (which is the superclass of `Graphics2D`), so it needs to appear within the class `Graphics` rather than the class `Graphics2D` in the stub file.
4. Write the element declaration that you wish to annotate, within the class you just created.
   For a method, this is the method signature, followed by "`;`".
   For a field, this is the type, field name, and "`;`", without an initializer.
5. Annotate the declaration that you just wrote.
6. Add import statements for any annotations you wrote.

36.5.6 Distributing stub files

If you are writing stub files but are not writing a checker, you can place the stub files anywhere that is convenient. However, please consider providing your stub files to the checker author, so that other users can also benefit from the stub files you wrote.

Stub files distributed with a checker appear in the same directory as the checker implementation (the `*Checker.java` file, see Section 37.2). For example, in the Checker Framework they appear in files such as

checker/src/main/java/org/checkerframework/checker/regex/apache-xerces.astub
checker/src/main/java/org/checkerframework/checker/signature/javaparser.astub

If you are distributing stub files with a checker implementation, you must configure your build system to copy the stub files into the distributed jar for your checker. Here are instructions if you are using the Checker Framework Gradle plugin:

- If the stub files appear in the same directory as the checker class, which is a subtype of `BaseTypeChecker`, use the following build configuration:

```
sourceSets {
  main {
      resources {
          srcDirs += ['src/main/java']
      }
  }
}
```

- If the stub files appear under the `src/main/resources/` directory in a subdirectory matching the package of your checker class, the default Gradle build configuration is sufficient.

In either case, recall that a `@StubFiles` annotation on the checker class lists stub files that are always used. For stub files whose use is optional (for example, because the behavior is unsound, or unsound except in certain circumstances), users must supply the `-Astubs=...` command-line option.

If a stub file contains annotations that are used by the framework rather than by any specific checker (such as purity annotations), and you wish to distribute it with the Checker Framework, put the stub file in directory `checker/src/main/resources/`. You can also do this if the stub file has annotations for multiple checkers. To use a stub file in directory `checker/src/main/resources/`, users must supply the `-Astubs=checker.jar/stub-file-name.astub` command-line option.

36.5.7 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. Some command-line options make the stub parser issue more warnings:

`-AstubWarnIfNotFound`
`-AstubNoWarnIfNotFound` The Checker Framework can issue a "Type not found" warning when a stub file contains a nonexistent element, for example when the stub file mentions a method that the class does not contain. By default, the Checker Framework warns about such problems in a stub file provided on the command line, but does not warn about built-in stub files. These command-line options turn the warnings on or off (respectively) for all stub files.
The `@NoAnnotationFileParserWarning` annotation on a package or type in a stub file causes no warning to be issued for that package or type, regardless of the command-line options.

`-AstubWarnIfNotFoundIgnoresClasses` Modifies the behavior of `-AstubWarnIfNotFound` to report only missing methods/fields, but ignore missing classes, even if other classes from the same package are present. Useful if a package spans more than one jar.

`-AstubWarnIfRedundantWithBytecode` Warn if a stub file entry is redundant with bytecode information. The warning means that the stub file's type is the same as the bytecode type, so that entry in the stub file has no effect. You could remove the entry in the stub file to make it shorter, or you could add an annotation to the stub file to make the entry have an effect.

`-AstubWarnNote` Issue a "note", not a warning, for the `-AstubWarn*` command-line arguments. A warning may prevent further compilation (depending on whether the `-Werror` command-line argument is passed), but a note permits compilation to proceed.

-AstubDebug Output debugging messages while parsing stub files, including about unknown classes, methods, and annotations. This overrides the `@NoAnnotationFileParserWarning` 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 an error of the form

```
AnnotationFileParser: Method isLLowerCase(char) not found in type java.lang.Character
```

by removing the extra "L" in the method name.

Fix an error of the form

```
AnnotationFileParser: Method enableForegroundNdefPush(Activity,NdefPushCallback)
      not found in type android.nfc.NfcAdapter
```

by removing the method `enableForegroundNdefPush(...)` from the stub file, because it is not defined in class `android.nfc.NfcAdapter` in the version of the library you are using.

36.6 Ajava files

There are two types of annotation files, which are files containing annotations that can be read by the Checker Framework. Section 36.5 describes stub files, which are used by programmers and type system designers. This section describes ajava files. Ajava files are primarily read and written by tools, such as whole-program type inference (see Section 35.2). (However, programmers are permitted to write `.ajava` files.) This section about ajava files is primarily of interest to the maintainers of such tools.

An ajava file is simply a valid Java source file. The Checker Framework reads its annotations. (This includes annotations that are written on anonymous and local classes — annotations in such locations are ignored in stub files.)

36.6.1 Using an Ajava file

The `-Aajava` command-line argument works like `-Astubs` (see Section 36.5.1). It takes a colon-separated list of files and directories containing ajava files, which end in the `.ajava` extension.

For example:

```
javac -processor org.checkerframework.checker.interning.InterningChecker \
  -Aajava=path/to/String.ajava:ajavadir MyFile.java MyOtherFile.java ...
```

If there's an annotation in both a source file and the corresponding ajava file, the Checker Framework uses the greatest lower bound, as with the `-AmergeStubsWithSource` option.

36.6.2 Corresponding source files and ajava files

If the Checker Framework reads both a source file and an ajava file that corresponds to it, then the source file and the ajava file must have a specific relationship to one another, to prevent errors that could occur due to mismatches.

Names of ajava files

The ajava file must either (1) be found below one of the arguments to `-Aajava`, in a subdirectory matching its package name, or (2) be unambiguously specified directly with its full path. A pair of `.ajava` files is specified ambiguously if one of their fully-qualified names is a substring of the other, and both are specified as `.ajava` files (rather than being found under directories). For example, if the `-Aajava` argument was `-Aajava= /foo/Bar.ajava: /baz/foo/Bar.ajava`, the two `.ajava` files would be considered ambiguous and an `ambiguous.ajava` warning would be issued. Files found

under a directory are always unambiguous, because their file structure must match their package name; the solution to an `ambiguous.ajava` warning is to convert the specific files in the `-Aajava` argument to directories.

The ajava file must be named `ClassName-checker.qualified.name.ajava` where `checker.qualified.name` is the fully qualified name of the checker that the ajava file's annotations belong to.

For example, an ajava file with tainting annotations for a class `outerpackage.innerpackage.MyClass` would be located in a subdirectory `outerpackage/innerpackage` and it would be named `MyClass-org.checkerframework.checker.tainting`

Contents of an ajava file

The ajava file must contain the same source code as the source file, with the following exceptions:

- The two may differ in annotations.
- The two may have different import statements.
- The ajava file may have explicit receivers where the source file doesn't. If a source file has a method declaration `void myMethod()`, the ajava file might contain `void myMethod(MyClass this)` or `void myMethod(@Interned MyClass this)`.
- The ajava file may have explicit type bounds that are implicit in the source file. If a source file has a type argument `?`, the ajava file might contain `? extends Object` or `? extends @Nullable Object`.

Inserting annotations from ajava files

The `InsertAjavaAnnotations.java` program inserts annotations from ajava files into Java files. This can be used to insert the annotations inferred by whole program inference into the source code of a program, similarly to the `insert-annotations-to-source` script from the Annotation File Utilities project.

Its arguments are a directory containing ajava files and a directory containing Java files. The `checker.jar` file must be in the classpath. Here is an example invocation:

```
java -cp $CHECKERFRAMEWORK/checker/dist/checker.jar \
  org.checkerframework.framework.ajava.InsertAjavaAnnotations \
  <path-to-ajava-files> <path-to-java-files>
```

36.7 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 a flag that may help you in determining which library files are read.

`-verbose` Outputs information about compile phases — when the compiler reads/parses/attributes/writes any file. Also outputs the classpath and sourcepath paths.

A syntax error in a stub file or the annotated JDK can lead to the file being silently ignored. A typo in an annotation name, or a missing `import` statement, can lead to annotations being silently ignored.

Chapter 37

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 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. People who wish to edit the Checker Framework source code or make pull requests should read the Checker Framework Developer Manual (`https://checkerframework.org/manual/developer-manual.html`).

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

```
import java.lang.annotation.Documented;
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;
import org.checkerframework.common.subtyping.qual.Unqualified;
import org.checkerframework.framework.qual.SubtypeOf;

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

This checker is so short because it builds on the Subtyping Checker (Chapter 30). See Section 30.2 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:

- Customize an existing checker. Checkers that are designed for extension include the Subtyping Checker (Chapter 30, page 183), the Accumulation Checker (Chapter 38, page 277), the Fake Enumeration Checker (Chapter 9, page 83), and the Units Checker (Chapter 20, page 140).
- 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. A problem with this strategy is that you can get tangled up if you don't fully understand the subtleties of the existing checker that you are modifying. Usually, it is easier to follow the instructions in this chapter. (If you are going to copy a checker despite the dangers, one good choice to copy and modify is the Regex Checker (Chapter 14, page 113). A bad choice is the Nullness Checker (Chapter 3, page 31), 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 31, page 188 or even include it in the Checker Framework distribution.

37.1 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. 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 colspan="2"></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
(<code>.java</code> ↔ <code>.class</code> files)</td></tr>
<tr><td colspan="8">Java 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.

37.2 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                 interface to the compiler
  | MyPropVisitor.java                 [optional] type rules
  | MyPropAnnotatedTypeFactory.java    [optional] type introduction and dataflow rules
```

`MyPropChecker.java` is occasionally optional, such as if you are building on the Subtyping Checker. If you want to create an artifact containing just the qualifiers (similar to the Checker Framework's `checker-qual` artifact), you should put the `qual/` directory in a separate Maven module or Gradle subproject.

Sections 37.5–37.9 describe the individual components of a type system as written using the Checker Framework:

37.5 **Type qualifiers and hierarchy.** You define the annotations for the type system and the subtyping relationships among qualified types (for instance, `@NonNull Object` is a subtype of `@Nullable Object`). This is also where you specify the default annotation that applies whenever the programmer wrote no annotation and no other defaulting rule applies.

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

37.7 **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 in every assignment, the type of the right-hand side is a subtype of the type of the left-hand side. Your checker automatically inherits these subtyping rules from the Base Checker (Chapter 30), 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.

37.8 **Type introduction rules.** You specify the type of some expressions where the rules differ from the built-in framework rules.

37.9 **Dataflow rules.** These optional rules enhance flow-sensitive type qualifier inference (also sometimes called "local variable type inference").

37.3 Compiling and using a custom checker

You can place your checker's source files wherever you like. One choice is to write your checker in a fork of the Checker Framework repository `https://github.com/typetools/checker-framework`. Another choice is to write it in a stand-alone repository. Here is a template for a stand-alone repository: `https://github.com/typetools/templatefora-checker`; at that URL, click the "Use this template" button.

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 same path (the classpath or processorpath) as the Checker Framework. Invoking a custom checker that builds on the Subtyping Checker is slightly different (Section 30.1).

37.4 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. Write the user manual. Do this before you start coding. 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 37.13 gives a suggested structure for the manual chapter, which will help you avoid omitting any parts. If your checker is in a clone of the Checker Framework, don't forget to add a LaTeX `\include` directive to `manual.tex`. 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 should "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. It is more efficient to find problems now rather than after coding up your design.
2. Implement the type qualifiers and hierarchy (Section 37.5). Compile your changes.
   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:

```
import org.checkerframework.checker.mycheckername.qual.UnknownSign;
import org.checkerframework.checker.mycheckername.qual.NonNegative;

class TestHierarchy {
  void testHierarchy(@UnknownSign int us, @NonNegative int nn) {
    @UnknownSign int a = us;
    @UnknownSign int b = nn;
```

```
    // :: error: [assignment]
    @NonNegative int c = us; // expected error on this line
    @NonNegative int d = nn;
  }
}
```

If you are working in a clone of the Checker Framework, put the above test case in file `checker/tests/mycheckername/TestHi`
Type-check your test files using the Subtyping Checker (Chapter 30, page 183), using a command like

```
javacheck \
  -processor org.checkerframework.common.subtyping.SubtypingChecker \
  -Aquals=org.checkerframework.checker.mycheckername.qual.MyTopQual,org.checkerframework.checker.mycheckername.qu
  TestHierarchy.java
```

3. Write the checker class itself (Section 37.6).
   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 ...
   ```

   If the default qualifier is not the top qualifier, you may see `warning: (inconsistent.constructor.type)`.
   If the checker should never issue such an error (because a newly-constructed object can be correctly viewed as any type at all), then copy `TaintingVisitor.java` into your checker, changing "Tainting".
4. Create test infrastructure. If your checker source code is in a clone of the Checker Framework repository, integrate your checker with the Checker Framework's Gradle targets for testing (Section 37.11). This will make it much more convenient to run tests, and to ensure that they are passing, as your work proceeds.
   If there are errors in the "all-systems" tests, don't worry about them until after you have annotated the JDK. However, specifying `@RelevantJavaTypes` (Section 37.5.5) may eliminate some of those errors, especially if the default qualifier is not the top qualifier.
5. Annotate parts of the JDK, if relevant (Section 37.10). You will clone the Checker Framework JDK, naming it `jdk` in a sibling directory of `checker-framework` (that is, they have the same parent). Most changes will consist of adding an `import` statement, plus uses of qualifiers, in Java files under `src/java.base/share/classes/`. Also follow the instructions in section "Qualifier definitions" of file `README.md`.
   Write test cases for a few of the annotated JDK methods to ensure that the annotations are being properly read by your checker.
   If there are errors in the Checker Framework's "all-systems" tests, you can suppress them if they are expected. See `framework/tests/all-systems/README` for more information.
6. Implement type rules, if any (Section 37.7). 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), read the Javadoc to familiarize yourself with the utility routines in the `org.checkerframework.javacutil` package, especially `AnnotationBuilder`, `AnnotationUtils`, `ElementUtils`, `TreeUtils`, `TypeAnnotationUtils`, and `TypesUtils`. You will learn 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 testArrayIndexing(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 37.8). Some type systems need JDK annotations but don't have any type introduction rules.
   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 testTypeIntroduction(String[] myArray) {
  @NonNegative int nn1 = -1;  // expected error on this line
  @NonNegative int nn2 = 0;
  @NonNegative int nn3 = 1;
  @NonNegative int nn4 = myArray.length;
}
```

8. Optionally, implement dataflow refinement rules (Section 37.9). Dataflow refinement rules are only possible for a type system that supports a run-time test (Section 33.7.4).
   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 testDataflow(@UnknownSign int us, @NonNegative int nn) {
  @NonNegative int nn2;
  nn2 = us;  // expected error on this line
  if (us > 0) {
    nn2 = us;
  }
  if (us >= 0) {
    nn2 = us;
  }
  if (0 < us) {
    nn2 = us;
  }
  if (0 <= us) {
    nn2 = us;
  }
  nn = us;  // expected error on this line
}
```

37.5 Annotations: Type qualifiers and hierarchy

A type system designer specifies the qualifiers in the type system (Section 37.5.1) 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 37.5.2), or procedurally through subclassing `QualifierHierarchy` or `TypeHierarchy` (Section 37.5.3).

37.5.1 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.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})
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 the additional value `ElementType.TYPE_PARAMETER`, but no other `ElementType` values.

The qualifiers should be in a `qual` subpackage of the type-checker package — by default, the checker reflectively determines the qualifiers, and the only thing that matters for that is the package name. As mentioned in Section 37.2, the files for that `qual` package can be either in a subdirectory or in a separate Gradle module, depending on whether they should be distributable separately. The Checker Framework automatically treats any annotation that is declared in the `qual` package as a type qualifier. (See Section 37.6.1 for more details.) 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`.

Your type system should include a top qualifier and a bottom qualifier (Section 37.5.7). The top qualifier is conventionally named *`CheckerName`*`Unknown`. Most type systems should also include a polymorphic qualifier `@Poly`*`MyTypeSystem`* (Section 37.5.2).

One of the annotations must be meta-annotated with `@DefaultQualifierInHierarchy` or `@DefaultFor(TypeUseLocation.OTHER`
It is usually easiest to make the top type be the default; this may or may not be the correct default for the type system design.

Choose good names for the qualifiers, because users will write these in their source code. The Javadoc of every type qualifier should include a precise English description and an example use of the qualifier.

In source code, declaration annotations are written on their own line, and type qualifiers are written on the same line as the type they apply to. Section 40.6.10 explains how to configure tools to format type annotations properly.

37.5.2 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 32.2), 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 30.2.
- `@PolymorphicQualifier` denotes that a qualifier is a polymorphic qualifier. For example:

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

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

  A polymorphic qualifier must not have a `@SubtypeOf` meta-annotation nor be mentioned in any other `@SubtypeOf` meta-annotation.
  For how to restrict instantiation, see Section 37.5.2. For a general description of polymorphic qualifiers, see Section 32.2.

The declarative and procedural mechanisms for specifying the hierarchy can be used together. If any of the annotations representing type qualifiers have elements, then the relationships between those qualifiers must be defined procedurally.

Restricting instantiation

Sometimes, you wish to restrict how a polymorphic qualifier can be instantiated. An example is the Signedness Checker (Chapter 21, page 144). Its two main qualifiers are `@Signed` and `@Unsigned`, and the top qualifier (which should be rarely used) is `@UnknownSignedness`.

As described so far, there is no way for a programmer to require that two method arguments have the same signedness type. Writing

```
void foo(@PolySigned Object a, @PolySigned Object b) { ... }
```

permits every invocation, because one of its instantiations is

```
void foo(@UnknownSignedness Object a, @UnknownSignedness Object b) { ... }
```

which is consistent with any two arguments, regardless of their signedness type.

It is possible to permit only method calls whose arguments have exactly the same type qualifier. This is enabled for an entire type system. To do so, create a subclass of `DefaultQualifierPolymorphism`, override its `combine` method, and make method `createQualifierPolymorphism` in the type factory return an instance of the new subclass. The Signedness Checker gives an example.

Changing the qualifier polymorphism allows a loophole: upcasts to top. For example, the type system permits a call like

```
Objects.equals((@UnknownSignedness) mySigned, (@UnknownSignedness) myUnsigned)
```

A type system can forbid upcasts to top (including in assignments, not just explicit casts) to close this loophole.

If you change the qualifier polymorphism, your type system may need class qualifier polymorphism (see Section 32.3). And, you may need to suppress false positive warnings within the body of methods whose formal parameter types are annotated with the polymorphic qualifier (e.g., `@PolySigned`).

37.5.3 Procedurally defining the qualifier hierarchy

The declarative syntax suffices for most cases. More complex type hierarchies can be expressed by overriding, in your subclass of `BaseAnnotatedTypeFactory`, 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 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".)

37.5.4 Defining the default annotation

A type system applies the default qualifier where the user has not written a qualifier (and no other default qualifier is applicable), as explained in Section 33.5.

The type system designer must specify the default annotation. The designer can specify the 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, after the declaration of the element in the qualifier file, append the string `default` *`value`*, where *value* is the value you want to be the default. For instance, `int value() default 0;` would make `value` default to zero. Alternatively, the procedural method described above can be used.

The default qualifier applies to most, but not all, unannotated types. Section 33.5.3 describes other defaulting rules that are automatically added to every checker. Also, Section 33.5 describes other meta-annotations used to specify default annotations.

37.5.5 Relevant Java types

Sometimes, a checker is only relevant to certain Java types. The `@RelevantJavaTypes` annotation on the checker class indicates that its qualifiers may only be written on those types, their subtypes, and their supertypes.

As an example of relevant types, the Format String Checker is relevant only to `CharSequence` and its subtypes such as `String`. You cannot write `@Format` on a `List` or a `Number`. As another example, the Index Checker and the Signedness Checker are only relevant to integral numeric types. You cannot write `@Positive` on a `List` or a `float`.

If a user writes an annotation on an irrelevant type, the Checker Framework issues an `anno.on.irrelevant` error.

Irrelevant types are ordinarily defaulted to the top annotation. A checker designer may change the default, and it may differ from type to type. For example, the Signedness Checker treats all types that are not numeric integral types as `@Signed`, but treats the `char` and `Character` types as `@Unsigned`.

Relevant types and supertypes

If a type is relevant, so are all its supertypes including `Object`. This prevents qualifiers from being lost via an assignment to a supertype. For example, in this code:

```
@SomeAnn MyClass x = ...;
Object y = x;
MyClass z = (MyClass) y;
```

variables `x` and `z` have the same type qualifiers, without them being lost via the assignment to `Object`.

37.5.6 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 `@Bottom`. In a client program, a use of type `T` may require type qualifier `@Bottom` 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.

37.5.7 Completeness of the qualifier hierarchy

When you define a type system, its qualifier hierarchy must be a lattice: every set of qualifiers has a unique least upper bound and a unique greatest lower bound. This implies that there must be a top qualifier that is a supertype of all other qualifiers, and there must be a bottom qualifier that is a subtype of all other qualifiers. Furthermore, the top qualifier and bottom qualifier should be defined specifically for the type system. Don't reuse an existing qualifier from the Checker Framework such as `@Unqualified`.

It is possible that a single type-checker checks multiple qualifier hierarchies. An example is the Nullness Checker, which has three separate qualifier hierarchies, one each for nullness, initialization, and map keys. In this case, each qualifier 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 qualifier hierarchy must have a bottom qualifier — a qualifier that is a (direct or indirect) subtype of every other qualifier.

`null` has the bottom type. Because the only value with type `Void` is `null`, uses of the type `Void` are also bottom. (The only exception is if the type system has special treatment for `null` values, as the Nullness Checker does. In that case, add the meta-annotation `@QualifierForLiterals(LiteralKind.NULL)` to the correct qualifier.) 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. Also, 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`".

It might seem that the hierarchy shown in Figure 3.5 lacks a bottom qualifier. The actual implementation does contain a bottom qualifier, `@FBCBottom`, that users rarely write.

Top qualifier Your qualifier hierarchy must have a single top qualifier — a qualifier that is a (direct or indirect) supertype of every other qualifier. There are many reasons this is essential. One reason is that the default type for local variables is the top qualifier, and there must be an unambiguous choice for the default type.

37.5.8 Annotations whose argument is a Java expression (dependent type annotations)

Sometimes, an annotation needs to refer to a Java expression. Section 33.8 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 element, `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 33.8. 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 of those methods, constructors, and fields. 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 a 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 `JavaExpression` object by calling `fromTree(ExpressionTree)` or `fromNode(Node)`. Then, call `toString()` on the `JavaExpression` object.

37.5.9 Repeatable annotations

Some pre- and post-condition annotations that have multiple elements (that is, annotations that take multiple arguments) should be repeatable, so that programmers can specify them more than once. An example is `@EnsuresNonNullIf`; it could not be defined with each of its elements being a list, as (for example) `@KeyFor` is.

Make an annotation *A* repeatable by defining a nested annotation (within *A*'s definition) named `List`, and writing `@Repeatable(A.List.class)` on *A*'s definition.

37.6 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 Java compiler and the checker. It invokes the type-rule check visitor on every Java source file being compiled. The checker uses `SourceChecker.reportError` and `SourceChecker.reportWarning` to issue errors and warnings.

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 `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 `reportError` and `reportWarning` (like `"type.incompatible"`) and the values are the strings to be printed (`"cannot assign ..."`). The `messages.properties` file only needs to mention the new messages that the checker defines. It is also allowed to override messages defined in superclasses, but this is rarely needed. Section 34.1.3 (page 224) discusses best practices when using a message key in a `@SuppressWarnings` annotation.

37.6.1 Indicating supported annotations

A checker must indicate the annotations that it supports (that make up its type hierarchy).

By default, a checker supports all type annotations in a subdirectory named `qual` that is in the same directory as the checker (unless you are creating multiple distribution artifacts, one for your checker and one for its qualifiers). A type annotation is meta-annotated with either `@Target(ElementType.TYPE_USE)` or `@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})`.

To indicate support for annotations that are located outside of the `qual` subdirectory, or annotations that have other `ElementType` values, checker writers can override the `createSupportedTypeQualifiers` method (see its Javadoc for details). It is required to define `createSupportedTypeQualifiers` if you are mixing qualifiers from multiple

directories (including when extending an existing checker that has its own qualifiers), and if you are using the Buck build tool, whose class loader cannot find the qualifier directory.

An aggregate checker (which extends `AggregateChecker`) does not need to specify its type qualifiers, but each of its component checkers should do so.

37.6.2 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, except the error messages are interleaved such that all errors about a line of code appear together.
   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 `getSupportedCheckers` method, like the following:

   ```
   public class MyUnitCheckers extends AggregateChecker {
     @Override
     protected Collection<Class<? extends SourceChecker>> getSupportedCheckers() {
       Collection<Class<? extends SourceChecker>> checkers = new ArrayList<>(3);
       Collections.addAll(
         checkers,
         DistanceUnitChecker.class, VelocityUnitChecker.class, MassUnitChecker.class);
       return checkers;
     }
   }
   ```

   An example of an aggregate checker is `I18nChecker` (see Section 17.2, page 130), 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 `getTypeFactoryOfSubcheckerOrNull`.
   An example of a compound checker is `SignednessChecker` (see Chapter 21, page 144) for which `ValueChecker` is a subchecker.

37.6.3 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 mypackage.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 mypackage.MyChecker -Amyoption=p1,p2
```

The value is returned as a single string and you have to perform the required parsing of the option.

Default options

The default options passed to a custom checker may be customized by overriding the `getOptions` method in the custom checker class.

For example, to ignore the `introduce.eliminate` warnings raised by the Optional Checker (Chapter 5), override `getOptions` as follows:

```
@Override
public Map<String, String> getOptions() {
  Map<String, String> options = new HashMap<>(super.getOptions());
  options.put("suppressWarnings", "optional:introduce.eliminate");
  return options;
}
```

in the `OptionalChecker` class.

37.7 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 jdk.compiler API, and it issues an error or warning (by calling `reportError` or `reportWarning`) 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) when a type is bound to an incompatible generic type variable
- invalid method invocation (method.invocation) when a method is invoked on an object whose type is incompatible with the method receiver type
- invalid overriding parameter type (override.param) when a parameter in a method declaration is incompatible with that parameter in the overridden method's declaration
- invalid overriding return type (override.return) when the return type in a method declaration is incompatible with the return type in the overridden method's declaration
- invalid overriding receiver type (override.receiver) when a receiver in a method declaration is incompatible with that receiver in the overridden method's declaration
- invalid overriding type parameter bound (override.type.parameter.upper, override.type.parameter.lower) when a type parameter bound in a method declaration does not contain that bound in the overridden method's declaration

37.7.1 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 39.8).

37.7.2 Avoid hardcoding

If a method's contract is expressible in the type system's annotation syntax, then you should write annotations, in a stub file or annotated JDK (Chapter 36).

If the contract is not expressible, you can create a declaration annotation, and your checker can treat methods with that annotation specially. An example is the `@FormatMethod` annotation; see Section 15.5.

A final alternative is to write a type-checking rule for method invocation, where your rule checks the (class and) name of the method being called and then treats the method in a special way.

[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.

37.8 Type factory: Type introduction rules

The annotated type of expressions and types is defined via type introduction rules in the type factory. For most expressions and types, these rules are the same regardless of the type system. For example, the type of a method invocation expression is the return type of the invoked method, viewpoint-adapted for the call site. The framework implements these rules so that all type systems automatically use them. For other expressions, such as string literals, their (annotated) types depend on the type system, so the framework provides a way to specify what qualifiers should apply to these expressions.

Defaulting rules are type introduction rules for computing the annotated type for an unannotated type; these rules are explained in Section 37.5.4. The meta-annotation `@QualifierForLiterals` can be written on an annotation declaration to specify that the annotation should be applied to the type of literals listed in the meta-annotation.

37.8.1 Procedurally specifying type introduction rules

If the meta-annotations are not sufficiently expressive, then you can write your own type introduction rules. There are three ways to do so. Each makes changes to an `AnnotatedTypeMirror`, which is the Checker Framework's representation of an annotated type.

1. Define a subclass of `TreeAnnotator`, typically as a private inner class of your `AnnotatedTypeFactory`. There is a method of `TreeAnnotator` for every AST node, and the visitor has access to both the tree (the AST node) and its type.
   A `TreeAnnotator` is best suited to changing the type of an expression. Don't try to use a `TreeAnnotator` to change the type of a variable or method declaration, because that change will not generally be seen at uses of the variable or method.
   After defining your `TreeAnnotator`, override `createTreeAnnotator` (in your subclass of `AnnotatedTypeFactory`) to return a `ListTreeAnnotator` containing your new `TreeAnnotator`, as in:

   ```
   @Override
   protected TreeAnnotator createTreeAnnotator() {
       return new ListTreeAnnotator(super.createTreeAnnotator(), new MyTreeAnnotator(this));
   }
   ```

   In code like the above, you might choose to put your `TreeAnnotator` first. Several tree annotators are run by default, and among them, `PropagationTreeAnnotator` adds annotations to `AnnotatedTypeMirror`s that do not have an annotation, but has no effect on those that have an annotation.
2. Define a subclass of a `TypeAnnotator`, typically as a private inner class of your `AnnotatedTypeFactory`. There is a method of `TypeAnnotator` for every kind of type, and the visitor has access to only the type (including base type and qualifiers). In your subclass of `AnnotatedTypeFactory`, override `createTypeAnnotator` to return a `ListTypeAnnotator` containing that annotator, as in

   ```
   @Override
   protected TypeAnnotator createTypeAnnotator() {
       return new ListTypeAnnotator(new MyTypeAnnotator(this), super.createTypeAnnotator());
   }
   ```

   (or put your `TypeAnnotator` last, depending on the behavior you want).
3. Create a subclass of `AnnotatedTypeFactory` and override two `addComputedTypeAnnotations` methods: `addComputedTypeAnnotations(Tree,AnnotatedTypeMirror)` (or `addComputedTypeAnnotations(Tree,AnnotatedTypeMi` if extending `GenericAnnotatedTypeFactory`) and `addComputedTypeAnnotations(Element,AnnotatedTypeMirror)`.
   The methods can make arbitrary changes to the annotations on a type.
   Recall that `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 annotations and type refinement (see Section 33.4 for explanations of these concepts).
   The approach of overriding `addComputedTypeAnnotations` is a last resort, if your logic cannot be implemented using a `TreeAnnotator` or a `TypeAnnotator`. The implementation of `addComputedTypeAnnotations` in

GenericAnnotatedTypeFactory calls the tree annotator and the type annotator (in that order), but by overriding the method you can cause your logic to be run even earlier or even later.

37.8.2 Recursion in TreeAnnotators

A subclass of TreeAnnotator should not, in general, call getAnnotatedType() more than once. This is primarily a concern for visitBinary() and visitCompoundAssignment(). For example, the implementation of visitBinary() should not start with

```
AnnotatedTypeMirror lExpr = getAnnotatedType(tree.getLeftOperand());
AnnotatedTypeMirror rExpr = getAnnotatedType(tree.getRightOperand());
```

The reason is that PropagationTreeAnnotator already does such recursion. If two TreeAnnotators do so, then expressions at the base of a large binary tree will be visited exponentially many times, leading to slowdowns or the appearance of an infinite loop.

One approach is to not call getAnnotatedType() in visitBinary(); this is what UpperBoundAnnotatedTypeFactory does.

Another approach is to disable PropagationTreeAnnotator, doing all work in a different TreeAnnotator. Create a subclass of PropagationTreeAnnotator whose visitBinary() method does no work, and use that subclass in place of PropagationTreeAnnotator itself. This is what RegexAnnotatedTypeFactory and UnitsAnnotatedTypeFactory do.

37.9 Dataflow: enhancing flow-sensitive type refinement

By default, every checker performs flow-sensitive type refinement, as described in Section 33.7.

This section of the manual explains how to enhance the Checker Framework's built-in 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 33.7.4 (page 215) gives examples of type systems with and without run-time tests.

The steps to customizing type refinement are:

1. §37.9.1 Determine which expressions will be refined.
2. §37.9.2 Create required class and configure its use.
3. §37.9.3 Override methods that handle Nodes of interest.
4. §37.9.4 Implement the refinement.

The Regex Checker's dataflow customization for the RegexUtil.asRegex run-time check is used as a running example.

If needed, you can find more details about the implementation of type refinement, and the control flow graph (CFG) data structure that it uses, in the Dataflow Manual.

37.9.1 Determine which expressions to refine

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 37.9.2–37.9.4 explain how to create a transfer class when the `@EnsuresQualifier` and `@EnsuresQualifierIf` annotations are insufficient.

37.9.2 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, except for a constructor:

```
public MyCheckerTransfer(CFAnalysis analysis) {
  super(analysis);
}
```

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 a new instance of the class.)

(As a reminder, use of `@EnsuresQualifier` and `@EnsuresQualifierIf` may obviate the need for a transfer class.)

37.9.3 Override methods that handle Nodes of interest

Decide what source code syntax is relevant to the run-time checks or run-time operations you are trying to support. The CFG (control flow graph) represents source code as `Node`s. Some `Node`s correspond to a node in the abstract syntax tree of the program being checked, while others are synthesized during CFG construction and have no corresponding AST node (see Section 37.9.3).

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 based on a call to the `RegexUtil.isRegex` method. 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`, and `VariableDeclarationNode`.

A `Node` is basically equivalent to a javac compiler `Tree`.

See Section 37.14 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: ""
```

37.9.4 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 37.9.1:

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 `Tree`s.)
   For example, when `RegexTransfer` evaluates a `RegexUtil.asRegex` invocation, it updates the `TransferResult`'s result type. This changes the type of the `RegexUtil.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 branch. 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 `RegexUtil.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 static method asRegex(s, groups) in a class named RegexUtil?
if (receiverName.equals("RegexUtil")
    && ElementUtils.matchesElement(method,
           "asRegex", 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 `RegexUtil.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 24, page 161).

```
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 `RegexUtil.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.

37.9.5 Further customizing flow-sensitive inference

By default, the Checker Framework assumes that modifications to an object's fields do not change its type qualifiers. This is true for all provenance-based type systems (see Section 33.7.4), for all value-based type systems over immutable values, and for some other type systems. To indicate that **side effects can change a value's type qualifiers**, write `sideEffectsUnrefineAliases = true;` in the type factory constructor, before the call to `this.postInit();`. Note that this setting may lead to large numbers of false positive warnings. You can reduce the number of false positive warnings by writing more precise side effect annotations.

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 37.9), put the following two lines at the beginning of the constructor for your subtype of `BaseAnnotatedTypeFactory`:

```
// disable flow inference
super(checker, /*useFlow=*/ false);
```

37.10 Annotated JDK and other annotated libraries

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. You have two options:

- Write JDK annotations in a fork of `https://github.com/typetools/jdk`.
  If your checker is written in a fork of `https://github.com/typetools/jdk`, then the checker and the JDK must use the same fork name (GitHub organization) and branch name; this is necessary so that the CI jobs use the right annotated JDK.
  Clone the JDK and the Checker Framework in the same place; that is, your working copy of the JDK (a `jdk` directory) should be a sibling of your working copy of the Checker Framework (a `checker-framework` directory).
  Here are some tips:
  - Add an `@AnnotatedFor` annotation to each file you annotate.
  - Whenever you add a file, fully annotate it, as described in Section 36.1.4.
  - 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 not type-checked.
  - Double-check your work. If you forget to import an annotation, or you misspell it in the `import` statement or at some use, it will be *silently ignored*, which is frustrating.
- Write JDK annotations as stub files (partial Java source files).
  Create a file `jdk.astub` in the checker's main source directory. You can also create `jdk`*N*`.astub` files that contain methods or classes that only exist in certain JDK versions. The JDK stub files will be automatically used by the checker, unless the user supplies the command-line option `-Aignorejdkastub`.
  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.
  When a stub file should be used by multiple checkers (for example, if it contains purity annotations that are needed by multiple distinct checkers), the stub file should appear in directory `checker/src/main/resources/`. In the distribution, it will appear at the top level of the `checker.jar` file. It will not be used automatically; a user must pass the `-Astubs=checker.jar/`*`stubfilename.astub`* command-line argument (see Section 2.2.1).
  While creating a stub file, you may find the debugging options described in Section 36.5.7 useful.

37.11 Testing framework

The Checker Framework provides a convenient way to write tests for your checker. Each test case is a Java file, with inline indications of what errors and warnings (if any) a checker should emit. An example is

```
class MyNullnessTest {
  void method() {
    Object nullable = null;
    // :: error: [dereference.of.nullable]
    nullable.toString();
  }
}
```

When the Nullness Checker is run on the above code, it should produce exactly one error, whose message key is `dereference.of.nullable`, on the line following the "// ::" comment.

The testing infrastructure is extensively documented in file `checker-framework/checker/tests/README.md`.

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.

37.12 Debugging options

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

37.12.1 Amount of detail in messages

- `-AprintAllQualifiers`: print all type qualifiers, including qualifiers meta-annotated with `@InvisibleQualifier`, which are usually not shown.
- `-AprintVerboseGenerics`: print more information about type parameters and wildcards when they appear in warning messages. Supplying this also implies `-AprintAllQualifiers`.
- `-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.
- `-Aonelinemsg`: print each error message on a single line, with logical lines separated by "/". This is useful when using a tool that only shows the first line of the error.
- `-AnoPrintErrorStack`: don't print a stack trace when an internal Checker Framework error occurs. Setting this option is rare. You should only do it if you have discovered a bug in a checker, you have already reported the bug, and you want to continue using the checker on a large codebase without being inundated in stack traces.
- `-AnoWarnMemoryConstraints`: when memory constraints are impeding performance, issue a note rather than a warning. This prevents the message from being elevated to an error by `-Werror`.
- `-AdumpOnErrors`: output a stack trace when reporting errors or warnings.
- `-AexceptionLineSeparator`: control what line separator is used when printing exceptions. This is useful when tools, such as Maven, only display the first line of exception messages. For example, `-AexceptionLineSeparator="
"`.

37.12.2 Format of output

- `-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.

The javac-diagnostics-wrapper (`https://github.com/eisopux/javac-diagnostics-wrapper`) tool can transform javac's textual output into other formats, such as JSON in LSP (Language Server Protocol) format.

37.12.3 Stub and JDK libraries

- `-Aignorejdkastub`: ignore the `jdk.astub` and `jdk`*N*`.astub` files in the checker directory. Files passed through the `-Astubs` option are still processed. This is useful when experimenting with an alternative stub file.
- `-ApermitMissingJdk`: don't issue an error if no annotated JDK can be found.
- `-AparseAllJdk`: parse all JDK files at startup rather than as needed.
- `-AstubDebug`: print debugging messages while processing stub files. Section 36.5.7 describes more diagnostic command-line arguments.

37.12.4 Progress tracing

- `-Afilenames`: print the name of each file before type-checking it. This can be useful for determining that a long compilation job is making progress.
  This option can also help to keep a CI job alive (for example, Travis CI terminates any job that does not produce output for 10 minutes). This does not work if you are using Maven, because Maven does not print the full output of the Java compiler (see `https://github.com/codehaus-plexus/plexus-compiler/issues/149`). (Also, maven-compiler-plugin is buggy: it sometimes doesn't print any output if it cannot parse it.)

- `-Ashowchecks`: print debugging information for each pseudo-assignment check (`commonAssignmentCheck`) performed by `BaseTypeVisitor`; see Section 37.7.
- `-AshowWpiFailedInferences`: print debugging information about failed inference steps during whole-program inference. Must be used with `-Ainfer`; see Section 35.2.

If the Checker Framework is in an infinite loop, you can use `-Afilenames` and `-Ashowchecks` (which is *very* verbose) to determine what source code triggers the infinite loop.

While the Checker Framework is in its infinite loop (no more progress is being made), get a traceback (stack trace) from each Java thread, using any of the following four techniques:

- From a different shell than the Checker Framework is running in (running just `jcmd` gives a list of process IDs):
  - `jstack` *`pid`*
  - `jcmd` *`pid`* `Thread.print`
  - `kill -QUIT` *`pid`*
- From the same shell that the Checker Framework is running in:
  - Press `ctrl-\`

37.12.5 Saving the command-line arguments to a file

- `-AoutputArgsToFile`: This saves to a file the final command-line parameters as passed to the compiler. The file can be used as a script to re-execute the same compilation command. (The script file must be marked as executable on Unix, or must have a `.bat` extension on Windows.) Example usage: `-AoutputArgsToFile=$HOME/scriptfile` The `-AoutputArgsToFile` command-line argument is processed by `CheckerMain`, not by the annotation processor. That means that it can be supplied only when you use the Checker Framework compiler (the "Checker Framework javac wrapper"), and it cannot be written in a file containing command-line arguments passed to the compiler using the `@argfile` syntax.

37.12.6 Visualizing the dataflow graph

To understand control flow in your program and the resulting type refinement, you can create a graphical representation of the CFG.

Typical use is:

```
javacheck -processor mypackage.MyProcessor -Aflowdotdir=. MyClass.java
for dotfile in *.dot; do dot -Tpdf -o "$dotfile.pdf" "$dotfile"; done
```

or (for more verbose output)

```
javacheck -processor mypackage.MyProcessor \
  -Acfgviz=org.checkerframework.dataflow.cfg.visualize.DOTCFGVisualizer,outdir=.,verbose \
  MyClass.java
for dotfile in *.dot; do dot -Tpdf -o "$dotfile.pdf" "$dotfile"; done
```

where the first command creates file `MyClass.dot` that represents the CFG, and the second command writes the CFG to a PDF file. The `dot` program is part of Graphviz.

In the output, conditional basic blocks are represented as octagons with two successors. Special basic blocks are represented as ovals (e.g., the entry and exit point of the method).

Creating a CFG while running a type-checker

To create a CFG while running a type-checker, use the following command-line options.
(See above for typical usage.)

- `-Aflowdotdir=`*`somedir`*: Specify directory for `.dot` files visualizing the CFG. Shorthand for `-Acfgviz=org.checkerframework.dataflow.cfg.visualize.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 37.9). The graph also contains information about flow-sensitively refined types of various expressions at many program points.
  The argument is a comma-separated sequence. The first element is the fully-qualified name of the `org.checkerframework.dataflo` implementation that should be used. Currently, the possibilities are:
  - `org.checkerframework.dataflow.cfg.visualize.DOTCFGVisualizer`
  - `org.checkerframework.dataflow.cfg.visualize.StringCFGVisualizer`

  The remaining keys or key-value pairs are passed to `CFGVisualizer.init`. Supported keys include
  - `verbose` outputs the store after each basic block (in addition to before); shows transfer result values, unique IDs, and the processing order
  - `outdir` directory into which to write files
  - `checkerName`

Creating a CFG by running CFGVisualizeLauncher

You can also use `CFGVisualizeLauncher` to generate a DOT or String representation of the control flow graph of a given method in a given class. The CFG is generated and output, but no dataflow analysis is performed.

```
java -cp $CHECKERFRAMEWORK/checker/dist/checker.jar \
  org.checkerframework.dataflow.cfg.visualize.CFGVisualizeLauncher \
  MyClass.java --class MyClass --method test --pdf
```

The above command will generate the corresponding `.dot` and `.pdf` files for the method `test` in the class `MyClass` in the project directory. To generate a string representation of the graph to standard output, remove `-pdf` but add `-string`. For example:

```
java -cp $CHECKERFRAMEWORK/checker/dist/checker.jar \
  org.checkerframework.dataflow.cfg.visualize.CFGVisualizeLauncher \
  MyClass.java --class MyClass --method test --string
```

For more details about invoking `CFGVisualizeLauncher`, run it with no arguments.

37.12.7 Miscellaneous debugging options

- `-AresourceStats`: Whether to output resource statistics at JVM shutdown.
- `-AatfDoNotCache`: If provided, the Checker Framework will not cache results but will recompute them. This makes the Checker Framework run slower. If the Checker Framework behaves differently with and without this flag, then there is a bug in its caching code. Please report that bug.
- `-AatfCacheSize`: The size of the Checker Framework's internal caches. Ignored if `-AatfDoNotCache` is provided. Most users have no need to set this.

37.12.8 Examples

The following example demonstrates how these options are used:

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

37.12.9 Using an external debugger

You can use any standard debugger to observe the execution of your checker.

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
```

37.13 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 each 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, and the polymorphic qualifier's Javadoc should also refer to the `qualifier-polymorphism` section. For example, in `PolyPresent.java`:

```
 * @checker_framework.manual #optional-checker Optional Checker
 * @checker_framework.manual #qualifier-polymorphism Qualifier polymorphism
```

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 33.7.4. Add the new file to `docs/manual/Makefile`. Keep the lists in the same order as the manual chapters, to help us notice if anything is missing.

For a chapter or (sub)*section, use `\sectionAndLabel{Section title}{section-label}`. Section labels should start with the checker name (as in `bellyrub-examples`) and not with "`sec:`". Figure labels should start with "fig-*checkername*" and not with "fig:". The `\sectionAndLabel` convention is required by the `lwarp-postprocess` script, which makes the HTML version of the manual use stable, label-based anchors in cross-references. 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.

37.14 javac implementation survival guide

Since this section of the manual was written, the useful "Hitchhiker's Guide to javac" has become available at `https://openjdk.org/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.

37.14.1 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 presents 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 classes/interfaces, `ExecutableElement` for methods/constructors, and `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, such as 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 `EnhancedForLoopTree` for an enhanced `for` statement (also known as a foreach loop).

You should limit your use of trees. A checker uses `Tree`s 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.

To obtain the position (line number and column number) of a `Tree`, use a `com.sun.source.util.Trees` instance to obtain a `com.sun.source.util.SourcePositions` via `getSourcePositions()`, then call `getStartPosition(...)` (or `getEndPosition(...)`) on it. `com.sun.source.tree.CompilationUnitTree` has a `LineMap` that lets you convert positions to line numbers. Alternatively, you can cast the `Tree` to the javac-internal type `com.sun.tools.javac.tree.JCTree` and call its `pos()` method, which returns a `JCDiagnostic.DiagnosticPosition`; doing so requires your code to be compiled and run with `--add-exports jdk.compiler/com.sun.tools.javac.tree=ALL-UNNAMED` and `--add-exports jdk.compiler/com.sun.tools.javac.util=ALL-UNNAMED`.

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 Subinterface` to determine which kind of `TypeMirror` or `Element` an expression is, because some of the classes that implement the `TypeMirror` and `Element` subinterfaces implement multiple subinterfaces. For example, `type instanceof DeclaredType` and `type instanceof UnionType` both return true if `type` is a `com.sun.tools.javac.code.Type.UnionClassType` object.

Instead, use the `TypeMirror.getKind()` or `Element.getKind()` method. For example, if `type` is a `com.sun.tools.javac.code.T` object, then `type.getKind() == TypeKind.DECLARED` is false and `type.getKind() == TypeKind.UNION` is true.

For `Tree`s, you can use either `Tree.getKind()` or `instanceof`.

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 OpenJDK 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 to distinguish the class names (e.g., `TypesUtils`, `TreeUtils`, `ElementUtils`), with one notable exception: `AnnotatedTypes`.

Equality for annotations

`AnnotationMirror` is an interface that is implemented both by javac and the Checker Framework. The documentation of `AnnotationMirror` says, "Annotations should be compared using the equals method. There is no guarantee that any particular annotation will always be represented by the same object." The second sentence is true, but the first sentence is wrong. You should never compare `AnnotationMirror`s using `equals()`, which (for some implementations) is reference equality. `AnnotationUtils` has various methods that should be used instead. Also, `AnnotationMirrorMap` and `AnnotationMirrorSet` can be used.

37.14.2 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-checking annotation processor is one that extends `AbstractTypeProcessor`; it gets 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 method is where you would insert any sort of method call to walk the AST, and the latter just returns a constant indicating the source version that is supported. 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 37.7.1.

Learning more about javac

The javac compiler interfaces can be daunting to a newcomer, and their 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 `https://netbeans.apache.org/wiki/main/javahowto/Java_DevelopersGuide/` and the compiler mailing list archives at `https://mail.openjdk.org/pipermail/compiler-dev/` (subscribe at `https://mail.openjdk.org/mailman/listinfo/compiler-dev`).

37.15 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 38

Building an accumulation checker

This chapter describes how to build a checker for an accumulation analysis. If you want to *use* an existing checker, you do not need to read this chapter.

An *accumulation analysis* is a program analysis where the analysis abstraction is a monotonically increasing set — that is, the analysis learns new facts, and facts are never retracted. Typically, some operation in code is legal only when the set is large enough — that is, the estimate has accumulated sufficiently many facts.

The Called Methods Checker (Chapter 7, page 68) is an accumulation analysis. The Test Accumulation Checker is a simplified version of the Called Methods Checker that you can also use as a model.

Accumulation analysis is a special case of typestate analysis in which (1) the order in which operations are performed does not affect what is subsequently legal, and (2) the accumulation does not add restrictions; that is, as more operations are performed, more operations become legal. Unlike a traditional typestate analysis, an accumulation analysis does not require an alias analysis for soundness. It can therefore be implemented as a flow-sensitive type system.

The rest of this chapter assumes you have read how to create a checker (Chapter 37, page 250).

Defining type qualifiers Define 2 or 3 type qualifiers.

- The "accumulator" type qualifier has a single argument: a `String[]` named `value` that defaults to an empty array. Note that the Checker Framework's support for accumulation analysis requires you to accumulate a string representation of whatever you are accumulating. For example, when accumulating which methods have been called, you might choose to accumulate method names.
  The accumulator qualifier should have no supertypes (`@SubtypeOf({})`) and should be the default qualifier in the hierarchy (`@DefaultQualifierInHierarchy`).
  An example of such a qualifier can be found in the Checker Framework's tests: TestAccumulation.java.
- Define a bottom type, analogous to TestAccumulationBottom.java. It should take no arguments, and should be a subtype of the accumulator type you defined earlier.
- Optionally, define a predicate annotation, analogous to TestAccumulationPredicate.java. It must have a single argument named `value` of type `String`. The predicate syntax supports
  - `||` disjunctions
  - `&&` conjunctions
  - `!` logical complement. `"!x"` means "it is not true that `x` was definitely accumulated" or, equivalently, "there is some path on which `x` was not accumulated". Note that this does **not** mean "`x` was not accumulated" — it is not a violation of the specification `"!x"` if `x` is accumulated on some paths, but not others.
  - `(...)` parentheses for precedence

Setting up the checker Define a new class that extends `AccumulationChecker`. It does not need any content.

Define a new class that extends `AccumulationAnnotatedTypeFactory`. You must create a new constructor whose only argument is a `BaseTypeChecker`. Your constructor should call one of the `super` constructors defined in `AccumulationAnnotatedTypeFactory` (which one depends on whether or not you defined a predicate annotation).

Adding accumulation logic Define a class that extends `AccumulationTransfer`. To update the estimate of what has been accumulated, override some method in `CFAbstractTransfer` to call `accumulate`.

For example, to accumulate the names of methods called, a checker would override `CFAbstractTransfer.visitMethodInvocation` to: call `super` to get a `TransferResult`, compute the method name from the `MethodInvocationNode`, and then call `accumulate`.

Enforcing program properties At this point, your checker ensures that all annotations are consistent with one another and the source code, and it flow-sensitively refines the annotations. To enforce properties in the code being type-checked, write type rules (Section 37.7) that are specific to your type system.

38.1 Publications

The paper "Accumulation Analysis" [KSSE22] describes theoretical properties of accumulation analysis. The papers "Verifying Object Construction" [KRS+20] and "Lightweight and modular resource leak verification" [KSSE21] describe specific accumulation analyses.

Chapter 39

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 (Section 39.1)
- Android Studio and the Android Gradle Plugin (Section 39.2)
- Ant (Section 39.3)
- Bazel (Section 39.4)
- Buck (Section 39.5)
- Command line, via Checker Framework javac wrapper (Section 39.6)
- Command line, via JDK javac (Section 39.7)
- Eclipse (Section 39.8)
- Gradle (Section 39.9)
- IntelliJ IDEA (Section 39.10)
- javac diagnostics wrapper (Section 39.11)
- Lombok (Section 39.12)
- Maven (Section 39.13)
- NetBeans (Section 39.14)
- sbt (Section 39.15)
- tIDE (Section 39.16)

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 39.7. 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 or licensing restrictions.

39.1 Android

When creating an Android app, you may wish to use `checker-qual-android` whenever this document mentions `checker-qual`. This can lead to smaller dex files (smaller distributed apps).

The `checker-qual-android` artifact is identical to the `checker-qual` artifact, except that in `checker-qual-android` annotations have classfile retention. The default Android Gradle plugin retains types annotated with runtime-retention annotations in the main dex, but strips out class-retention annotations.

39.2 Android Studio and the Android Gradle Plugin

Android Studio 3.0 and later, and Android Gradle Plugin 3.0.0 and later, support type annotations. (See `https://developer.android.com/studio/write/java8-support` for more details.) This section explains how to configure your Android project to use the Checker Framework. All the changes should be made to the module's `build.gradle` file — not the project's `build.gradle` file.

1. 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", "")
        }
    }
}
```

2. Declare the Checker Framework dependencies:

```
dependencies {
    ... existing dependencies...
    ext.checkerFrameworkVersion = "4.4.0"
    implementation "org.checkerframework:checker-qual-android:${checkerFrameworkVersion}"
    // or if you use no annotations in source code the above line could be
    // compileOnly "org.checkerframework:checker-qual-android:${checkerFrameworkVersion}"
    annotationProcessor "org.checkerframework:checker:${checkerFrameworkVersion}"
}
```

3. To run the checkers, build using the `checkTypes` variant:

```
gradlew assembleCheckTypes
```

39.3 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, and that the `CHECKERFRAMEWORK` environment variable is set.

1. Set the `cfJavac` property:

```
<property environment="env"/>
<property name="checkerframework" value="${env.CHECKERFRAMEWORK}" />
<condition property="cfJavac" value="javac.bat" else="javac">
  <os family="windows" />
</condition>
<presetdef name="cf.javac">
  <javac fork="yes" executable="${checkerframework}/checker/bin/${cfJavac}" >
    <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 cf.javac instead of javac -->
  <cf.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"/>
    ...
  </cf.javac>
</target>
```

Fill in each ellipsis (...) from the original compilation target. However, do not copy any `fork=` setting from the original `<javac>` task invocation.
If your original compilation target set the bootclasspath, then you cannot use the javac wrapper script as the above instructions do. You should edit your Ant buildfile to make invocations similar to those described in Section 39.7, but accommodating your bootclasspath.
In the example, the target is named `check-nullness`, but you can name it whatever you like.

39.3.1 Explanation

This section explains each part of the Ant task.

1. Definition of `cf.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 correct version of `javac`.
   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.

39.4 Bazel

Bazel is a build system maintained by Google.

To run the Checker Framework, see Stack Overflow discussion 48555320 (`https://stackoverflow.com/questions/48555320/`).

39.5 Buck

Buck is a build system maintained by Meta.

Buck has support for annotation processors, but that support is undocumented because the Buck maintainers may change the syntax in the future and they don't wish to ever change anything that is documented. You can learn more about Buck and annotation processors at these URLs: `https://stackoverflow.com/questions/32915721/documentation-for-annotation-processors-buck`, `https://github.com/facebook/buck/issues/85`.

Here is an example `BUCK` build file showing how to enable the Checker Framework:

```
prebuilt_jar(
    name = 'checker-framework',
```

```
    binary_jar = 'checker-4.4.0.jar',
    visibility = [ 'PUBLIC' ]
)

prebuilt_jar(
    name = 'checker-qual',
    binary_jar = 'checker-qual-4.4.0.jar',
    visibility = [ 'PUBLIC' ]
)

java_library (
    name = 'hello',
    srcs = glob(['src/main/java/**/*.java']),
    java_version = '17',
    provided_deps = [ ':checker-framework', ':checker-qual' ],
# To add annotation processing
    annotation_processors = [ 'org.checkerframework.checker.units.UnitsChecker' ],
    annotation_processor_deps = [ ':checker-framework', ':checker-qual' ],
)
```

39.5.1 Troubleshooting

Use `-verbose 8` to see diagnostic output including a command line that ought to be equivalent to Buck's use of an in-process Java compiler. For example, you might run

```
./buck clean
./buck fetch //...
./buck build --verbose 8 //app:main
```

However, the command lines that Buck prints with `-v 8` are *not* what Buck actually executes. Buck's invocation of the Checker Framework may produce errors that cannot be reproduced from the command line. This is because Buck uses in-process compilation, potentially multi-threaded, and because Buck may use multiple classloaders that are cached.

You can force Buck to run an external javac, thus behaving exactly like the command line that its diagnostics print. Use the configuration flag `-c tools.javac=`*`path-to-javac`*, for example:

```
./buck clean && ./buck fetch //... && ./buck build //app:main -c tools.javac=$(which javac)
```

39.6 Command line, via Checker Framework javac wrapper

To perform pluggable type-checking from the command line, run the `javac` command that ships with the Checker Framework. This is called the "Checker Framework compiler" or the "Checker Framework `javac` wrapper". It is exactly the same as the OpenJDK `javac` compiler, with one small difference: it includes the Checker Framework jar file on its classpath.

You cannot use the `javac` wrapper if you wish to set the bootclasspath.

There are three ways to use the Checker Framework compiler from the command line. You can use any one of them. However, if you are using the Windows command shell, you must use the last one. Adjust the pathnames if you have installed the Checker Framework somewhere other than `$HOME/checker-framework-4.4.0/`.

- Option 1: Add directory `.../checker-framework-4.4.0/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=${HOME}/checker-framework-4.4.0
export PATH=${CHECKERFRAMEWORK}/checker/bin:${PATH}
```

After editing the file, log out and back in to ensure that the environment variable setting takes effect.

- Option 2: Whenever this document tells you to run `javac`, instead run `$CHECKERFRAMEWORK/checker/bin/javac`.
  You can simplify this by introducing an alias `javacheck`. Then, whenever this document tells you to run `javac`, instead run `javacheck`. Here is the syntax for your `~/.bashrc`, `~/.profile`, or `~/.bash_profile` file:

  ```
  export CHECKERFRAMEWORK=${HOME}/checker-framework-4.4.0
  alias javacheck='$CHECKERFRAMEWORK/checker/bin/javac'
  ```

  After editing the file, log out and back in to ensure that the environment variable setting and alias take effect.
- 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" -cp "myclasspath" -processor nullness MyFile
  ```

  You can simplify the above command by introducing an alias `javacheck`. Then, whenever this document tells you to run `javac`, instead run `javacheck`. For example:

  ```
  # Unix
  export CHECKERFRAMEWORK=${HOME}/checker-framework-4.4.0
  alias javacheck='java -jar "$CHECKERFRAMEWORK/checker/dist/checker.jar"'

  # Windows
  set CHECKERFRAMEWORK = C:\Program Files\checker-framework-4.4.0\
  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`.)

39.7 Command line, via JDK javac

This section explains how to use the Checker Framework with the OpenJDK or OracleJDK `javac`, rather than with the `javac` wrapper script described in Section 39.6.

This section assumes you have downloaded the Checker Framework release zip and set the environment variable `CHECKERFRAMEWORK` to point to the unzipped directory. Alternately, you can get all of the jars mentioned in this section from Maven Central:

- `checker-qual.jar`: `https://repo1.maven.org/maven2/org/checkerframework/checker-qual/4.4.0/checker-qual-4.4.0.jar`
- `checker-util.jar`: `https://repo1.maven.org/maven2/org/checkerframework/checker-util/4.4.0/checker-util-4.4.0.jar`
- `checker.jar`: `https://repo1.maven.org/maven2/org/checkerframework/checker/4.4.0/checker-4.4.0-all.jar`

These instructions should work on JDK 17 or later, but we only test with supported versions of Java: JDK 17, JDK 21, and JDK 25.

39.7.1 Non-modularized code

To compile non-modularized code:

```
javac \
-J--add-exports=jdk.compiler/com.sun.tools.javac.api=ALL-UNNAMED \
```

```
-J--add-exports=jdk.compiler/com.sun.tools.javac.code=ALL-UNNAMED \
-J--add-exports=jdk.compiler/com.sun.tools.javac.file=ALL-UNNAMED \
-J--add-exports=jdk.compiler/com.sun.tools.javac.main=ALL-UNNAMED \
-J--add-exports=jdk.compiler/com.sun.tools.javac.model=ALL-UNNAMED \
-J--add-exports=jdk.compiler/com.sun.tools.javac.parser=ALL-UNNAMED \
-J--add-exports=jdk.compiler/com.sun.tools.javac.processing=ALL-UNNAMED \
-J--add-exports=jdk.compiler/com.sun.tools.javac.tree=ALL-UNNAMED \
-J--add-exports=jdk.compiler/com.sun.tools.javac.util=ALL-UNNAMED \
-J--add-opens=jdk.compiler/com.sun.tools.javac.comp=ALL-UNNAMED \
-processorpath $CHECKERFRAMEWORK/checker/dist/checker.jar \
-cp $CHECKERFRAMEWORK/checker/dist/checker-qual.jar \
-processor org.checkerframework.checker.nullness.NullnessChecker
```

The arguments are explained above, except for two of them:

1. `-J-add-opens=jdk.compiler/com.sun.tools.javac.comp=ALL-UNNAMED` which opens the `jdk.compiler/com.sun.tools` package. This is required because the Checker Framework reflectively accesses private members of this package.
2. `-J-add-exports=...=ALL-UNNAMED` which exports the listed packages. This is required because the Checker Framework accesses non-exported members of these packages.

If these options are missing, `javac` may issue one of the following warnings:

```
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.checkerframework.javacutil.Resolver
  (file:$CHECKERFRAMEWORK/checker/dist/checker.jar) to method com.sun.tools.javac.comp.Resolve.findMethod(
WARNING: Please consider reporting this to the maintainers of org.checkerframework.javacutil.Resolver
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
```

```
java.lang.IllegalAccessError: class org.checkerframework.javacutil.AbstractTypeProcessor
  (in unnamed module ...) cannot access class com.sun.tools.javac.processing.JavacProcessingEnvironment
  (in module jdk.compiler) because module jdk.compiler does not export com.sun.tools.javac.processing
  to unnamed module ...
```

39.7.2 Modularized code

To compile a module, first add `requires org.checkerframework.checker.qual;` to your `module-info.java`. The Checker Framework inserts inferred annotations into bytecode even if none appear in source code, so you must do this even if you write no annotations in your code.

```
javac \
  -J--add-exports=jdk.compiler/com.sun.tools.javac.api=ALL-UNNAMED \
  -J--add-exports=jdk.compiler/com.sun.tools.javac.code=ALL-UNNAMED \
  -J--add-exports=jdk.compiler/com.sun.tools.javac.file=ALL-UNNAMED \
  -J--add-exports=jdk.compiler/com.sun.tools.javac.main=ALL-UNNAMED \
  -J--add-exports=jdk.compiler/com.sun.tools.javac.model=ALL-UNNAMED \
  -J--add-exports=jdk.compiler/com.sun.tools.javac.parser=ALL-UNNAMED \
  -J--add-exports=jdk.compiler/com.sun.tools.javac.processing=ALL-UNNAMED \
  -J--add-exports=jdk.compiler/com.sun.tools.javac.tree=ALL-UNNAMED \
  -J--add-exports=jdk.compiler/com.sun.tools.javac.util=ALL-UNNAMED \
  -J--add-opens=jdk.compiler/com.sun.tools.javac.comp=ALL-UNNAMED \
  -processorpath $CHECKERFRAMEWORK/checker/dist/checker.jar \
  --module-path $CHECKERFRAMEWORK/checker/dist/checker-qual.jar \
  -processor org.checkerframework.checker.nullness.NullnessChecker
```

`checker-qual.jar` must be on the module path rather than the classpath, but do not put `checker.jar` on the processor module path, as it is not modularized.

39.8 Eclipse

You need to run the Checker Framework via a build tool (Ant, Gradle, Maven, etc.), 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 an annotation processor that works with both compilers.)

There is no dedicated Eclipse plug-in for running the Checker Framework, but it's still easy to run the Checker Framework. First, create a target/task in your build system to run the Checker Framework. Then, run the target/task from Eclipse. Section 39.8.1 gives details for Ant, but other build systems are similar.

39.8.1 Using an Ant task

Add an Ant target as described in Section 39.3. You can run the Ant target by executing the following steps (instructions copied from `https://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.

39.8.2 Troubleshooting Eclipse

Eclipse issues an "Unhandled Token in @SuppressWarnings" warning if you write a `@SuppressWarnings` annotation containing a string that Eclipse 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.

39.9 Gradle

To run a checker on a project that uses the Gradle build system, use the Checker Framework Gradle plugin (`https://github.com/typetools/checker-framework-gradle-plugin`). Its documentation explains how to use it.

39.10 IntelliJ IDEA

This section tells you how to make IntelliJ IDEA automatically run a checker on every compile for Java projects and/or modules.

If your project uses a build tool (Ant, Gradle, Maven, etc.), *do not* use the instructions in this section. Follow the instructions for that build tool instead (they are in a different section of this chapter). To compile your project, run the build tool (possibly from within IntelliJ IDEA, possibly not).

39.10.1 Running a checker on every IntelliJ compilation

If your project does not use a build tool, use the following instructions to run a checker within IntelliJ IDEA on every compile:

1. Change the project SDK to 17 or later, as explained at `https://www.jetbrains.com/help/idea/sdk.html#change-project-sdk`.
2. Make sure the language level for your project is 8 or higher, as explained at `https://www.jetbrains.com/help/idea/project-page.html`.
3. Create and configure an annotation profile, following the instructions at `https://www.jetbrains.com/help/idea/annotation-processors-support.html`. When configuring the profile:
   (a) Add `.../checker-framework/checker/dist/checker.jar` to the processor path.
   (b) Add checkers to be run during compilation by writing the fully-qualified name of the checker in the "Processor FQ Name" section. An example of a fully-qualified checker name is "org.checkerframework.checker.nullness.NullnessChec
4. Add `.../checker-framework/checker/dist/checker-qual.jar` as a dependency to all modules you wish to type-check. (They should all have been associated with the annotation profile above.) Instructions appear at `https://www.jetbrains.com/help/idea/creating-and-managing-projects.html`.

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`.

39.10.2 Running a checker on every IntelliJ change or save

To make IntelliJ compile on every change or save, follow the instructions at `https://www.jetbrains.com/help/idea/compiling-applications.html#auto-build`.

You can also configure IntelliJ to automatically save (and thus automatically compile) your work periodically. Instructions appear at `https://www.jetbrains.com/help/idea/system-settings.html#sync`.

39.11 javac diagnostics wrapper

The javac diagnostics wrapper project can post-process the javac diagnostics output into other formats, such as the LSP (Language Server Protocol) JSON style.

39.12 Lombok

Project Lombok (`https://projectlombok.org/`) is a library that generates getter, setter, and builder methods, among other features. For example, if you declare a field:

```
@Getter @Setter
private @Regex String role;
```

then Lombok will generate getter and setter methods:

```
public @Regex String getRole() { return role; }
public void setRole(@Regex String role) { this.role = role; }
```

39.12.1 Annotations on generated code

As illustrated in the example above, Lombok copies type annotations from fields to generated methods, when the user writes Lombok's `@Getter`, `@Setter`, and `@Builder` annotations. Lombok does so only for certain type annotations (including all annotations in the Checker Framework distribution); see variable `BASE_COPYABLE_ANNOTATIONS` in file `HandlerUtil.java`.

To make Lombok copy other type annotations from fields to generated code, add those type annotations to the `lombok.copyableAnnotations` configuration key in your `lombok.config` file. For example:

```
lombok.copyableAnnotations += my.checker.qual.MyTypeAnnotation
```

Directory `docs/examples/lombok` contains an example Gradle project that augments the configuration key.

39.12.2 Type-checking code with Lombok annotations

If you run the Checker Framework and Lombok in the same `javac` invocation, the Checker Framework cannot type-check a class that contains Lombok annotations. The way that Lombok changes the class prevents the Checker Framework from seeing any of the class. (The Checker Framework works fine on classes that do not contain Lombok annotations, including if they call Lombok-generated code.)

Therefore, you must run the Checker Framework in a postpass after the `javac` that runs Lombok has completed. Use the Delombok tool (distributed with Lombok) to generate Java source code, then run the Checker Framework on that. The Checker Framework Gradle plugin does this for you automatically.

39.12.3 Maven projects using Lombok

The Lombok Maven Plugin can be used to delombok source code for Maven projects. The Lombok Maven Plugin FAQ discusses how to configure the plugin when code using Lombok is mixed with standard Java code. Once the Lombok Maven plugin is configured correctly, the Checker Framework Maven configuration can be added independently.

39.13 Maven

If you use the Maven tool, then you can enable Checker Framework checkers by following the instructions below. These instructions have been tested with Maven 3.9.3, and they should work with any 3.x version of Maven.

See the directory `docs/examples/MavenExample/` for examples of the use of Maven build files. This example can be used to verify that Maven is correctly downloading the Checker Framework from the Maven Central Repository and executing it.

Please note that the `-AoutputArgsToFile` command-line option (see Section 37.12.5) and shorthands for built-in checkers (see Section 2.2.4) 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.

Debugging your Maven configuration can be tricky because of a bug in maven-compiler-plugin versions before 3.10.1, and still affecting some users with 3.11.0 (`https://issues.apache.org/jira/browse/MCOMPILER-434`): Maven does not report annotation processor exceptions, even when the `-X` command-line argument is passed.

1. Declare a dependency on the Checker Framework artifacts from Maven Central. Find the existing `<dependencies>` section (not within `<dependencyManagement>`, but somewhere else) and add the following new `<dependency>` items:

   ```
   <dependencies>
     ... existing <dependency> items ...

     <!-- Annotations from the Checker Framework: nullness, interning, locking, ... -->
   ```

```
    <dependency>
      <groupId>org.checkerframework</groupId>
      <artifactId>checker-qual</artifactId>
      <version>4.4.0</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:*"
```

2. Direct the Maven compiler plugin to use the desired checkers by creating a new profile as shown below (the example uses the Nullness Checker).
   If your POM file does not already use the `maven-compiler-plugin` plugin, you can use the text below as is. If your POM file already uses the plugin, copy and edit the first profile to incorporate the existing configuration. If the existing configuration uses `proc:none`, you need to remove that or comment it out.

```
<profiles>
  <profile>
    <id>checkerframework</id>
    <!-- If you omit the activation block, run mvn with "-P checkerframework" to run checkers. -->
    <activation>
      <jdk>[17,)</jdk>
    </activation>
    <build>
      <plugins>
        <plugin>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.11.0</version>
          <configuration>
            <fork>true</fork> <!-- Must fork or else JVM arguments are ignored. -->
            <annotationProcessorPaths>
              <path>
                <groupId>org.checkerframework</groupId>
                <artifactId>checker</artifactId>
                <version>4.4.0</version>
              </path>
            </annotationProcessorPaths>
            <annotationProcessors>
              <!-- Add all the checkers you want to enable here -->
              <annotationProcessor>org.checkerframework.checker.nullness.NullnessChecker</annotationProcessor>
            </annotationProcessors>
            <compilerArgs>
              <arg>-J--add-exports=jdk.compiler/com.sun.tools.javac.api=ALL-UNNAMED</arg>
              <arg>-J--add-exports=jdk.compiler/com.sun.tools.javac.code=ALL-UNNAMED</arg>
              <arg>-J--add-exports=jdk.compiler/com.sun.tools.javac.file=ALL-UNNAMED</arg>
              <arg>-J--add-exports=jdk.compiler/com.sun.tools.javac.main=ALL-UNNAMED</arg>
              <arg>-J--add-exports=jdk.compiler/com.sun.tools.javac.model=ALL-UNNAMED</arg>
              <arg>-J--add-exports=jdk.compiler/com.sun.tools.javac.parser=ALL-UNNAMED</arg>
              <arg>-J--add-exports=jdk.compiler/com.sun.tools.javac.processing=ALL-UNNAMED</arg>
              <arg>-J--add-exports=jdk.compiler/com.sun.tools.javac.tree=ALL-UNNAMED</arg>
              <arg>-J--add-exports=jdk.compiler/com.sun.tools.javac.util=ALL-UNNAMED</arg>
              <arg>-J--add-opens=jdk.compiler/com.sun.tools.javac.comp=ALL-UNNAMED</arg>
              <arg>-Xmaxerrs</arg>
              <arg>10000</arg>
              <arg>-Xmaxwarns</arg>
              <arg>10000</arg>
              <!-- <arg>-Awarns</arg> --> <!-- -Awarns turns type-checking errors into warnings. -->
            </compilerArgs>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

If you wish to run checkers while compiling your source code but not your tests, wrap the `<configuration>...</configuration>` within

```
<executions>
  <execution>
    <id>default-compile</id>
    ...
  </execution>
</executions>
```

To compile without using the Checker Framework, pass `-P '!checkerframework'` on the Maven command line.

Warning: adding

```
<compilerArgs>
  ...
  <arg>-verbose</arg>
```

may suppress warnings from the stub parser.

If you use `animal-sniffer-maven-plugin`, you must upgrade to version 1.20 or later. Versions 1.19 and earlier are broken. If necessary, you can disable animal sniffer by modifying your POM file's properties:

```
<properties>
  ... existing <property> items ...

  <!-- Needed for animal-sniffer-maven-plugin version 1.19 which is broken (version 1.20 is fixed). -->
  <animal.sniffer.skip>true</animal.sniffer.skip>
</properties>
```

39.13.1 Maven, with a locally-built version of the Checker Framework

To use a locally-built version of the Checker Framework, first run:

```
./gradlew publishToMavenLocal
```

Then use the Maven instructions, but modify the version number for the Checker Framework artifacts. Instead of `4.4.0`, use the version number that is output when you run `./gradlew version`.

39.13.2 Maven troubleshooting

If you get an error like "module jdk.compiler does not export com.sun.tools.javac.processing to unnamed module" (discussed in Section 39.7.1) even though your `pom.xml` contains lines such as

```
<arg>-J--add-exports=jdk.compiler/com.sun.tools.javac.processing=ALL-UNNAMED</arg>
```

then perhaps you have omitted this line from your `pom.xml`:

```
        <fork>true</fork> <!-- Must fork or else JVM arguments are ignored. -->
```

39.14 NetBeans

There are two approaches to running a checker in NetBeans: modifying the project properties, or using a custom Ant target.

Note: The "compile and save" action in NetBeans 8.1 automatically runs the Checker Framework, but this functionality has not yet been incorporated into NetBeans 8.2. Additionally, JDK annotations are currently unavailable on NetBeans 8.1 and 8.2.

39.14.1 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.

39.14.2 Adding a checker via an Ant target

1. Set the `cfJavac` property:

```
<property environment="env"/>
<property name="checkerframework" value="${env.CHECKERFRAMEWORK}" />
<condition property="cfJavac" value="javac.bat" else="javac">
  <os family="windows" />
</condition>
<presetdef name="cf.javac">
  <javac fork="yes" executable="${checkerframework}/checker/bin/${cfJavac}" >
    <compilerarg value="-version"/>
    <compilerarg value="-implicit:class"/>
  </javac>
</presetdef>
```

2. Override the `-init-macrodef-javac-with-processors` target to use `cf.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="${empty.dir}"/>
          <mkdir dir="@{apgeneratedsrcdir}"/>
          <cf.javac debug="@{debug}" deprecation="${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/>
            </cf.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.

39.15 sbt

sbt is a build tool for Scala, Java, and more.

Adjust the `-processor` command-line argument for the processor(s) you wish to run (see Section 2.2).

```
javacOptions ++= Seq(
    "-J--add-exports=jdk.compiler/com.sun.tools.javac.api=ALL-UNNAMED",
    "-J--add-exports=jdk.compiler/com.sun.tools.javac.code=ALL-UNNAMED",
    "-J--add-exports=jdk.compiler/com.sun.tools.javac.file=ALL-UNNAMED",
    "-J--add-exports=jdk.compiler/com.sun.tools.javac.main=ALL-UNNAMED",
    "-J--add-exports=jdk.compiler/com.sun.tools.javac.model=ALL-UNNAMED",
    "-J--add-exports=jdk.compiler/com.sun.tools.javac.parser=ALL-UNNAMED",
    "-J--add-exports=jdk.compiler/com.sun.tools.javac.processing=ALL-UNNAMED",
    "-J--add-exports=jdk.compiler/com.sun.tools.javac.tree=ALL-UNNAMED",
    "-J--add-exports=jdk.compiler/com.sun.tools.javac.util=ALL-UNNAMED",
    "-J--add-opens=jdk.compiler/com.sun.tools.javac.comp=ALL-UNNAMED",
    "-processorpath $CHECKERFRAMEWORK/checker/dist/checker.jar",
    "-cp $CHECKERFRAMEWORK/checker/dist/checker-qual.jar",
    "-processor org.checkerframework.checker.nullness.NullnessChecker"
  )
```

39.15.1 For modularized code

For compiling modularized code:

```
javacOptions ++= Seq(
    "-J--add-exports=jdk.compiler/com.sun.tools.javac.api=ALL-UNNAMED",
    "-J--add-exports=jdk.compiler/com.sun.tools.javac.code=ALL-UNNAMED",
    "-J--add-exports=jdk.compiler/com.sun.tools.javac.file=ALL-UNNAMED",
    "-J--add-exports=jdk.compiler/com.sun.tools.javac.main=ALL-UNNAMED",
    "-J--add-exports=jdk.compiler/com.sun.tools.javac.model=ALL-UNNAMED",
    "-J--add-exports=jdk.compiler/com.sun.tools.javac.parser=ALL-UNNAMED",
    "-J--add-exports=jdk.compiler/com.sun.tools.javac.processing=ALL-UNNAMED",
    "-J--add-exports=jdk.compiler/com.sun.tools.javac.tree=ALL-UNNAMED",
    "-J--add-exports=jdk.compiler/com.sun.tools.javac.util=ALL-UNNAMED",
    "-J--add-opens=jdk.compiler/com.sun.tools.javac.comp=ALL-UNNAMED",
    "-processorpath $CHECKERFRAMEWORK/checker/dist/checker.jar",
    "--module-path $CHECKERFRAMEWORK/checker/dist/checker-qual.jar",
    "-processor org.checkerframework.checker.nullness.NullnessChecker"
  )
```

39.16 tIDE

tIDE, an open-source Java IDE, supports the Checker Framework. You can download it from `https://sourceforge.net/projects/tide/`.

39.17 Type inference tools

A type inference tool infers type annotations for a program's method signatures and fields, so that the programmer does not need to manually annotate the program's source code. Section 35.1 lists type inference tools.

Chapter 40

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:

40.1: Motivation for pluggable type-checking
40.1.1: I don't make type errors, so would pluggable type-checking help me?
40.1.2: Should I use pluggable types (type qualifiers), or should I use Java subtypes?

40.2: Getting started
40.2.1: How do I get started annotating an existing program?
40.2.2: Which checker should I start with?
40.2.3: How can I join the checker-framework-dev mailing list?

40.3: Usability of pluggable type-checking
40.3.1: Are type annotations easy to read and write?
40.3.2: Will my code become cluttered with type annotations?
40.3.3: Will using the Checker Framework slow down my program? Will it slow down the compiler?
40.3.4: How do I shorten the command line when invoking a checker?
40.3.5: Method pre-condition contracts, including formal parameter annotations, make no sense for public methods

40.4: How to handle warnings and errors
40.4.1: What should I do if a checker issues a warning about my code?
40.4.2: Why does a checker issue a warning even though the code checks a property?
40.4.3: What does a certain Checker Framework warning message mean?
40.4.4: What do square brackets mean in a Checker Framework warning message?
40.4.5: Can a pluggable type-checker guarantee that my code is correct?
40.4.6: What guarantee does the Checker Framework give for concurrent code?
40.4.7: How do I make compilation succeed even if a checker issues errors?
40.4.8: Why does the checker always say there are 100 errors or warnings?
40.4.9: Why does the Checker Framework report an error regarding a type I have not written in my program?
40.4.10: Why does the Checker Framework accept code on one line but reject it on the next?
40.4.11: How can I do run-time monitoring of properties that were not statically checked?

40.5: False positive warnings
40.5.1: What is a “false positive” warning?
40.5.2: How can I improve the Checker Framework to eliminate a false positive warning?
40.5.3: Why doesn’t the Checker Framework infer types for fields and method return types?
40.5.4: Why are some operations legal in constructors but not in methods?
40.5.5: Why doesn’t the Checker Framework track relationships between variables?
40.5.6: Why isn’t the Checker Framework path-sensitive?

40.6: Syntax of type annotations
40.6.1: What is a “receiver”?
40.6.2: What is the meaning of an annotation after a type, such as `@NonNull Object @Nullable`?
40.6.3: What is the meaning of array annotations such as `@NonNull Object @Nullable []`?
40.6.4: What is the meaning of varargs annotations such as `@English String @NonEmpty ...`?
40.6.5: What is the meaning of a type qualifier at a class declaration?
40.6.6: How are type qualifiers written on upper and lower bounds?
40.6.7: Why shouldn’t a qualifier apply to both types and declarations?
40.6.8: How do I annotate a fully-qualified type name?
40.6.9: What is the difference between type annotations and declaration annotations?
40.6.10: How should type annotations be formatted in source code? Where should I write type annotations?
40.6.11: How does the Checker Framework handle obsolete declaration annotations?
40.6.12: How do I convert from other tools’ declaration annotations to type annotations?

40.7: Semantics of type annotations
40.7.1: How can I handle typestate, or phases of my program with different data properties?
40.7.2: Why are explicit and implicit bounds defaulted differently?
40.7.3: How should I annotate code that uses generics?
40.7.4: Why are type annotations declared with `@Retention(RetentionPolicy.RUNTIME)`?

40.8: Creating a new checker
40.8.1: How do I create a new checker?
40.8.2: What properties can and cannot be handled by type-checking?
40.8.3: Why is there no declarative syntax for writing type rules?

40.9: Tool questions
40.9.1: How does pluggable type-checking work?
40.9.2: What classpath is needed to use an annotated library?
40.9.3: Why do `.class` files contain more annotations than the source code?
40.9.4: Is there a type-checker for managing checked and unchecked exceptions?
40.9.5: The Checker Framework runs too slowly
40.9.6: What does the Checker Framework version number mean?

40.10: Relationship to other tools
40.10.1: Why not just use a bug detector (like SpotBugs or Error Prone)?
40.10.2: How does the Checker Framework compare with Eclipse’s null analysis?
40.10.3: How does the Checker Framework compare with NullAway?
40.10.4: How does the Checker Framework compare with JSpecify?
40.10.5: How does the Checker Framework compare with the EISOP Checker Framework?
40.10.6: How does the Checker Framework compare with the JDK’s `Optional` type?
40.10.7: How does pluggable type-checking compare with JML?
40.10.8: Is the Checker Framework an official part of Java?

40.10.9: What is the relationship between the Checker Framework and JSR 305?
40.10.10: What is the relationship between the Checker Framework and JSR 308?

40.1 Motivation for pluggable type-checking

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

Occasionally, a developer claims to make 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.

40.1.2 Should I use pluggable types (type qualifiers), or should I use 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 subtype (interfaces or subclasses) or a type qualifier. As an example, suppose 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. You may need to write a conversion at every call to and return from a library call; these conversions add clutter, reduce performance, and (by creating new objects) break object equality properties. Brian Goetz also notes some problems in an article on the pseudo-typedef antipattern [Goe06].
It is possible to add annotations to code you do not maintain or cannot change; for example, you can write library annotations (see Chapter 36).

Composability A Java object has just one class at run time. If you care about multiple correctness properties, then using the Java type system requires a combinatorial number of classes (all possible combinations). By contrast, arbitrary type qualifiers can be used at the same time.

No new bugs A code change may introduce bugs, whereas adding annotations does not change the run-time behavior.

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 26).
- It includes special-case logic for type-checking specific methods, such as the Nullness Checker's treatment of `Map.get`.

Partial annotation You can add type annotations to just part of your code, and suppress warnings at entry points to the annotated portion. By contrast, using a new Java type requires using it everywhere or writing conversion routines at every entry point.

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 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 refinement (Section 33.5), 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.

40.2 Getting started

40.2.1 How do I get started annotating an existing program?

See Section 2.4.2.

40.2.2 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 most often 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 31), 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. You will be the best judge of this for your own code, of course.

Some of the third-party checkers (see Chapter 31, page 188) have known bugs that limit their usability. (Report the ones that affect you, so the developers will prioritize fixing them.)

40.2.3 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.

Anyone is welcome to join `checker-framework-discuss@googlegroups.com` and send mail to it. This list is for user-focused discussions. It is not for submitting bug reports, which should use the issue tracker (Section 41.2).

40.3 Usability of pluggable type-checking

40.3.1 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.

40.3.2 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.

40.3.3 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 javac compiler emits identical bytecodes whether or not the Checker Framework is used, 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. It can increase the compilation time by 2–10 times — or more, if you run many pluggable type-checkers at once. For workarounds, see Section 40.9.5.

40.3.4 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 ...`. You can use shorthand for built-in checkers, such as `javac -processor nullness ...` (see Section 2.2.4), or you can use auto-discovery to eliminate the need for the `-processor` command-line option (see Section 2.2.3).

40.3.5 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 in 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.

40.4 How to handle warnings and errors

40.4.1 What should I do if a checker issues a warning about my code?

For a discussion of this issue, see Section 2.4.5.

40.4.2 Why does a checker issue a warning even though the code checks a property?

Suppose a checker issues a warning about a line of code. One way to eliminate a warning is to perform a run-time test (Section 33.7.4), such as adding `if (x != null)` before a dereference of `x` to avoid a warning about a potential null pointer dereference. However, sometimes the checker issues a warning even though the code contains a run-time test. Two common reasons are non-determinism in the expression being tested and side effects to the expression after the run-time test. For details and examples, see Section 33.7.5.

40.4.3 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 open an issue (Section 41.2). 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.

40.4.4 What do square brackets mean in a Checker Framework warning message?

In a message like this:

```
found   : ? extends T extends @UnknownKeyFor Object
required: [extends @UnknownKeyFor Object super @UnknownKeyFor null]
```

the square brackets enclose the upper and lower bounds that the type parameter needs to be within.

40.4.5 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, if your code uses a library that has not been type-checked, then the library might be incorrect, or your code might misuse the library. As another 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 in some places where 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.

40.4.6 What guarantee does the Checker Framework give for concurrent code?

The Lock Checker (see Chapter 10) 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 type refinement (Section 33.7) that the Checker Framework can do.

40.4.7 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.

40.4.8 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.

40.4.9 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 33.5. In other cases, this is because type refinement has given an expression a more specific type than you wrote or than was defaulted; see Section 33.7. Note that an invocation of an impure method may cause the loss of all information that was determined via type refinement; see Section 33.7.5.

40.4.10 Why does the Checker Framework accept code on one line but reject it on the next?

Sometimes, the Checker Framework permits code on one line, but rejects the same code a few lines later:

```
if (myField != null) {
  myField.hashCode();   // no warning
  someMethod();
  myField.hashCode();   // warning about potential NullPointerException
}
```

The reason is explained in the section on type refinement (Section 33.7), which also tells how to specify the side effects of `someMethod` (Section 33.7.5).

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

Some properties are not checked statically (see Chapter 34 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 14.2.4). But, it would be better to have more general support that does not require the user to explicitly insert method calls.

40.5 False positive warnings

40.5.1 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 2.4.5). You also might want to report the false positive in the Checker Framework issue tracker (Section 41.2), 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 40.5.2).

40.5.2 How can I improve the Checker Framework to eliminate a false positive warning?

As noted in Section 40.5.1, *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 are not a good use of your time to report, nor of 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.

40.5.3 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 m() {
    String local = "%d";
    String.format(local, 5);    // Format String Checker verifies the call is safe
```

```
    String.format(field, 5);    // Format String Checker warns the call might not be safe
    String.format(method(), 5); // Format String Checker warns the 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 33.7), 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 35.2.) 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 down the type-checker.

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. The Checker Framework performs modular type-checking, not 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 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.

40.5.4 Why are some operations legal in constructors but not in methods?

Java permits you to initialize an instance field either in the constructor or in a field initializer. The Checker Framework behaves as if the field initializer assignments are performed at the beginning of the constructor (after superclass construction), because that is how Java executes the program.

Here is an example:

```
class MyClass1 {
  @Nullable Object field;
  MyClass1() {
    field = new Object();
    field.toString(); // no possible NullPointerException
  }
```

```
}

class MyClass2 {
  @Nullable Object field = new Object(); // Same initialization as for MyClass1
  MyClass2() {
    field.toString(); // no possible NullPointerException
  }
}

class MyClass3 {
  @Nullable Object field;
  void myMethod() {
    field.toString(); // Nullness Checker warns: possible NullPointerException
  }
}
```

For consistency, the Checker Framework treats the `MyClass1` and `MyClass2` constructors the same.

40.5.5 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 not estimate relationships between two variables, except for specific relationships listed in Section 33.8.

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 type refinement (Section 33.7):

```
  void nullnessExample2(@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 11, page 98), 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
}
```

40.5.6 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 a 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 40.5.5).

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.

40.6 Syntax of type annotations

There is also a separate FAQ for the type annotations syntax (`https://checkerframework.org/jsr308/jsr308-faq.html`).

40.6.1 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 a period and 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 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`.

A method in an anonymous class has a receiver, but it is not possible to write the receiver in the formal parameter list because there is no name for the type of `this`.

40.6.2 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 40.6.3).

40.6.3 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 40.6.10 explains how the Checker Framework handles declaration annotations in the meantime.

40.6.4 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.

40.6.5 What is the meaning of a type qualifier at a class declaration?

See Section 33.3.

40.6.6 How are type qualifiers written on upper and lower bounds?

See Section 32.1.2.

40.6.7 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 Section 40.6.1.

Another example of a type-and-declaration annotation that represents poor design is JCIP's `@GuardedBy` annotation [GPB+06]. As discussed in Section 10.6.1, it means two different things when applied to a field or a method. To reduce confusion and increase expressiveness, the Lock Checker (see Chapter 10) uses the `@Holding` annotation for one of these meanings, rather than overloading `@GuardedBy` with two distinct meanings.

A final example is `@Nullable` and `@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 (which was released in March 2014), and should not be necessary any longer.

40.6.8 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
```

40.6.9 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 these 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 expression
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.

40.6.10 How should type annotations be formatted in source code? Where should I write type annotations?

The Java Language Specification, section 9.7.4 says: "It is customary, though not required, to write declaration annotations before all other modifiers, and type annotations immediately before the type to which they apply."

A type annotation should be written immediately before the type it qualifies, on the same line. There should not be any modifiers (such as `public`) between them. This emphasizes that the type qualifier plus the Java base type is a logical unit, which aids people reading the code.

```
// Wrong formatting
@Positive
public int numItems;

// Correct formatting
public @Positive int numItems;

// Wrong formatting
@Nullable
public Object getThing() {
  ...
}

// Correct formatting
public @Nullable Object getThing() {
  ...
}
```

By contrast, declaration annotations are conventionally written on their own line.

```
// Wrong formatting
@Deprecated URL toURL() {
```

```
  ...
}

// Correct formatting
@Deprecated
URL toURL() {
  ...
}
```

The preferred order of modifiers is

1. declaration annotations
2. keyword modifiers such as `public` and `strictfp`
3. type annotations
4. type

Here is an example of a declaration annotation and a type annotation on a class declaration:

```
@Deprecated
public @Interned class Level {
  ...
}
```

The popular google-java-format tool formats type annotations incorrectly, with type annotations on their own line instead of on the same line as the type they modify. (Actually, you can see from method `JavaInputAstVisitor.typeAnnotations()` that it handles two of the many type annotations — `@Nullable` and `@NonNull` — correctly, but all other type annotations incorrectly! This would be fixed by merging pull request #802.) Until this bug in google-java-format is fixed, you have these options:

- Run google-java-format via Spotless and use the `formatAnnotations` step to correct the formatting of type annotations.
- Run google-java-format via run-google-java-format.

If you define a new type qualifier — for example, because you are using the Subtyping Checker (Chapter 30) or Fake Enum Checker (Chapter 9), or because you have created a new checker (Chapter 37) — then the above tools will format your new type qualifiers incorrectly, like declaration annotations. Here is how to fix the formatting.

- For Spotless, change your configuration to indicate the new type qualifiers. If you use the Spotless plugin for Gradle, edit your `build.gradle` file to change `formatAnnotations()` to (for example) `formatAnnotations().addTypeAnnotatio`
- For run-google-java-format, make a pull request to change variable `type_annotations` in file fixup-google-java-format.py.

40.6.11 How does the Checker Framework handle obsolete declaration annotations?

When a declaration annotation is an alias for a type annotation, 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, and write them in the correct location (Section 40.6.10).

Users who are using old Java 5–7 declaration annotations (for instance, from the FindBugs tool, which has not been maintained since 2015, and from its successor SpotBugs, which has not yet adopted type annotations, even though type annotations were added to Java in 2014) can use annotations in package `org.checkerframework.checker.nullness.compatqual` to avoid name conflicts. These are available in package `checker-compat-qual` on Maven Central. Once users are ready to upgrade to Java 8+ type annotations, those compatibility annotations are no longer necessary.

40.6.12 How do I convert from other tools' declaration annotations to type annotations?

In order to convert from another tool's declaration annotations to the Checker Framework's type annotations, you need to change imports, rename annotations, move annotations written on arrays (see Section 40.6.11), and move annotations to simple type names.

For example, if a user writes

```
import android.support.annotation.Nullable;
...
@Nullable
public Map.Entry<Object, Object> intersects() { ... }
```

and then changes the import to `org.checkerframework.checker.nullness.qual.Nullable`, the code will not compile. (See Section 41.1.3 for more details.) The user needs to also move the `@Nullable` annotation to a location where Java permits type annotations:

```
import org.checkerframework.checker.nullness.qual.Nullable;
...
public Map.@Nullable Entry<Object, Object> intersects() { ... }
```

40.7 Semantics of type annotations

40.7.1 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 Section 31.30, page 193), but typestate checkers have not been as extensively tested.

40.7.2 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 33.5.3):

```
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 way.

```
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 14.1.

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.

Here are some style guidelines:

- Omit the `extends` clause when possible. To indicate no constraints on type qualifiers, write `class MyClass<T>` rather than `class MyClass<T extends @TOPTYPEANNO Object>`.
- When you write an `Object` upper bound, give an explicit type annotation. That is, write `class C<T extends @DEFAULTANNO Object> { ... }` even though it is equivalent to writing `class C<T extends Object> { ... }`.
  If you write just `extends Object`, then someone who is reading the code might think that it is irrelevant (which it is in plain Java). Also, some IDEs will suggest removing it.

40.7.3 How should I annotate code that uses generics?

Suppose unannotated code contains a declaration of a type parameter `<T>`. How should you annotate that declaration?

This is really a question about Java's generic types, so if you understand Java generics, this question is moot. However, Java generics can be hard to understand, so here is a brief explanation of some concrete annotations, using nullness annotations for concreteness.

`<T>` Any type argument may be supplied for `T`.
It is equivalent to `<T extends @Nullable Object>`, because `@Nullable Object` is the top type in the type hierarchy.

`<T extends @Nullable Object>` Any type argument may be supplied for `T`.
It is equivalent to `<T>`, as noted above.

`<T extends Object>` `T` can be instantiated by any type whose qualifier is `@NonNull`.
The default annotation is `@NonNull`, and annotation defaults apply to type uses such as `Object` but not to type variables such as `T`. Therefore, `<T extends Object>` is equivalent to `<T extends @NonNull Object>`. It permits any type argument that is a subtype of `@NonNull Object`, which is any type argument whose qualifier is `@NonNull`, since `@NonNull` is the bottom type qualifier in the type qualifier hierarchy.

<T extends @NonNull Object> `T` can be instantiated by any type whose qualifier is `@NonNull`.
It is equivalent to `<T extends Object>`, as noted above.

<@Nullable T> `T` can be instantiated by any type whose qualifier is `@Nullable`.
The annotation `@Nullable` before `T` applies to `T`'s implicit lower bound. There is no explicit upper bound (that is, no `extends`), so the upper bound is the top type, `@Nullable Object`, just as for `<T>`, which was discussed above. Therefore, `<@Nullable T>` is the same as `<T super @Nullable void extends @Nullable Object>`, except that the latter is not legal Java.

<T super @Nullable String> `T` can be instantiated by any supertype of `@Nullable String`, which is any supertype of `String` (`Object`, `Serializable`, `CharSequence`, etc.) so long as its type qualifier is `@Nullable`.

<T super @NonNull String> `T` can be instantiated by any supertype of `@NonNull String`. Since `@NonNull` is the bottom type qualifier, the instantiating type can have any type qualifier.

For more details about how the Checker Framework supports generics and polymorphism, see Chapter 32, page 194.

40.7.4 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). To avoid these costs, a project can use the `checker-qual-android` dependency; as explained in Section 39.1, it is identical to `checker-qual`, except that in `checker-qual-android` annotations have `CLASS` retention.

For the purpose of static checking at compile time, `CLASS` retention is 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.

40.8 Creating a new checker

40.8.1 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 37 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 30.2 explains this checker and tells you how to run it.

40.8.2 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 calls 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.

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

A type system implementer can declaratively specify the type qualifier hierarchy (Section 37.5.2) and the type introduction rules (Section 37.8). However, the Checker Framework uses a procedural syntax for specifying type-checking rules (Section 37.7). 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.

40.9 Tool questions

40.9.1 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").

40.9.2 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.

40.9.3 Why do `.class` files contain more annotations than the source code?

A `.class` file contains an annotation on every type, as computed by defaulting rules; see Section 33.4. As a consequence, compiling against such a `.class` file with JDK 22 or later requires every type in its signatures to be on the classpath; see Section 41.1.2.

When an overridden method has a side effect annotation, the overriding method must have one too. However, if the side effect annotation is declared with the `@InheritedAnnotation` meta-annotation, the Checker Framework automatically adds the missing annotation. This is the case for most side effect annotations — these annotations are propagated from annotated libraries, such as the JDK, to your code.

40.9.4 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 19, page 135) is a type-and-effect system that is distributed with the Checker Framework.

40.9.5 The Checker Framework runs too slowly

Using a pluggable type-checker increases compile times by a factor of 2–10. Slow compilation speed is probably the worst thing about the Checker Framework.

To improve performance:

- Ensure that the Checker Framework has enough memory. The Checker Framework uses more memory than `javac` does, and Java's default heap limit is so small that the Checker Framework might spend most of its time in garbage collection. (If this is the case, the Checker Framework will issue a warning "Garbage collection consumed over 25% of CPU during the past minute.") To permit the Checker Framework to use up to 4 GB of memory, pass an argument like `-Xmx4g` or set an environment variable like `export _JAVA_OPTIONS=-Xmx4g`. Also consider upgrading to a more recent release of Java, because older versions of Java have worse memory management.
- Set your build system to perform *incremental compilation*. When compiling just a few source files (the size of a typical edit or commit), you won't notice the slowdown even if the Checker Framework is very slow. If you compile all files in a large project, you will definitely notice a slowdown. You should structure your build system to make compiling all files rare, by declaring dependencies and using caching. (Note: Maven lacks dependency-driven build and caching. If your project uses Maven, consider switching to a more capable build system such as Gradle.)
- Write generic type arguments. Often, generic type inference is the slowest part of type-checking. You can significantly speed up type-checking by explicitly writing a few generic type arguments. To determine where to write them, temporarily set `-AslowTypecheckingSeconds` to a small value, such as 1. Write type arguments where `slow.typechecking` warnings are issued.

If the Checker Framework is still too slow for you to run on every compilation, you can run it periodically, such as in a Git commit hook or in continuous integration.

The Checker Framework team does not currently have the resources to fix performance problems, but we welcome community contributions.

Here are some reasons that the Checker Framework is slow.

- It has to do all the same work as the compiler does, such as resolving overloading and overriding, inferring generics, and type-checking.
- Its analysis is general; it interprets user-defined type systems whereas `javac` hard-codes one type system and integrates it with other processing.
- Its analysis is much richer. `javac` can take certain shortcuts that are not correct for all possible type systems.
- It builds a control flow graph and performs a fixpoint analysis on it, which `javac` does not do.
- It manipulates multiple representations of data, including `javac`'s internal representation, source code, control flow graph, and its own internal representation. These transformations and lookups take time.

- It switches between dataflow analysis and type analysis, and this sometimes causes it to redo work.
- When running a compound or aggregate checker, it computes the control flow graph multiple times, and it makes multiple passes over the program rather than just one.

40.9.6 What does the Checker Framework version number mean?

As explained in Section 1.3, a change in the middle number of the Checker Framework version string indicates a possible incompatibility.

Another policy is "semantic versioning" as defined at `https://semver.org/`. This version number policy, unfortunately, does not define key terms, such as "backwards compatible bug fixes". It includes an escape hatch, "Semantic Versioning is all about conveying meaning by how the version number changes." It has not been updated in over a decade (since January 2013), has only ever been updated once (from 1.0.0 to 2.0.0), and has about 100 open issues. It is a proposal, not a standard or an official definition. Projects that claim to follow it often do so loosely.

If the Checker Framework strictly used semver.org's definition, every release would be a new major version, and the current version string would be past version 200.0.0. Always incrementing the major version number would not convey meaning to its users — for example, it would not indicate major new functionality or important behavior differences.

40.10 Relationship to other tools

40.10.1 Why not just use a bug detector (like SpotBugs or Error Prone)?

A pluggable type-checker is a verification tool that prevents or detects all errors of a given variety. If it issues no warnings, your code has no errors of a given variety (for details about the guarantee, see Section 2.3).

An alternate approach is to use a bug detector such as Error Prone, FindBugs [HP04, HSP05], SpotBugs, Jlint [Art01], PMD [Cop05], or the tools built into Eclipse (see Section 40.10.2) and IntelliJ. The NullAway and Eradicate tools are more like sound type-checking than bug detection, but all of those tools accept unsoundness — that is, false negatives or missed warnings — in exchange for analysis speed.

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

- A type-checker reports *all* errors in your code. If a type-checker reports no warnings, then you have a guarantee (a proof) of correctness (Section 40.4.5).
  A bug detector aims to find *some* of the most obvious errors. Even if a bug detector reports no warnings, there may still be errors in your code.
- A type-checker requires you to annotate your code with type qualifiers, or to run an inference tool that does so for you. That is, it requires you to write a specification, then it verifies that your code meets its specification.
  Some bug detectors do not require annotations. This means that it may be easier to get started running a bug detector. The tool makes guesses about the intended behavior of your code, leading to false alarms or missed alarms.
- A verification tool may issue more warnings for a programmer to investigate. Some bug detectors internally generate many warnings, then use heuristics to discard some of them. The cost is missed alarms, when the tool's heuristics classified the warnings as likely false positives and discarded them.
- A type-checker uses a more sophisticated and precise analysis. For example, it can take advantage of method annotations and annotations on generic type parameters, such as `List<@NonNull String>`. An example specific to the Nullness Checker (Chapter 3): no other tool correctly handles map keys or initialization.
  A bug detector does a more lightweight analysis. This means that a bug detector usually runs faster, giving feedback to the programmer more rapidly and avoiding slowdowns. Its analysis is often narrow, avoiding properties that are tricky to reason about or that might lead to false alarms. The cost is missed alarms, when analysis is too weak to find the errors.
- Neither type-checking nor bug detection subsumes the other. A type-checker finds problems that a bug detector cannot. A bug detector finds problems that a type-checker does not: there is no need for the type-checker to address style rules, when a bug detector is adequate.
  If your code is important to you, it is advantageous to run both types of tools.

For a case study that compared the nullness analysis of FindBugs (equivalent to SpotBugs), Jlint, PMD, and the Checker Framework, see section 6 of the paper "Practical pluggable types for Java" [PAC+08]. 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. A follow-up 10 years later found that Eclipse's nullness analysis found 0 of the errors, and IntelliJ's nullness analyses found 3 of the errors that the Nullness Checker found.

The JSR 308 [Ern08b] documentation also contains a discussion of related work.

40.10.2 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 31) 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 with good 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.) The Checker Framework allows the user to specify whether code will be run concurrently or not via the `-AconcurrentSemantics` command-line option (see Section 40.4.6).

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 15) 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.

In a case study, the Nullness Checker found 9 errors in a program, and Eclipse's analysis found 0.

40.10.3 How does the Checker Framework compare with NullAway?

NullAway is a lightweight, unsound type-checker whose aim is similar to that of the Nullness Checker (Chapter 3, page 31). For both tools, the user writes nullness annotations, and then the tool checks them.

NullAway is faster than the Nullness Checker and requires fewer annotations.

NullAway is unsound: even if NullAway issues no warnings, your code might crash with a null pointer exception. Two differences are that NullAway makes unchecked assumptions about getter methods, and that NullAway assumes all objects are always fully initialized. NullAway forces all generic arguments to be non-null, which is not an unsoundness but is less flexible than the Nullness Checker.

40.10.4 How does the Checker Framework compare with JSpecify?

JSpecify defines the two annotations `@NonNull` and `@Nullable`. This is one more possibility in addition to the other definitions of those annotations listed in Section 3.7. JSpecify is not an official standard.

You can use the JSpecify annotations or the Checker Framework ones, whichever you prefer. All nullness checkers we know of (e.g., Eclipse, EISOP, IntelliJ, NullAway, the Nullness Checker, SpotBugs) recognize the JSpecify and Nullness Checker versions of those two type annotations, so there is no tooling reason to prefer one over the other. JSpecify has only nullness annotations, and none for the dozens of other type-checkers that the Checker Framework supports.

The JSpecify specification language is weaker than what the Nullness Checker supports. For example, JSpecify lacks a polymorphic qualifier such as `@PolyNull`, and it lacks method specifications (Section 3.2.2). It lacks annotations for initialization (Section 3.8), but you could consider that a benefit, if you always suppress all initialization warnings,

opting for convenience and simplicity over sound checking. Even if your code does not use all the specification power of the Nullness Checker, you will benefit from expressive specifications on libraries that your code uses.

If you want precise, sound type-checking, you will need to use some Nullness Checker annotations. You can mix two sets of annotations (those of JSpecify and of the Nullness Checker), or you can use just those of the Nullness Checker. Both approaches work.

JSpecify contains a notion of "unspecified nullness". It is typically used for warning suppression in unannotated code, but doing so has two problems. First, it makes your specification tool-dependent, because a JSpecify-compliant tool may have *any* behavior (including issuing warnings or not issuing warnings) for code that uses "unspecified nullness" or uses a library that uses "unspecified nullness". Second, it mixes notions of nullness specification and warning suppression, which are separate concepts. The Checker Framework has well-defined behavior for all specifications, and it has rich facilities for warning suppression (Chapter 34). For example, the `-AskipUses` command-line option (Section 34.4) suppresses warnings relating to uses of unannotated libraries.

The Checker Framework does not need JSpecify's `@NullMarked`, which makes a checker treat unannotated references as `@NonNull`. The Nullness Checker by default treats unannotated references as `@NonNull`. The Checker Framework does not need JSpecify's `@NullUnmarked`, which sets the default to "unspecified nullness". The Nullness Checker supports explicit warning suppressions instead.

40.10.5 How does the Checker Framework compare with the EISOP Checker Framework?

The EISOP Checker Framework (also called the EISOP Framework) is a fork of the Checker Framework. The main difference is that the EISOP Nullness Checker supports one interpretation of the full JSpecify "standard" (see Section 40.10.4), whereas the Checker Framework's Nullness Checker only supports the non-ambiguous parts of JSpecify.

The EISOP Checker Framework is an "unfriendly fork" in that it incorporates improvements from the Checker Framework, but its developers do not contribute back to the Checker Framework: they only incorporate their improvements and bug fixes in their own version. The EISOP Checker Framework has introduced some bugs that do not exist in the Checker Framework, and it has fixed some bugs that exist in the Checker Framework, so neither one is strictly more correct than the other. If you use the EISOP Checker Framework, please encourage its developers to be good open-source citizens and contribute back to the Checker Framework. We wish the EISOP project well, because any project that gets more people to verify their code (by writing nullness specifications and running a checker) is a net positive for the programming community.

40.10.6 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. The Optional Checker (see Chapter 5) guarantees that programmers use `Optional` correctly.

Section 3.7.3 explains the relationship between nullness and `Optional` and the benefits of each.

40.10.7 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.

40.10.8 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 became part of Java with Java 8 (released in March 2014). For more about type annotations, see the Type Annotations (JSR 308) FAQ.

40.10.9 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, SpotBugs, IntelliJ, and other tools; see Section 3.7.

40.10.10 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 41

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 40). If you think you have found a bug in the Checker Framework, please report it (see Section 41.2). For discussions of broad interest, you can use the mailing list, `checker-framework-discuss@googlegroups.com`. For archives and to subscribe, see `https://groups.google.com/forum/#!forum/checker-framework-discuss`. If you want to help out, you can give feedback (including on the documentation), choose a bug and fix it (start with those labeled "good first issue"), or select a project from the ideas list at `https://checkerframework.org/manual/new-contributor-projects.html`.

41.1 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.

41.1.1 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.

41.1.2 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 41.2.) This section describes some possible problems and solutions.

- An error that includes `java.lang.NoSuchFieldError: RELEASE` means that you are not using the correct compiler (you are not using a Java 9+ compiler).
- 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 like one of the following,

```
...\build.xml:59: Error running ${env.CHECKERFRAMEWORK}\checker\bin\javac.bat compiler
```

```
.../bin/javac: Command not found
```

```
error: Annotation processor 'org.checkerframework.checker.signedness.SignednessChecker' not found
```

then the problem may be that you have not set the `CHECKERFRAMEWORK` environment variable, as described in Section 39.6. 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 39.6.

- If you get the error

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

then an annotation is not present at run time that was present at compile time. For example, maybe when you compiled the code, the `@Nullable` annotation was available, but it was not available at run time.

- If you get an error like

```
mylibrary.jar(MyClass.class):
  error: Cannot attach type annotations
    @org.checkerframework.checker.mustcall.qual.MustCall({}), ...
  to MyClass.myMethod:
    class file for com.example.SomeType not found
```

then the classpath is missing a class that is referenced by a library's signatures. In this example, `SomeType` appears in the signature of `MyClass.myMethod`, but `SomeType`'s class file is not on the classpath.
To fix the error, put the missing class on the classpath when compiling the code that uses the library. Either:

  - The library can declare its dependency on the missing class as part of its API, so that the dependency is passed on to the library's clients. For example, in Maven, use `compile` scope rather than `provided` scope or `<optional>true</optional>`; in Gradle, use an `api` dependency (which requires the `java-library` plugin) rather than an `implementation` or `compileOnly` dependency.
  - The code that uses the library can declare a dependency on the missing class itself. For example, in Maven, add a dependency with `provided` scope; in Gradle, add a `compileOnly` dependency (or a `testCompileOnly` dependency, if the error occurs when compiling tests).

  Here are more details about the error.
  Starting with JDK 22, when `javac` reads a class file, it attaches the type annotations in the class file to the types in the signatures of the class's methods and fields (see JDK-8225377). Doing so requires the class files for those types, even if your code never uses the method or field. If a class file is missing, `javac` issues this error. JDK 17 and JDK 21 also require the class files, if you pass `-XDaddTypeAnnotationsToSymbol=true`.
  This error is not specific to the Checker Framework: `javac` issues it whenever a library contains type annotations, such as ones written explicitly in the library's source code. However, the error is more common when a library is compiled with the Checker Framework, because the Checker Framework writes an annotation on every type in the library's signatures (see Section 40.9.3). This is the case even if the library's source code contains no annotations.
  If you use Maven with `<fork>true</fork>`, then Maven may hide this error message and report only "Compilation failure" (see MCOMPILER-434).

- The Checker Framework sometimes runs out of memory when processing very large Java files, or files with very large methods.

If you get an error such as

```
error: SourceChecker.typeProcess: unexpected Throwable (OutOfMemoryError) while processing ...
  ; message: GC overhead limit exceeded
```

(all on one line), then either give the JVM more memory when running the Checker Framework, or split your files and methods into smaller ones, or both.

- Versions of Error Prone before 2.4.0 are incompatible with the Checker Framework. (More precisely, each version of Error Prone before 2.4.0 is compatible with only one specific version of the Checker Framework.) Those versions of Error Prone use outdated versions of the Checker Framework's dataflow analysis library, each of which is compatible with only one version of the Checker Framework.
  If you wish to use both Error Prone and the Checker Framework, then use Error Prone version 2.4.0 or later. If you cannot upgrade to Error Prone 2.4.0 or later, there is a way to still use both tools. The Gradle plugin documentation (`https://github.com/typetools/checker-framework-gradle-plugin#incompatibility-with-error-prone-234-an`) shows how to do so if you use the Gradle build system.
- Versions 0.9.0 and 0.9.1 of NullAway are compatible only with Checker Framework version 3.6.0 through 3.8.0. Those versions of NullAway use version 3.6.0 of the Checker Framework's dataflow analysis library without shading it. As of version 0.9.2, NullAway uses a shaded version of the library, so this conflict no longer exists. If you want to use a version of the Checker Framework newer than 3.8.0 with NullAway in the same build, then use NullAway 0.9.2 or later.

41.1.3 Unexpected warnings not related to type-checking

This section gives solutions for some warning messages that are not related to type errors in your code.

- 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 one of the following:

  `@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`.

  `@Nullable Map.Entry` You must write `Outer.@Nullable StaticNestedClass` rather than `@Nullable Outer.StaticNestedClass`. Since a static nested class does not depend on its outer class, the annotation on the outer class would have no effect and is forbidden.

  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 `pakkage.Outer.Nested`.
- An "error: package ... does not exist" or "error: cannot find symbol" about classes in your own project or its dependencies means that you have left your own project or its dependencies off the classpath when you invoked the compiler.

41.1.4 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.
  - Perhaps your proof was incomplete or incorrect. Your proof might depend 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.
  - Perhaps your proof is incorrect, and the errors indicate where.
  - Perhaps 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 2.4.5, 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 33.7.5 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 36). The annotations might appear in stub files (`.astub` files distributed with the checker's source code) or in annotated libraries (which appear in `https://github.com/typetools/`).
- 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 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 call `addStandardLiteralQualifiers` on the `LiteralTreeAnnotator`.
- 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 Figures 3.2 and 3.3 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.
- 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] Expression in dependent type annotation invalid:
Use "#1" rather than "value"
```

Section 33.8 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) { ... }
```

- The Checker Framework can issue false positive warnings in dead code. An example is

```
void m() {
  String t = "";
  t.toString();
  try {
  } catch (Throwable e) {
      t.toString(); // Nullness Checker issues spurious (dereference.of.nullable)
  }
}
```

- If the Checker Framework issues an error or warning that you cannot reproduce using a smaller test case, then try running the Checker Framework with the `-AatfDoNotCache` command-line argument. If the Checker Framework behaves differently with and without that command-line argument, then there is a bug related to its internal caches. Please report that bug at `https://github.com/typetools/checker-framework/issues`.

41.1.5 Unexpected compilation output when running javac without a pluggable type-checker

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;
```

41.2 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 41.2.1. You can also use the issue tracker to make suggestions and feature requests. We also welcome pull requests with annotated libraries, bug fixes, new features, new checkers, 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 should include at least the following 4 parts: commands, inputs, output, and expectation.

Commands Provide one or more commands that can be pasted into a command shell to reproduce the problem.

In the simplest case, you will provide one command of the form `javac -processor ... MyFile.java`. In more complex cases, it might be a set of commands to clone a repository, check out a branch, and run a build command. Include commands to install software and set environment variables if necessary.

If you encountered the problem in an IDE, reproduce it from the command line; if you cannot, report the bug to the IDE integration. If you encountered the problem when using a build system, it is most useful to the Checker Framework developers if you provide a single `javac` command rather than a build file.

It can be helpful to add

`-version -Aversion -AprintGitProperties -verbose -AprintVerboseGenerics`

to the javac options. This causes the compiler to output debugging information, including its version number.

Inputs 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. This is not necessary if the commands you provided obtain the code that is type-checked.

If you cannot share your code, create a new, small test case. (A small test case is helpful even if your code is not secret!) If the Checker Framework crashed, the progress tracing options (Section 37.12.4) can be helpful in determining which file the Checker Framework was processing when it crashed.

Minimization: If your command invokes a build system such as Gradle or Maven, it can be helpful to the maintainers if you are able to reduce it to a single `javac` invocation on a single file. This also rules out the possibility that the problem is with the build system integration rather than the checker itself. If you use an invasive annotation processor such as Lombok, then try to reproduce the problem without it — this will indicate whether the problem is in the Checker Framework proper or in its interaction with Lombok.

Output 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, don't use a screenshot, and don't provide just part of the output.

Expectation If the problem is not a crash, then 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 reveal 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.

It can also be helpful to indicate what you have already done to try to understand the problem. Did you do any additional experiments? What parts of the manual did you read, and what else did you search for in the manual? Without this information, the maintainers may give you redundant suggestions or may waste time re-doing work you have already done.

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! That is not a good use of your time to report, nor of the maintainers' time to evaluate and fix.

41.2.1 Problems with annotated libraries

If a checker reports a warning because a library contains *incorrect* annotations, then please open a pull request that adds the annotations. If the library's maintainers have added annotations, make a pull request to them. If the Checker Framework maintainers have added annotations, you can find the annotated source in `https://github.com/typetools/`.

If a checker reports a warning because a library is *not annotated*, please do not open a GitHub issue. 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 36.5.

For either approach, please see Chapter 36, page 239 for tips about writing library annotations.

Thanks for your contribution to the annotated libraries!

41.3 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.

41.3.1 Install prerequisites

You need to install several packages in order to build the Checker Framework. Follow the instructions for your operating system. If your OS is not listed, adapt the instructions for an existing OS; for example, other Linux distributions will be similar to Ubuntu.

Put the setting of the `JAVA_HOME` environment variable in a startup file such as `.bash_profile` or `.bashrc`, if `JAVA_HOME` is not already set appropriately. It must be the location of your JDK installation (not the JRE installation). You may need to log out and log back in for the setting to take effect.

Ubuntu Run the following commands, which should be the same as those in `checker/bin-devel/Dockerfile-ubuntu-jdk`*XX*`-plus`, where *XX* is a JDK version such as 17, 21, 25, etc.:
(If your build fails because it is missing a package, try the commands there and report a bug in this manual.)

```
sudo apt -qqy update
# Dependencies for building and running the Checker Framework.
sudo apt -qqy install \
  ant \
  cpp \
  git \
  jq \
  jtreg8 \
  libcurl3-gnutls \
  make \
  maven \
  python3-requests \
  python3-setuptools \
  unzip \
  wget
# Dependencies for developing the Checker Framework.
sudo apt -qqy install \
  autoconf \
  devscripts \
  dia \
  graphviz \
  imagemagick \
  junit \
  latexmk \
  librsvg2-bin \
  libasound2-dev libcups2-dev libfontconfig1-dev \
  libx11-dev libxext-dev libxrender-dev libxrandr-dev libxtst-dev libxt-dev \
  pdf2svg \
  poppler-utils \
  rsync \
  shellcheck \
  shfmt \
  texlive-extra-utils \
  texlive-font-utils \
  texlive-fonts-recommended \
  texlive-latex-base \
  texlive-latex-extra \
  texlive-latex-recommended

# Install uv (manages Python dependencies).
curl -LsSf https://astral.sh/uv/install.sh | sh
export PATH=$HOME/.local/bin:$PATH

# Install prek (pre-commit hook).
uv tool install prek
```

```
# Install the JDK.
sudo apt -qqy install \
  openjdk-25-jdk \
&& update-java-alternatives --set java-1.25.0-openjdk-amd64
export JAVA25_HOME=/usr/lib/jvm/java-25-openjdk-amd64
```

You may have to answer questions about which time zone your computer is in.
In a startup file, write:

```
export JAVA_HOME=${JAVA_HOME:-$(dirname "$(dirname "$(readlink -f "$(which javac)")")")}
```

macOS If you employ homebrew to install packages, run the following commands:

```
brew update
brew install git graphviz ant maven librsvg poppler unzip make
brew install --cask temurin17
brew install --cask mactex
```

Note: Running `brew install -cask temurin17` is only necessary if you do not already have a JDK that is compatible with the Checker Framework installed.
If `echo $JAVA_HOME` outputs nothing (that is, if the `JAVA_HOME` environment variable is not set), then write the following in a startup file:

```
export JAVA_HOME=$(/usr/libexec/java_home -v XX)
```

or

```
export JAVA_HOME=/Library/Java/JavaVirtualMachines/adoptopenjdk-XX.jdk/Contents/Home/
```

In the command, *XX* is a JDK version such as 17, 21, 25, etc.

Windows To build on Windows 10 or later, run `bash` to obtain a version of Ubuntu (provided by the Windows Subsystem for Linux) and follow the Ubuntu instructions.

41.3.2 Obtain the source

Obtain the latest source code from the version control repository:

```
git clone https://github.com/typetools/checker-framework.git checker-framework
export CHECKERFRAMEWORK=$(pwd)/checker-framework
```

You might want to add an `export CHECKERFRAMEWORK=...` line to your `.bashrc` file.

41.3.3 Build the Checker Framework

1. As with any Gradle project, run `./gradlew assemble` to build the Checker Framework:

   ```
   cd $CHECKERFRAMEWORK
   ./gradlew assemble
   ```

   Your antivirus program (e.g., Windows Security Virus & threat protection) might make the build run very slowly, or might make it seem to stall. Just give it time.
   If compilation fails, but you have not made any changes, then perhaps one of the supporting repositories is in an inconsistent state; see Section 41.3.7.
2. Once it is built, you may wish to put the Checker Framework's `javac` even earlier in your `PATH`:

   ```
   export PATH=$CHECKERFRAMEWORK/checker/bin:${PATH}
   ```

   The Checker Framework's `javac` ensures that all required libraries are on your classpath and boot classpath, but is otherwise identical to the javac Java compiler.
   Putting the Checker Framework's `javac` earlier in your `PATH` will ensure that the Checker Framework's version is used.

3. If you wish to use Maven or Gradle, publish the Checker Framework artifacts to a local Maven repository by running `./gradlew publishToMavenLocal`:

   ```
   cd $CHECKERFRAMEWORK
   ./gradlew publishToMavenLocal
   ```

   Then use the Maven or Gradle instructions, but modify the version number for the Checker Framework artifacts. Use the version number that is output when you run `./gradlew version`.
4. Test that everything works:
   - Run `./gradlew allTests`:

     ```
     cd $CHECKERFRAMEWORK
     ./gradlew allTests
     ```
   - Run the Nullness Checker examples (see Section 3.5, page 38).

41.3.4 Build the Checker Framework Manual (this document)

1. Install needed packages; see Section 41.3.1 for instructions.
2. Run `make` in the `docs/manual` directory to build both the PDF and HTML versions of the manual.

41.3.5 Code style, IDE configuration, pull requests, etc.

Please see the Checker Framework Developer Manual.

41.3.6 Enable continuous integration builds

We strongly recommend that you enable continuous integration (CI) builds on your fork of the projects, so that you learn quickly about errors. See the CI instructions in the Checker Framework Developer Manual.

Debugging continuous integration builds

If a continuous integration job is failing, you can reproduce the problem locally. The CI 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/cf-ubuntu-jdk21
docker run -it mdernst/cf-ubuntu-jdk21 /bin/bash
```

Then, you can run arbitrary commands, including those that appear in the CI configuration files. For example:

```
git clone --filter=blob:none -b docker-ubuntu-24-10 https://github.com/mernst/checker-framework.git
cd checker-framework
export CHECKERFRAMEWORK=$(pwd)
alias javacheck='$CHECKERFRAMEWORK/checker/bin/javac'
export NO_WRITE_VERIFICATION_METADATA=true
checker/bin-devel/test-cftests-junit.sh
```

41.3.7 Related repositories

Building the Checker Framework depends on other repositories, such as Annotated JDK. You need to have compatible versions of each.

After you `git pull` in one of the repositories, you should do so in the others as well. The script `checker/bin-devel/clone-related` pulls all the related repositories, so to build the latest development version of the Checker Framework you can do

```
cd $CHECKERFRAMEWORK
git pull
checker/bin-devel/clone-related.sh
```

41.4 Contributing

We welcome contributions and pull requests. Section 41.3 tells you how to set up your development environment to compile the Checker Framework, and the Checker Framework Developer Manual gives more information.

One good way to contribute is to give feedback about your use of the tool. There is a list of potential projects for new contributors at `https://checkerframework.org/manual/new-contributor-projects.html`. Or, you can fix a bug that you have encountered during your use of the Checker Framework.

41.4.1 Contributing fixes (creating a pull request)

Please see the pull requests section of the Checker Framework Developer Manual. Thanks in advance for your contributions!

The easiest bugs to fix for a newcomer are labeled as "good first issue".

Please do not spam the issue tracker with requests to assign an issue to you. If you want to contribute by fixing an issue, just open a pull request that fixes it. You can do that even if the issue is already assigned to someone.

Please do not spam the issue tracker asking how to get started fixing a bug. If you have concrete questions, please ask them and we will be happy to assist you. If you don't know how to get started fixing a particular bug, then you should contribute to the project in other ways. Thanks!

41.5 Credits and changelog

Differences from previous versions of the checkers and framework can be found in the `docs/CHANGELOG.md` file. This file is included in the Checker Framework distribution and is also available on the web at `https://checkerframework.org/CHANGELOG.md`.

Developers who have contributed code to the Checker Framework include Abhijit Kulkarni, Abraham Lin, Adian Qian, Aditya Singh, Akash Srivastava, Alex Liu, Alexander Schwartz, Alvin Abdagic, Anant Jain, Anatoly Kupriyanov, Andy Turner, Ankit Yadav, Aosen Xiong, Arie van Deursen, Artem Pyanykh, Arthur Baars, Ashish Rana, Asumu Takikawa, Atul Dada, Ayush Agarwal, Baorui Zhou, Basil Peace, Benno Stein, Bohdan Sharipov, Brian Corcoran, Calvin Loncaric, Charles Chen, Charlie Garrett, Chris Povirk, Chris Toxiadis, Christopher Mackie, Colin S. Gordon, Craig Day, Damika Anupama, Dan Brotherston, Dan Brown, Daniel Gao, Daniel Zhu, David Lazar, David McArthur, Di Wang, Dilraj Singh, Dmitriy Shepelev, Eric Spishak, Ethan Koenig, Farzan Mirshekari, Felipe R. Monteiro, Florian Lanzinger, Gagan Dhakrey, Gautam Korlam, Google Inc. (via @wmdietlGC), Haaris Ahmed, Haifeng Shi, Hamed Taghani, Heath Borders, Ivory Wang, Jakub Vrána, James Yoo, Jason Waataja, Javier Thaine, Jeff Luo, Jenny Xiang, Jeroen Meijer, Jianchu Li, Jiangqi Zhang, Jiasen (Jason) Xu, Joe Schafer, John Krah, John Vandenberg, JohnnyJayJay, Jonathan Burke, Jonathan Nieder, Joshua Peterson, Jugal Mistry, Junhao Hu, Kanak Das, Kartikeya Goswami, Kenneth Knowles, Kivanc Muslu, Konstantin Weitz, Kurt Alfred Kluever, Lazaro Clapp, Leo Liu, Liam Miller-Cushon, Lian Sun, Luqman Aden, Mahmood Ali, Manu Sridharan, Mark Roberts, Marko Bekhta, Markus Frohme, Martin Kellogg, Matt Mullen, Maximilian Gama, Michael Bayne, Michael Coblenz, Michael Ernst, Michael Hixson, Michael Sloan, Michal Stehlik, Mier Ta, Mrigank Arora, Muyeed Ahmed, Narges Shadab, Neil Brown, Nhat Dinh, Nhat Nguyen, Nicholas Breen, Nikhil Shinde, Nima Karimipour, Nitin Kumar Das, Oleg Shchelykalnov, Olek Wojnar, Pascal Wittmann, Patrick Meiring, Paul Vines, Paulo Barros, Philip Lai, Piyush Jha, Pratik Bhusal, Prionti Nasir, Priti Chattopadhyay, Rashmi Mudduluru, Ravi Roshan, Renato Athaydes, René Just, René Kraneis, Rob Bygrave, Rui Fernandes, Ruturaj Mohanty, Ryan Oblak, Sadaf Tajik, Sagar Tewari, Sanjay Malakar, Sascha Kehrli, Sean C. Sullivan, Sean McLaughlin, Sebastian Schuberth, Shinya Yoshida, Shubham Kapoor, Shubham Raj, Sidney Monteiro, Simon Gerst, Stefan Heule, Steph Dietzel, Stephan Schroevers, Stuart Pernsteiner, Suzanne Millstein, Thomas Schweizer, Thomas Weißschuh, Tony Wang, Trask Stalnaker, Travis Haagen, Utsav Oza, Vatsal Sura, Vitalii Ananiev, Vivek Singh Solanki, Vladimir Sitnikov, Vlastimil Dort, Weitian Xing, Werner Dietl, Yuki Yang, Zhiping Cai.

In addition, too many users to list have provided valuable feedback, which has improved the toolset's design and implementation. Thanks for your help!

41.6 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. This 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 may need to be licensed under the GPL. Because of the classpath exception, you can include parts of the Checker Framework in your product, even if your product uses a different license. For example, Google's Error Prone and Uber's NullAway both use the Checker Framework internally, but neither one is 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`.

41.7 Publications

Here are some papers that use the Checker Framework; for instance, they build a type-checker on top of the Checker Framework, or their methodology depends on using the Checker Framework. (The list does not include papers that merely cite the Checker Framework.) This list is *incomplete*; if you know of a paper that is missing, please let us know so that we can add it. Most educational use of the Checker Framework is never published, and most commercial use of the Checker Framework is never discussed publicly. Most of the papers are available for free online; do a web search on the title.

- Practical pluggable types for Java (ISSTA 2008) [PAC+08]
- Building and using pluggable type systems with the Checker Framework (ECOOP 2008) [Ern08a]
- Type-Based Object Immutability with Flexible Initialization (ECOOP 2009) [HP09]
- The nullness analyser of Julia (LPAR 2010) [Spo10]
- Static checking of safety critical Java annotations (JTRES 2010) [TPV10]
- Ownership and immutability in generic Java (OOPSLA 2010) [ZPL+10]
- Rethinking the economics of software engineering (FOSER 2010) [SE10]
- Applying uniqueness to the Java language (Master's Thesis, 2010) [Har10]
- TIFI+: A Type Checker for Object Immutability with Flexible Initialization (Diploma thesis, 2010) [Noa10]
- Building and using pluggable type-checkers (ICSE 2011) [DDE+11]
- Inference of field initialization (ICSE 2011) [SE11]
- EnerJ: approximate data types for safe and general low-power computation (PLDI 2011) [SDF+11]
- Tunable static inference for generic universe types (ECOOP 2011) [DEM11]
- Separating ownership topology and encapsulation with generic universe types (TOPLAS 2011) [DDM11]
- The design, implementation and evaluation of a pluggable type checker for thread-locality in Java (Master's Thesis, 2011) [She11]
- Towards Effective Inference and Checking of Ownership Types (IWACO 2011) [HM11]
- Static Dominance Inference (TOOLS 2011) [MV11]
- A Static Memory Safety Annotation System for Safety Critical Java (RTSS 2011) [TPNV11]
- Inference and checking of object ownership (ECOOP 2012) [HDME12]
- A type system for regular expressions (FTfJP 2012) [SDE12]
- Verification games: making verification fun (FTfJP 2012) [DDE+12]
- Reim & ReImInfer: checking and inference of reference immutability and method purity (OOPSLA 2012) [HMDE12]
- ReImInfer: method purity inference for Java (FSE 2012) [HM12]
- Inference and checking of context-sensitive pluggable types (FSE 2012) [MH12]

- Documenting Java database access with type annotations (WorldComp 2012) [Ber12]
- Immutability (book chapter, 2013) [PÖZE13]
- JavaUI: Effects for controlling UI object access (ECOOP 2013) [GDEG13]
- Reified type parameters using Java annotations (GPCE 2013) [GBS13]
- Evaluating the Accuracy of Annotations in the Loci 3.0 Pluggable Type Checker (Master's Thesis, 2013) [Zaz13]
- Composing polymorphic information flow systems with reference immutability (FTfJP 2013) [MH13]
- Type-based taint analysis for Java web applications (FASE 2014) [HDM14]
- A type system for format strings (ISSTA 2014) [WKSE14]
- Collaborative verification of information flow for a high-assurance app store (CCS 2014) [EJM+14]
- Cascade: A universal type qualifier inference tool (ICSE 2015) [VPEJ15]
- An inference and checking framework for context-sensitive pluggable types (PhD thesis, 2014) [Hua14]
- CFL-reachability and context-sensitive integrity types (PPPJ 2014) [MHD14]
- Static analysis of implicit control flow: resolving Java reflection and Android intents (ASE 2015) [BJM+15]
- Scalable and Precise Taint Analysis for Android (ISSTA 2015) [HDMD15]
- A context-sensitive security type system for Java (Master's thesis, 2015) [Kai15]
- Locking discipline inference and checking (ICSE 2016) [ELM+16]
- Semantics for locking specifications (NFM 2016) [EMMS16]
- JCrypt: Towards computation over encrypted data (PPPJ 2016) [DMD16a]
- Preventing signedness errors in numerical computations in Java (FSE 2016) [Mac16]
- Enforcing correct array indexes with a type system (FSE 2016) [San16]
- Gradual Pluggable Typing in Java (Master's Thesis, 2016) [Bro16]
- Security analysis of the IRMA app using SPARTA and fuzzing (Bachelor Thesis, 2016) [Bri16]
- Static Analysis and Program Transformation for Secure Computation on the Cloud (ISSTA 2016 Doctoral Symposium) [DMD16b]
- Towards using concurrent Java API correctly (ICECCS 2016) [LBSD16]
- Reflection-aware static analysis of Android apps (ASE 2016) [LBOK16]
- A crowdsourcing game for formal software verification (Bachelor's thesis) [Iri16]
- Exploring Language Support for Immutability (ICSE 2016) [CSA+16]
- Granullar: Gradual nullable types for Java (CC 2017) [BDL17]
- Type checking for reliable APIs (WAPI 2017) [KS17]
- Glacier: transitive class immutability for Java (ICSE 2017) [CNA+17]
- A General Pluggable Type Inference Framework and its use for Data-flow Analysis (Master's Thesis, 2017) [Li17]
- Null safety benchmarks for object initialization (ISP RAS, 2017) [Kog17]
- Static analysis and program transformation for secure computation on the cloud (PhD Thesis, 2017) [Don17]
- Spartan Jester: end-to-end information flow control for hybrid Android applications (SPW 2017) [SCN17]
- Lightweight verification of array indexing (ISSTA 2018) [KDME18]
- Safe Stream-based Programming with Refinement Types (ASE 2018) [SCSC18]
- Don't miss the end: Preventing unsafe end-of-file comparisons (NFM 2018) [CD18]
- Context sensitive typechecking and inference: ownership and immutability (Master's Thesis, 2018) [Ta18]
- Usable and Sound Static Analysis through its Integration into Automated and Interactive Workflows (Master's Thesis, 2018) [Val18]
- Pluggable properties for program understanding: Ontic type checking and inference (Master's Thesis, 2018) [Che18]
- Definite reference mutability (ECOOP 2018) [Mil18]
- Googletest to CUTE converter (Technical report, 2018) [GV18]
- NullAway: Practical type-based null safety for Java (ESEC/FSE 2019) [BCS19]
- Compile-time detection of machine image sniping (ASE 2019) [Kel19]
- Type-Directed Bounding of Collections in Reactive Programs (VMCAI 2019) [LČCT19]
- Scala with Explicit Nulls (Master's Thesis, 2019) [NR19]
- A machine-checked proof of security for AWS Key Management Service (CCS, 2019) [ABB+19]
- Synthesis of assurance cases for software certification (ICSE NIER 2020) [BKM20]
- Verifying object construction (ICSE 2020) [KRS+20]

- Precise inference of expressive units of measurement types (OOPSLA 2020) [XLD20]
- Rethinking safe consistency in distributed object-oriented programming (OOPSLA 2020) [KEW+20]
- Continuous compliance (ASE 2020) [KSTE20]
- Light-weight verification of cryptographic API usage (Master's Thesis, 2020) [Xin20]
- Type checking and whole-program inference for value range analysis (Master's Thesis, 2020) [Xia20]
- User-Centered Design of Principled Programming Languages (PhD thesis, 2020) [Cob20]
- Behavioural types for memory and method safety in a core object-oriented language (APLAS 2020) [BFG+20]
- Scala with Explicit Nulls (ECOOP 2020) [NZL+20]
- Property Types in Java: Combining Type Systems and Deductive Verification (Master's Thesis, 2021) [Lan21]
- Interval Type Inference: Improvements and Evaluations (Master's Thesis, 2021) [Wan21]
- Coping with the reality: adding crucial features to a typestate-oriented language (Master's Thesis, 2021) [dLM21]
- Java Typestate Checker (Coordination 2021) [MGR21]
- Ensuring correct cryptographic algorithm and provider usage at compile time (FTfJP 2021) [XCD21]
- Scalability and Precision by Combining Expressive Type Systems and Deductive Verification (OOPSLA 2021) [LWUD21]
- Automatic Annotation of Confidential Data in Java Code (FPS 2021) [BBRS21]
- Accumulation Analysis (ECOOP 2022) [KSSE22]
- Lightweight Verification via Specialized Typecheckers (PhD thesis, 2022) [Kel22]

Bibliography

[ABB+19] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Matthew Campagna, Ernie Cohen, Benjamin Gregoire, Vitor Pereira, Bernardo Portela, Pierre-Yves Strub, and Serdar Tasiran. A machine-checked proof of security for AWS Key Management Service. In *CCS 2019: Proceedings of the 21st ACM Conference on Computer and Communications Security*, page 63–78, London, UK, November 2019.

[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.

[BBRS21] Iulia Bastys, Pauline Bolignano, Franco Raimondi, and Daniel Schoepe. Automatic annotation of confidential data in Java code. In *FPS 2021: 14th International Symposium on Foundations & Practice of Security*, Paris, France, December 2021.

[BCS19] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. NullAway: Practical type-based null safety for Java. In *ESEC/FSE 2019: The ACM 27th joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 740–750, Tallinn, Estonia, August 2019.

[BDL17] Dan Brotherston, Werner Dietl, and Ondřej Lhoták. Granullar: Gradual nullable types for Java. In *CC 2017: 26th International Conference on Compiler Construction*, pages 87–97, Austin, TX, USA, February 2017.

[Ber12] Paul L. Bergstein. Documenting Java database access with type annotations. In *WorldComp 2012: Proceedings of The 2012 World Congress in Computer Science, Computer Engineering, and Applied Computing*, 2012.

[BFG+20] Mario Bravetti, Adrian Francalanza, Iaroslav Golovanov, Hans Hüttel, Mathias S. Jakobsen, Mikkel K. Kettunen, and António Ravara. Behavioural types for memory and method safety in a core object-oriented language. In *APLAS 2020: 18th Asian Symposium on Programming Languages and Systems*, page 105–124, Fukuoka, Japan, November 2020.

[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.

[BKM20] Hamid Bagheri, Eunsuk Kang, and Niloofar Mansoor. Synthesis of assurance cases for software certification. In *ICSE NIER, Proceedings of the 42nd International Conference on Software Engineering, New Ideas and Emerging Results Track*, pages 61–64, Seoul, Korea, May 2020.

[Bri16] Laurens Brinker. Security analysis of the IRMA app using SPARTA and fuzzing. Bachelor thesis, Radboud University, Nijmegen, Netherlands, 2016.

[Bro16] Daniel Brotherston. Gradual pluggable typing in Java. Master's thesis, U. of Waterloo, Waterloo, Ontario, Canada, 2016.

[CD18] Charles Zhuo Chen and Werner Dietl. Don't miss the end: Preventing unsafe end-of-file comparisons. In *NFM 2018: 10th NASA Formal Methods Symposium*, pages 87–94, Newport News, VA, USA, April 2018.

[Che18] Zhuo Chen. Pluggable properties for program understanding: Ontic type checking and inference. Master's thesis, U. of Waterloo, Waterloo, Ontario, Canada, 2018.

[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.

[Cob20] Michael Coblenz. *User-Centered Design of Principled Programming Languages*. PhD thesis, Carnegie Mellon University, aug 2020. TR CMU-CS-20-127.

[Cop05] Tom Copeland. *PMD Applied*. Centennial Books, November 2005.

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

[CSA+16] Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, Brad Myers, Sam Weber, and Forrest Shull. Exploring language support for immutability. In *ICSE 2016, Proceedings of the 38th International Conference on Software Engineering*, pages 736–747, Austin, TX, USA, May 2016.

[Dar06] Joe Darcy. JSR 269: Pluggable annotation processing API. `https://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.

[DDE+12] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Nathaniel Mote, Brian Walker, Seth Cooper, Timothy Pavlik, and Zoran Popović. Verification games: Making verification fun. In *FTfJP: 14th Workshop on Formal Techniques for Java-like Programs*, pages 42–49, Beijing, China, June 2012.

[DDM11] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Separating ownership topology and encapsulation with Generic Universe Types. *ACM Transactions on Programming Languages and Systems*, 33(6):20:1–62, December 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.

[dLM21] João Daniel da Luz Mota. Coping with the reality: adding crucial features to a typestate-oriented language. Master's thesis, Universidade Nova de Lisboa, Faculdade de Ciências e Tocnologia, February 2021.

[DMD16a] Yao Dong, Ana Milanova, and Julian Dolby. JCrypt: Towards computation over encrypted data. In *PPPJ 2016: Proceedings of the 13th International Conference on Principles and Practice of Programming in Java: Virtual Machines, Languages, and Tools*, Lugano, Switzerland, July 2016.

[DMD16b] Yao Dong, Ana Milanova, and Julian Dolby. Static analysis and program transformation for secure computation on the cloud. In *ISSTA 2016 Doctoral Symposium*, Saarbrücken, Genmany, July 2016.

[Don17] Yao Dong. *Static analysis and program transformation for secure computation on the cloud*. PhD thesis, Rensselaer Polytechnic Institute, December 2017.

[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+16] Michael D. Ernst, Alberto Lovato, Damiano Macedonio, Fausto Spoto, and Javier Thaine. Locking discipline inference and checking. In *ICSE 2016, Proceedings of the 38th International Conference on Software Engineering*, pages 1133–1144, Austin, TX, USA, May 2016.

[EMMS16] Michael D. Ernst, Damiano Macedonio, Massimo Merro, and Fausto Spoto. Semantics for locking specifications. In *NFM 2016: 8th NASA Formal Methods Symposium*, pages 355–372, Minneapolis, MN, USA, June 2016.

[Ern08a] Michael D. Ernst. Building and using pluggable type systems with the Checker Framework. In *ECOOP 2008 — Object-Oriented Programming, 22nd European Conference*, Paphos, Cyprus, July 2008. Tool demo.

[Ern08b] Michael D. Ernst. Type Annotations specification (JSR 308). `https://checkerframework.org/jsr308/`, September 12, 2008.

[GBS13] Prodromos Gerakios, Aggelos Biboudis, and Yannis Smaragdakis. Reified type parameters using java annotations. In *GPCE 2013: Proceedings of the 12th International Conference on Generative Programming and Component Engineering*, pages 61–64, Indianapolis, IN, USA, October 2013.

[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://web.archive.org/web/20171025205847/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.

[GV18] Sascha Gschwind and Renato Venzin. Googletest to CUTE converter. Technical report, University of Applied Sciences Rapperswil, Rapperswil, Switzerland, 2018.

[Har10] Artemus Harper. Applying uniqueness to the Java language. Master's thesis, Washington State University, 2010.

[HDM14] Wei Huang, Yao Dong, and Ana Milanova. Type-based taint analysis for Java web applications. In *FASE 2014: Fundamental Approaches to Software Engineering*, pages 140–154, Grenoble, France, April 2014.

[HDMD15] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. Scalable and precise taint analysis for Android. In *ISSTA 2015, Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 106–117, Baltimore, MD, USA, July 2015.

[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.

[HM11] Wei Huang and Ana Milanova. Towards effective inference and checking of ownership types. In *IWACO 2011: International Workshop on Aliasing, Confinement and Ownership in object-oriented programming*, Lancaster, UK, July 2011.

[HM12] Wei Huang and Ana Milanova. ReImInfer: Method purity inference for Java. In *FSE 2012: Proceedings of the ACM SIGSOFT 20th Symposium on the Foundations of Software Engineering*, pages 1–4, Cary, NC, USA, November 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.

[HP09] Christian Haack and Erik Poll. Type-based object immutability with flexible initialization. In *ECOOP 2009 — Object-Oriented Programming, 23rd European Conference*, pages 520–545, Genova, Italy, July 2009.

[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.

[Hua14] Wei Huang. *An inference and checking framework for context-sensitive pluggable types*. PhD thesis, Rennselaer Polytechnic Institute, 2014.

[Iri16] Karamitrou Irini. A crowdsourcing game for formal software verification. Bachelor's thesis, University of Thessaly Polytechnic School Department of Electrical and Computer Engineering, Thessaly, Greece, 2016.

[Kai15] Benjamin Kaiser. A context-sensitive security type system for Java. Master's thesis, Rennselaer Polytechnic Institute, Troy, NY, USA, April 2015.

[KDME18] Martin Kellogg, Vlastimil Dort, Suzanne Millstein, and Michael D. Ernst. Lightweight verification of array indexing. In *ISSTA 2018, Proceedings of the 2018 International Symposium on Software Testing and Analysis*, pages 3–14, Amsterdam, Netherlands, July 2018.

[Kel19] Martin Kellogg. Compile-time detection of machine image sniping. In *ASE 2019: Proceedings of the 34th Annual International Conference on Automated Software Engineering*, pages 1256–1258, San Diego, CA, USA, September 2019.

[Kel22] Martin Kellogg. *Lightweight Verification via Specialized Typecheckers*. PhD thesis, University of Washington Paul G. Allen School of Computer Science and Engineering, Seattle, WA, USA, June 2022.

[KEW+20] Mirko Köhler, Nafise Eskandani, Pascal Weisenburger, Alessandro Margara, and Guido Salvaneschi. Rethinking safe consistency in distributed object-oriented programming. In *OOPSLA 2020, Object-Oriented Programming Systems, Languages, and Applications*, Chicago, IL, USA, November 2020.

[Kog17] Alexander V. Kogtenkov. Null safety benchmarks for object initialization. *Proceedings of the Institute for System Programming of RAS*, 29(6):135–150, 2017.

[KRS+20] Martin Kellogg, Manli Ran, Manu Sridharan, Martin Schäf, and Michael D. Ernst. Verifying object construction. In *ICSE 2020, Proceedings of the 42nd International Conference on Software Engineering*, pages 1447–1458, Seoul, Korea, May 2020.

[KS17] Maria Kechagia and Diomidis Spinellis. Type checking for reliable APIs. In *WAPI '17: Proceedings of the 1st International Workshop on API Usage and Evolution*, pages 15–18, Buenos Aires, Argentina, May 2017.

[KSSE21] Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. Lightweight and modular resource leak verification. In *ESEC/FSE 2021: The ACM 29th joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 181–192, Athens, Greece, August 2021.

[KSSE22] Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. Accumulation analysis. In *ECOOP 2022 — Object-Oriented Programming, 33rd European Conference*, pages 10:1–10:31, Berlin, Germany, June 2022.

[KSTE20] Martin Kellogg, Martin Schäf, Serdar Tasiran, and Michael D. Ernst. Continuous compliance. In *ASE 2020: Proceedings of the 35th Annual International Conference on Automated Software Engineering*, pages 511–523, Melbourne, Australia, September 2020.

[Lan21] Florian Lanzinger. Property types in Java: Combining type systems and deductive verification. Master's thesis, Karlsruher Institut für Technologie, feb 2021.

[LBOK16] Li Li, Tegawendé F. Bissyandé, Damien Octeau, and Jacques Klein. Reflection-aware static analysis of Android apps. In *ASE 2016: Proceedings of the 31st Annual International Conference on Automated Software Engineering*, pages 756–761, Singapore, Singapore, September 2016.

[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.

[LBSD16] Shuang Liu, Guangdong Bai, Jun Sun, and J. Dong. Towards using concurrent Java API correctly. In *ICECCS 2016: 21st International Conference on Engineering of Complex Computer Systems*, pages 219–222, Dubai, UAE, June 2016.

[LČCT19] Tianhan Lu, Pavol Černý, Bor-Yuh Evan Chang, and Ashutosh Trivedi. Type-directed bounding of collections in reactive programs. In *VMCAI 2019: 20th International Conference on Verification, Model Checking and Abstract Interpretation*, pages 275–296, Cascais, Portugal, January 2019.

[Li17] Jianchu Li. A general pluggable type inference framework and its use for data-flow analysis. Master's thesis, U. of Waterloo, Waterloo, Ontario, Canada, 2017.

[LWUD21] Florian Lanzinger, Alexander Weigl, Mattias Ulbrich, and Werner Dietl. Scalability and precision by combining expressive type systems and deductive verification. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA), October 2021.

[Mac16] Christopher A. Mackie. Preventing signedness errors in numerical computations in Java. In *FSE 2016: Proceedings of the ACM SIGSOFT 24th Symposium on the Foundations of Software Engineering*, pages 1148–1150, Seattle, WA, USA, November 2016.

[MGR21] João Mota, Marco Giunti, and António Ravara. Java typestate checker. In *Coordination 2021: 23rd IFIP WG 6.1 International Conference on Coordination Models and Languages*, page 121–133, Valletta, Malta, June 2021.

[MH12] Ana Milanova and Wei Huang. Inference and checking of context-sensitive pluggable types. In *FSE 2012: Proceedings of the ACM SIGSOFT 20th Symposium on the Foundations of Software Engineering*, pages 1–4, Cary, NC, USA, November 2012.

[MH13] Ana Milanova and Wei Huang. Composing polymorphic information flow systems with reference immutability. In *FTfJP: 14th Workshop on Formal Techniques for Java-like Programs*, pages 5:1–5:7, Montpellier, France, July 2013.

[MHD14] Ana Milanova, Wei Huang, and Yao Dong. CFL-reachability and context-sensitive integrity types. In *PPPJ 2014: Proceedings of the 2014 International Conference on Principles and Practice of Programming in Java: Virtual Machines, Languages, and Tools*, page 99–109, Cracow, Poland, September 2014.

[Mil18] Ana Milanova. Definite reference mutability. In *ECOOP 2018 — Object-Oriented Programming, 32nd European Conference*, pages 25:1–25:30, Amsterdam, Netherlands, July 2018.

[MV11] Ana Milanova and Jan Vitek. Static dominance inference. In *TOOLS 2011: Objects, Models, Components, Patterns, 49th International Conference*, pages 211–227, Zurich, Switzerland, June 2011.

[MWME21] Rashmi Mudduluru, Jason Waataja, Suzanne Millstein, and Michael D. Ernst. Verifying determinism in sequential programs. In *ICSE 2021, Proceedings of the 43rd International Conference on Software Engineering*, pages 37–49, Madrid, Spain, May 2021.

[Noa10] Gunther Noack. TIFI+: A type checker for object immutability with flexible initialization. Diploma thesis, University of Kaiserslautern, March 2010.

[NR19] Abel Nieto Rodriguez. Scala with explicit nulls. Master's thesis, University of Waterloo Department of Computer Science, Waterloo, Ontario, Canada, December 2019.

[NZL+20] Abel Nieto, Yaoyu Zhao, Ondřej Lhoták, Angela Chang, and Justin Pu. Scala with explicit nulls. In *ECOOP 2020 — Object-Oriented Programming, 33rd European Conference*, pages 25:1–25:26, online, November 2020.

[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.

[PÖZE13] Alex Potanin, Johan Östlund, Yoav Zibin, and Michael D. Ernst. Immutability. In *Aliasing in Object-Oriented Programming*, volume 7850 of *LNCS*, pages 233–269. Springer-Verlag, April 2013.

[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.

[San16] Joseph Santino. Enforcing correct array indexes with a type system. In *FSE 2016: Proceedings of the ACM SIGSOFT 24th Symposium on the Foundations of Software Engineering*, pages 1142–1144, Seattle, WA, USA, November 2016.

[SCN17] Julian Sexton, Andrey Chudnov, and David A. Naumann. Spartan Jester: end-to-end information flow control for hybrid Android applications. In *SPW 2017: Symposium Security and Privacy Workshops*, pages 157–162, San Jose, CA, USA, May 2017.

[SCSC18] Benno Stein, Lazaro Clapp, Manu Sridharan, and Bor-Yuh Evan Chang. Safe stream-based programming with refinement types. In *ASE 2018: Proceedings of the 33rd Annual International Conference on Automated Software Engineering*, pages 565–576, Montpellier, France, September 2018.

[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.

[SE10] Todd W. Schiller and Michael D. Ernst. Rethinking the economics of software engineering. In *FoSER: Workshop on the Future of Software Engineering Research*, pages 325–330, Santa Fe, NM, USA, November 2010.

[SE11] Fausto Spoto and Michael D. Ernst. Inference of field initialization. In *ICSE 2011, Proceedings of the 33rd International Conference on Software Engineering*, pages 231–240, Waikiki, Hawaii, USA, May 2011.

[SGT+23] Narges Shadab, Pritam Gharat, Shrey Tiwari, Michael D. Ernst, Martin Kellogg, Shuvendu Lahiri, Akash Lal, and Manu Sridharan. Inference of resource management specifications. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2, article #282):1705–1728, October 2023.

[She11] Amanj Sherwany. The design, implementation and evaluation of a pluggable type checker for thread-locality in Java. Master's thesis, Uppsala University, Department of Information Technology, Uppsala, Sweden, 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.

[Spo10] F. Spoto. The nullness analyser of Julia. In *LPAR 2010: Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 405–424, Dakar, Senegal, April 2010.

[Sun21] Lian Sun. An immutability type system for classes and objects: Improvements, experiments, and comparisons. Master's thesis, University of Waterloo Department of Electrical and Computer Engineering, Waterloo, Ontario, Canada, April 2021.

[Ta18] Mier Ta. Context sensitive typechecking and inference: Ownership and immutability. Master's thesis, University of Waterloo Department of Electrical and Computer Engineering, Waterloo, Ontario, Canada, 2018.

[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.

[TPNV11] Daniel Tang, Ales Plsek, Kelvin Nilsen, and Jan Vitek. A static memory safety annotation system for Safety Critical Java. In *RTSS 2011: The 32nd IEEE Real-Time Systems Symposium*, Vienna, Austria, November 2011.

[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.

[Val18] Lembit Valgma. Usable and sound static analysis through its integration into automated and interactive workflows. Master's thesis, U. of Tartu, Tartu, Estonia, 2018.

[VPEJ15] Mohsen Vakilian, Amarin Phaosawasdi, Michael D. Ernst, and Ralph E. Johnson. Cascade: A universal programmer-assisted type qualifier inference tool. In *ICSE 2015, Proceedings of the 37th International Conference on Software Engineering*, pages 234–245, Florence, Italy, May 2015.

[Wan21] Di Wang. Interval type inference: Improvements and evaluations. Master's thesis, University of Waterloo Department of Electrical and Computer Engineering, Waterloo, Ontario, Canada, December 2021.

[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.

[XCD21] Weitian Xing, Yuanhui Cheng, and Werner Dietl. Ensuring correct cryptographic algorithm and provider usage at compile time. In *FTfJP: 239d Workshop on Formal Techniques for Java-like Programs*, page 43–50, Online, July 2021.

[Xia20] Tongtong Xiang. Type checking and whole-program inference for value range analysis. Master's thesis, University of Waterloo Department of Electrical and Computer Engineering, Waterloo, Ontario, Canada, October 2020.

[Xin20] Weitian Xing. Light-weight verification of cryptographic API usage. Master's thesis, U. of Waterloo, Waterloo, Ontario, Canada, 2020.

[XLD20] Tongtong Xiang, Jeff Y. Luo, and Werner Dietl. Precise inference of expressive units of measurement types. In *OOPSLA 2020, Object-Oriented Programming Systems, Languages, and Applications*, Chicago, IL, USA, November 2020.

[Zaz13] Nosheen Zaza. Evaluating the accuracy of annotations in the Loci 3.0 pluggable type checker. Master's thesis, Uppsala University, Uppsala, Sweden, 2013.

[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.