The Checker Framework Manual:
Custom pluggable types for Java

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

Version 1.8.0 (4 Apr 2014)

For the impatient: Section 1.3 (page 11) describes how to **install and use** pluggable type-checkers.

Contents

Chapter 1

Introduction

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

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

1. Nullness Checker for null pointer errors (see Chapter 3, page 22)
2. Initialization Checker to ensure all fields are set in the constructor (see Chapter 3.8, page 32)
3. Map Key Checker to track which values are keys in a map (see Chapter 3.9, page 42)
4. Interning Checker for errors in equality testing and interning (see Chapter 4, page 44)
5. Lock Checker for concurrency and lock errors (see Chapter 5, page 47)
6. Fake Enum Checker to allow type-safe fake enum patterns (see Chapter 6, page 51)
7. Tainting Checker for trust and security errors (see Chapter 7, page 54)
8. Regex Checker to prevent use of syntactically invalid regular expressions (see Chapter 8, page 56)
9. Format String Checker to ensure that format strings have the right number and type of `%` directives (see Chapter 9, page 59)
10. Property File Checker to ensure that valid keys are used for property files and resource bundles (see Chapter 10, page 65)
11. Internationalization Checker to ensure that code is properly internationalized (see Chapter 10.2, page 66).
12. Signature String Checker to ensure that the string representation of a type is properly used, for example in `Class.forName` (see Chapter 11, page 68).
13. Units Checker to ensure operations are performed on correct units of measurement (see Chapter 13, page 74)
14. Linear Checker to control aliasing and prevent re-use (see Chapter 14, page 77)
15. IGJ Checker for mutation errors (incorrect side effects), based on the IGJ type system (see Chapter 15, page 79)
16. Javari Checker for mutation errors (incorrect side effects), based on the Javari type system (see Chapter 16, page 83)
17. Subtyping Checker for customized checking without writing any code (see Chapter 17, page 85)
18. Third-party checkers that are distributed separately from the Checker Framework (see Chapter 18, page 88)

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 17 and 23.

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 3–18).
- Skim the advanced material that will enable you to make more effective use of a type system (Chapters 19–26), so that you will know what is available and can find it later. Skip Chapter 23 on creating a new checker.

1.2 How it works: Pluggable types

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

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

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

1.3 Installation

This section describes how to install the Checker Framework for use from the command line, Ant, and Maven. If you wish to use the Checker Framework from Eclipse, see the Checker Framework Eclipse Plugin webpage: `http://types.cs.washington.edu/checker-framework/eclipse/`. The Checker Framework release contains everything that you need, both to run checkers and to write your own checkers. As an alternative, you can build the latest development version from source (Section 26.3, page 156).

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

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

1. Download the Checker Framework distribution:
   `http://types.cs.washington.edu/checker-framework/current/checker-framework.zip`
2. Unzip it to create a `checker-framework` directory.
3. Optionally, update your execution path or create an alias.
   When doing pluggable type-checking, you need to use the "Checker Framework compiler". The Checker Framework compiler in turn uses the "Type Annotations compiler", an advance version of the OpenJDK 8 javac compiler that understands type annotations. The Type Annotations compiler is backward-compatible, so using it as your Java compiler, even when you are not doing pluggable type-checking, should have no negative consequences.
   You can use the Checker Framework compiler in three ways. You can use any one of them. However, if you are using the Windows command shell, you must use the last one.
   First, set a `CHECKERFRAMEWORK` environment variable to the `.../checker-framework` directory. Alternately, you can use an absolute path where the below instructions use the `CHECKERFRAMEWORK` environment variable.
   - Add directory `$CHECKERFRAMEWORK/checker/bin` to your path, *before* any other directory that contains a `javac` executable. Now, whenever you run `javac`, you will use the updated compiler. If you are using the bash shell, a way to do this is to add the following to your `~/.bashrc` file:
     ```
     export PATH=${CHECKERFRAMEWORK}/checker/bin:${PATH}
     ```
   - Whenever this document tells you to run `javac`, you can instead run `$CHECKERFRAMEWORK/checker/bin/javac`. You can simplify this by introducing an alias. Then, whenever this document tells you to run `javac`, instead use that alias. Here is the syntax for your `~/.bashrc` file:

```
alias javacheck='$CHECKERFRAMEWORK/checker/bin/javac'
```

- 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 ...
  ```

  More generally, anywhere that you 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`.
  You can simplify the above command by introducing an alias. Then, whenever this document tells you to run `javac`, instead use that alias. For example:

  ```
  # Unix
  alias javacheck='java -jar $CHECKERFRAMEWORK/checker/dist/checker.jar'

  # Windows
  doskey javacheck=java -jar %CHECKERFRAMEWORK%\checker\dist\checker.jar $*
  ```

To ensure that it was installed properly, run `javac -version` (possibly using the full pathname to `javac` or the alias, if you did not add the Type Annotations `javac` to your path).

The output should be:

```
javac 1.7.0-jsr308-1.8.0
```

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

Section 1.4 walks you through a simple example. More detailed instructions for using a checker appear in Chapter 2. There is also a tutorial (`http://types.cs.washington.edu/checker-framework/tutorial/`) that walks you through how to use the Checker Framework in Eclipse or on the command line.

1.4 Example use: detecting a null pointer bug

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

To run a checker on a source file, just run javac as usual, passing the `-processor` flag. (You can also use an IDE or other build tool; see Chapter 24.)

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

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

then you will instead use the command line:

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

but take note that the `javac` command must refer to the Type Annotations compiler (see Section 1.3).

If you usually do your coding within an IDE, you will need to configure the IDE. This manual contains instructions for Ant (Section 24.2), Maven (Section 24.3), IntelliJ IDEA (Section 24.5), Eclipse (Section 24.6), and tIDE (Section 24.7). Otherwise, see your IDE documentation for details.

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

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

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

2. Run the Nullness Checker on the class. Either run this command:

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

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

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

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

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

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

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

Chapter 2

Using a checker

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

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

1. The programmer writes annotations, such as `@NonNull` and `@Interned`, that specify additional information about Java types. (Or, the programmer uses an inference tool to automatically insert annotations in his code: see Sections 3.3.7 and 16.2.2.) It is possible to annotate only part of your code: see Section 21.1.
2. The checker reports whether the program contains any erroneous code — that is, code that is inconsistent with the annotations.

This chapter is structured as follows:

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

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

- Chapter 20: Advanced type system features
- Chapter 21: Handling warnings and legacy code
- Chapter 22: Annotating libraries
- Chapter 23: How to create a new checker
- Chapter 24: Integration with external tools

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

2.1 Writing annotations

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

```
@Interned String intern() { ... }               // return value
int compareTo(@NonNull String other) { ... }    // parameter
String toString(@ReadOnly MyClass this) { ... } // receiver ("this" parameter)
```

```
@NonNull List<@Interned String> messages;       // generics:  non-null list of interned Strings
@Interned String @NonNull [] messages;          // arrays:  non-null array of interned Strings
myDate = (@ReadOnly Date) readonlyObject;       // cast
```

You can also write the annotations within comments, as in `List</*@NonNull*/ String>`. The Type Annotations compiler, which is distributed with the Checker Framework, will still process the annotations. However, your code will remain compilable by people who are not using the Type Annotations compiler. For more details, see Section 21.3.1.

2.1.1 Distributing your annotated project

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

- To perform pluggable type-checking, all of the Checker Framework (which also contains the annotation declarations) is needed.
- To compile the code:
  - If you wrote annotations in comments (see Section 21.3.1) and/or used implicit import statements (see Section 21.3.2), then the code can be compiled by any Java compiler, without needing declarations of the annotations.
  - Otherwise, compiling the code requires a declaration of the annotations. These appear in the full Checker Framework. Additionally, the Checker Framework distribution `.zip` file contains a small jar file, `checker-qual.jar`, that only contains the definitions of the distributed qualifiers, without any support for type-checking.
- To run the code:
  - If you compiled the code without using the annotation declarations, then no annotation declarations are needed.
  - If you compiled the code using the annotation declarations, then users may need to have the annotation declarations on their classpath.

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

2.2 Running a checker

To run a checker plugin, run the compiler `javac` as usual, but pass the `-processor` *`plugin_class`* command-line option. (You can run a checker from within your favorite IDE or build system. See Chapter 24 for details about Ant (Section 24.2), Maven (Section 24.3), IntelliJ IDEA (Section 24.5), Eclipse (Section 24.6), and tIDE (Section 24.7), and about customizing other IDEs and build tools.) Remember that you must be using the Type Annotations version of `javac`, which you already installed (see Section 1.3).

Two concrete examples (using the Nullness Checker) are:

```
javac -processor org.checkerframework.checker.nullness.NullnessChecker MyFile.java
java -jar $CHECKERFRAMEWORK/checker/dist/checker.jar -processor org.checkerframework.checker.nullness.NullnessChecker MyFile.java
```

Note that the two invocations above are equivalent. Each one invokes the Checker Framework compiler, which in turn invokes the Type Annotations compiler with an annotated JDK on its classpath. For more information on annotated JDKs, see Section 22.3.

The checker is run on only the Java files that javac compiles. This includes all Java files specified on the command line (or created by another annotation processor). It may also include other of your Java files (but not if a more recent `.class` file exists). Even when the checker does not analyze a class (say, the class was already compiled, or source code is not available), it does check the *uses* of those classes in the source code being compiled.

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

2.2.1 Summary of command-line options

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

Unsound checking: ignore some errors

- `-AskipUses`, `-AonlyUses` Suppress all errors and warnings at all uses of a given class — or at all uses except those of a given class. See Section 21.2.4
- `-AskipDefs`, `-AonlyDefs` Suppress all errors and warnings within the definition of a given class — or everywhere except within the definition of a given class. See Section 21.2.5
- `-AsuppressWarnings` Suppress all warnings matching the given key; see Section 21.2.2
- `-AignoreRawTypeArguments` Ignore subtype tests for type arguments that were inferred for a raw type. If possible, it is better to write the type arguments. See Section 19.1.1.
- `-AassumeSideEffectFree` Unsoundly assume that every method is side-effect-free; see Section 20.4.3.
- `-AassumeAssertionsAreEnabled`, `-AassumeAssertionsAreDisabled` Whether to assume that assertions are enabled or disabled; see Section 20.4.4.

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

- `-AenablePurity` Check the bodies of methods marked `@SideEffectFree`, `@Deterministic`, and `@Pure` to ensure the method satisfies the annotation. By default, the Checker Framework unsoundly trusts the method annotation. See Section 20.4.3.
- `-AinvariantArrays` Make array subtyping invariant; that is, two arrays are subtypes of one another only if they have exactly the same element type. By default, the Checker Framework unsoundly permits covariant array subtyping, just as Java does. See Section 20.1.
- `-AconcurrentSemantics` Whether to assume concurrent semantics (field values may change at any time) or sequential semantics; see Section 25.4.4.

Type-checking modes: enable/disable functionality

- `-Alint` Enable or disable optional checks; see Section 21.2.6.
- `-AshowSuppressWarningKeys` With each warning, show all possible keys to suppress that warning; see Section 21.2.2
- `-AsuggestPureMethods` Suggest methods that could be marked `@SideEffectFree`, `@Deterministic`, or `@Pure`; see Section 20.4.3.
- `-AcheckCastElementType` In a cast, require that parameterized type arguments and array elements are the same. By default, the Checker Framework unsoundly permits them to differ, just as Java does. See Section 19.1.6 and Section 20.1.
- `-Awarns` Treat checker errors as warnings. If you use this, you may wish to also supply `-Xmaxwarns 10000`, because by default `javac` prints at most 100 warnings.

Stub libraries

- `-Astubs` List of stub files or directories; see Section 22.2.1.
- `-AstubWarnIfNotFound` Warn if a stub file entry could not be found; see Section 22.2.1.

Debugging

- `-AprintAllQualifiers`, `-Adetailedmsgtext`, `-AprintErrorStack`, `-Anomsgtext` Amount of detail in messages; see Section 23.8.1.
- `-Aignorejdkastub`, `-Anocheckjdk` `-AstubDebug`, Stub and JDK libraries; see Section 23.8.2
- `-Afilenames`, `-Ashowchecks` Progress tracing; see Section 23.8.3
- `-Aflowdotdir`, `-AresourceStats` Miscellaneous debugging options; see Section 23.8.4

Some checkers support additional options, which are described in that checker's manual section. For example, `-Aquals` tells the Subtyping Checker (see Chapter 17) and the Fenum Checker (see Chapter 6) which annotations to check.

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

- `-processor` Names the checker to be run; see Section 2.2
- `-processorpath` Indicates where to search for the checker; should also contain any qualifiers used by the Subtyping Checker; see Section 17.2
- `-proc:{none,only}` Controls whether checking happens; `-proc:none` means to skip checking; `-proc:only` means to do only checking, without any subsequent compilation; see Section 2.2.2
- `-Xbootclasspath/p:` Indicates where to find the annotated JDK classes; see Section 22.3
- `-implicit:class` Suppresses warnings about implicitly compiled files (not named on the command line); see Section 24.2
- `-XDTA:noannotationsincomments` and `-XDTA:spacesincomments` to turn off parsing annotation comments and to turn on parsing annotation comments even when they contain spaces; applicable only to the Type Annotations compiler; see Section 21.3.1
- `-J` Supply an argument to the JVM that is running javac; example:
  `-J-Djsr308_imports=org.checkerframework.checker.nullness.qual.*:org.checkerframework.dataflow.qual.*`
  See Section 21.3.2

2.2.2 Checker auto-discovery

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

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

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

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

2.3 What the checker guarantees

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

There are some limitations to the guarantee.

- A compiler plugin can check only those parts of your program that you run it on. If you compile some parts of your program without running the checker, then there is no guarantee that the entire program satisfies the property being checked. Some examples of un-checked code are:
  - Code compiled without the `-processor` switch, including any external library supplied as a `.class` file.
  - Code compiled with the `-AskipUses`, `-AonlyUses`, `-AskipDefs` or `-AonlyDefs` properties (see Section 21.2).
  - Suppression of warnings, such as via the `@SuppressWarnings` annotation (see Section 21.2).
  - Native methods (because the implementation is not Java code, it cannot be checked).

  In each of these cases, any *use* of the code is checked — for example, a call to a native method must be compatible with any annotations on the native method's signature. However, the annotations on the un-checked code are trusted; there is no verification that the implementation of the native method satisfies the annotations.
- The Checker Framework is, by default, unsound in a few places where a conservative analysis would issue too many false positive warnings. These are listed in Section 2.2.1; as of January 2014, they are: (1) purity

annotations are trusted, (2) array types are covariant, and (3) programs are assumed to be single-threaded. You can supply a command-line argument to make the Checker Framework sound for each of these cases.
- Specific checkers may have other limitations; see their documentation for details.

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

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

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

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

If you find that a checker fails to issue a warning that it should, then please report a bug (see Section 26.2).

2.4 Tips about writing annotations

2.4.1 How to get started annotating legacy code

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

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

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.

It is best to annotate one package at a time, and to annotate the entire package so that you don't forget any classes (failing to annotate a class can lead to unexpected results). Start as close to the leaves of the call tree as possible, such as with libraries — that is, start with methods/classes/packages that have few dependencies on other code or, equivalently, start with code that a lot of your other code depends on. The reason for this is that it is easiest to annotate a class if the code it calls has already been annotated.

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

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

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

Also see Chapter 21, which tells you what to do when you are unable to eliminate checker warnings.

2.4.2 Do not annotate local variables unless necessary

The checker infers annotations for local variables (see Section 20.4). Usually, you only need to annotate fields and method signatures. After doing those, you can add annotations inside method bodies if the checker is unable to infer the correct annotation, if you need to suppress a warning (see Section 21.2), etc.

2.4.3 Annotations indicate normal behavior

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

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

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

`@Nullable` (see Section 3.2) might seem like a reasonable annotation for the parameter, for two reasons. First, `null` is a legal argument with a well-defined semantics: throw an exception. Second, `@Nullable` describes a possible program execution: it might be possible for `null` to flow there, if your program has a bug.

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

Therefore, you should mark such parameters as `@NonNull`, indicating the intended use of the method. When you use the `@NonNull` annotation, the checker is able to issue compile-time warnings about possible run-time exceptions, which is its purpose. Marking the parameter as `@Nullable` would suppress such warnings, which is undesirable.

2.4.4 Subclasses must respect superclass annotations

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

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

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

There exist implementations of Multiset that permit `null` elements, and implementations of Multiset that do not permit `null` elements. A client with a variable `Multiset ms` does not know which variety of Multiset `ms` refers to. However, the `@Nullable` annotation promises that `ms.add(null, 1)` is permissible. (Recall from Section 2.4.3 that annotations should indicate normal behavior.)

If parameter `element` on line 141 were to be annotated, the correct annotation would be `@NonNull`. Suppose a client has a reference to same Multiset `ms`. The only way the client can be sure not to throw an exception is to pass only non-`null` elements to `ms.add()`. A particular class that implements Multiset could declare `add` to take a `@Nullable` parameter. That still satisfies the original contract. It strengthens the contract by promising even more: a client with such a reference can pass any non-`null` value to `add()`, and may also pass `null`.

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

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

or, more generally, as

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

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

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

2.4.5 Annotations on constructor invocations

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

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

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

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

In general, you should only use an annotation on a constructor invocation when you know that the cast is guaranteed to succeed. An example from the IGJ checker (Chapter 15) is `new @Immutable MyClass()` or `new @Mutable MyClass()`, where you know that every other reference to the class is annotated `@ReadOnly`.

2.4.6 When to use (and not use) type qualifiers

For some programming tasks, you can use either a Java subclass or a type qualifier. For instance, suppose that your code currently uses `String` to represent an address. You could create a new `Address` class and refactor your code to use it, or you could create a `@Address` annotation and apply it to some uses of `String` in your code. If both of these are truly possible, then it is probably more foolproof to use the Java class. We do not encourage you to use type qualifiers as a poor substitute for classes. However, sometimes type qualifiers are a better choice.

Using a new class may make your code incompatible with existing libraries or clients. Brian Goetz expands on this issues in an article on the pseudo-typedef antipattern [Goe06]. Even if compatibility is not a concern, a code change may introduce bugs, whereas adding annotations does not change the run-time behavior. It is possible to add annotations

to existing code, including code you do not maintain or cannot change (for code that strictly cannot be changed, it is recommended to add annotations in comments — see Section 21.3.1). It is possible to annotate primitive types without converting them to wrappers, which would make the code both uglier and slower.

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

Type qualifiers permit you to remove operations, with a compile-time guarantee. An example is mutating methods that are forbidden by immutable types (see Chapters 15 and 16). More generally, type qualifiers permit creating a new supertype, not just a subtype, of an existing Java type.

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

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

When you first run a type-checker on your code, it is likely to issue warnings or errors. For each warning, try to understand why the checker issues it. For example, if you are using the Nullness Checker (Chapter 3, page 22), try to understand why it cannot prove that no null pointer exception ever occurs. There are three general reasons, listed below. You will need to examine your code, and possibly write test cases, to understand the reason.

1. There is a bug in your code, such as an actual possible null dereference. Fix your code to prevent that crash.
2. There is a weakness in the annotations. Improve the annotations. For example, continuing the Nullness Checker example, if a particular variable is annotated as `@Nullable` but it actually never contains `null` at run time, then change the annotation to `@NonNull`. The weakness might be in the annotations in your code, or in the annotations in a library that your code calls. Another possible problem is that a library is unannotated (see Chapter 22, page 116).
3. There is a weakness in the type-checker. Then your code is safe — it never suffers the error at run time — but the checker cannot prove this fact. The checker is not omniscient, and some tricky coding paradigms are beyond its analysis capabilities. In this case, you should suppress the warning; see Chapter 21.2, page 109. (Alternatively, if the weakness is a bug in the checker, then please report the bug; see Chapter 26.2, page 155.)

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

Chapter 3

Nullness Checker

If the Nullness Checker issues no warnings for a given program, then running that program will never throw a null pointer exception. This guarantee enables a programmer to prevent errors from occurring when a program is run. See Section 3.1 for more details about the guarantee and what is checked.

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

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

The NullnessChecker is actually an ensemble of three pluggable type-checkers that work together: the Nullness Checker proper (which is the main focus of this chapter), the Initialization Checker (Section 3.8), and the Map Key Checker (Section 3.9).

3.1 What the Nullness Checker checks

The checker issues a warning in these cases:

1. When an expression of non-`@NonNull` type is dereferenced, because it might cause a null pointer exception. Dereferences occur not only when a field is accessed, but when an array is indexed, an exception is thrown, a lock is taken in a synchronized block, and more. For a complete description of all checks performed by the Nullness Checker, see the Javadoc for `NullnessVisitor`.
2. When an expression of `@NonNull` type might become null, because it is a misuse of the type: the null value could flow to a dereference that the checker does not warn about.
   As a special case of an of `@NonNull` type becoming null, the checker also warns whenever a field of `@NonNull` type is not initialized in a constructor. Also see the discussion of the `-Alint=uninitialized` command-line option below.

This example illustrates the programming errors that the checker detects:

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

Parameter passing and return values are checked analogously to assignments.

The Nullness Checker also checks the correctness, and correct use, of rawness annotations for checking initialization (see Section 3.8.6) and of map key annotations (see Section 3.9).

The checker performs additional checks if certain `-Alint` command-line options are provided. (See Section 21.2.6 for more details about the `-Alint` command-line option.)

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

3.2 Nullness annotations

The Nullness Checker uses three separate type hierarchies: one for nullness, one for rawness (Section 3.8.6), and one for map keys (Section 3.9) The Nullness Checker has four varieties of annotations: nullness type qualifiers, nullness method annotations, rawness type qualifiers, and map key type qualifiers.

3.2.1 Nullness qualifiers

The nullness hierarchy contains these qualifiers:

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

`@NonNull` indicates a type that does not include the null value. The type `boolean` is non-null; a variable of type `boolean` always has one of the values `true` or `false`. The type `@NonNull Boolean` is also non-null: a variable of type `@NonNull Boolean` always has one of the values `TRUE` or `FALSE` — never `null`. Dereferencing an expression of non-null type can never cause a null pointer exception.
The `@NonNull` annotation is rarely written in a program, because it is the default (see Section 3.3.2).

`@PolyNull` indicates qualifier polymorphism. For a description of `@PolyNull`, see Section 19.2.

`@MonotonicNonNull` indicates a reference that may be `null`, but if it ever becomes non-`null`, then it never becomes `null` again. This is appropriate for lazily-initialized fields, among other uses. When the variable is read, its type is treated as `@Nullable`, but when the variable is assigned, its type is treated as `@NonNull`.
Because the Nullness Checker works intraprocedurally (it analyzes one method at a time), when a `MonotonicNonNull` field is first read within a method, the field cannot be assumed to be non-null. The benefit of MonotonicNonNull over Nullable is its different interaction with flow-sensitive type qualifier refinement (Section 20.4). After a check of a MonotonicNonNull field, all subsequent accesses *within that method* can be assumed to be NonNull, even after arbitrary external method calls that have access to the given field.
It is permitted to initialize a MonotonicNonNull field to null, but the field may not be assigned to null anywhere else in the program. If you supply the `noInitForMonotonicNonNull` lint flag (for example, supply `-Alint=noInitForMonotonicNonNull` on the command line), then @MonotonicNonNull fields are not allowed to have initializers.
Use of `@MonotonicNonNull` on a static field is a code smell: it may indicate poor design. You should consider whether it is possible to make the field a member field that is set in the constructor.

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

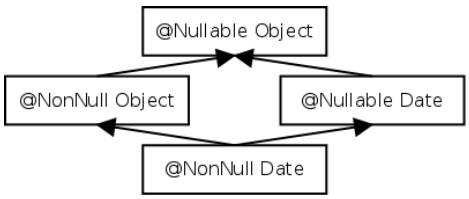

Figure 3.1: Partial type hierarchy for the Nullness type system. Java's `Object` is expressed as `@Nullable Object`. Programmers can omit most type qualifiers, because the default annotation (Section 3.3.2) is usually correct. The Nullness Checker verifies three type hierarchies: this one for nullness, one for initialization (Section 3.8), and one for map keys (Section 3.9).

3.2.2 Nullness method annotations

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

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

`@EnsuresNonNull`

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

3.2.3 Initialization qualifiers

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

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

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

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

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

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

3.2.4 Map key qualifiers

The Nullness Checker invokes a Map Key Checker, whose annotation, `@KeyFor`, indicates that a value is a key for a given map. This indicates whether `map.containsKey(value)` would evaluate to `true`.

`@KeyFor`

Use of this annotation can help you to type-check more code. For details, see Section 3.9.

3.3 Writing nullness annotations

3.3.1 Implicit qualifiers

As described in Section 20.3, the Nullness Checker adds implicit qualifiers, reducing the number of annotations that must appear in your code. For example, enum types are implicitly non-null, so you never need to write `@NonNull MyEnumType`.

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

3.3.2 Default annotation

Unannotated references are treated as if they had a default annotation. The standard defaulting rule is "CLIMB to top", described in Section 20.3.2. Its effect is to default all types to `@NonNull`, except that `@Nullable` is used for casts, locals, instanceof, and implicit bounds. A user can choose a different defaulting rule.

3.3.3 Conditional nullness

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

Consider `java.lang.Class`. Method `Class.getComponentType()` may return null, but it is specified to return a non-null value if `Class.isArray()` is true. You could declare this relationship in the following way (this particular example is already done for you in the annotated JDK that comes with the Checker Framework):

```
class Class {
  @EnsuresNonNullIf(expression="getComponentType()", result=true)
  public native boolean isArray();

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

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

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

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

The argument to `@EnsuresNonNullIf` is a Java expression, including method calls (as shown above), method formal parameters, fields, etc.; for details, see Section 20.5. More examples of the use of these annotations appear in the Javadoc for `@EnsuresNonNullIf`.

3.3.4 Nullness and arrays

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

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

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

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

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

3.3.5 Run-time checks for nullness

When you perform a run-time check for nullness, such as `if (x != null) ...`, then the Nullness Checker refines the type of `x` to `@NonNull` within the scope of the test. For more details, see Section 20.4.

3.3.6 Additional details

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

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

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

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

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

It can be tedious to write annotations in your code. Tools exist that can automatically infer annotations and insert them in your source code. (This is different than type qualifier refinement for local variables (Section 20.4), which infers a more specific type for local variables and uses them during type-checking but does not insert them in your source code. Type qualifier refinement is always enabled, no matter how annotations on signatures got inserted in your source code.)

Your choice of tool depends on what default annotation (see Section 3.3.2) your code uses. You only need one of these tools.

- Inference of `@Nullable`: If your code uses the standard CLIMB-to-top default (Section 20.3.2) or the `NonNull` default, then use the AnnotateNullable tool of the Daikon invariant detector.
- Inference of `@NonNull`: If your code uses the Nullable default, use one of these tools:
  - Julia analyzer,
  - Nit: Nullability Inference Tool,
  - Non-null checker and inferencer of the JastAdd Extensible Compiler.

3.4 Suppressing nullness warnings

The Checker Framework supplies several ways to suppress warnings, most notably the `@SuppressWarnings("nullness")` annotation (see Section 21.2). An example use is

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

void myMethod() {
  // The programmer knows that this particular call never returns null,
  // perhaps based on the arguments or the state of other objects.
  @SuppressWarnings("nullness")
  @NonNull Object o2 = getObject(...);
```

The Nullness Checker supports an additional warning suppression key, `nullness:generic.argument`. Use of `@SuppressWarnings("nullness:generic.argument")` causes the Nullness Checker to suppress warnings related to misuse of generic type arguments. One use for this key is when a class is declared to take only `@NonNull` type arguments, but you want to instantiate the class with a `@Nullable` type argument, as in `List<@Nullable Object>`. For a more complete explanation of this example, see Section 25.6.1, page 148.

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

3.4.1 Suppressing warnings with assertions and method calls

Occasionally, it is inconvenient or verbose to use the `@SuppressWarnings` annotation. For example, Java does not permit annotations such as `@SuppressWarnings` to appear on statements. In such cases, you may be able to use the `@AssumeAssertion` string in an `assert` message (see Section 21.2.3).

For situations when all of these approaches are inconvenient, the Nullness Checker provides an additional way to suppress warnings: via the `castNonNull` method. This is appropriate when the Nullness Checker issues a warning, but the programmer knows for sure that the warning is a false positive, because the value cannot ever be null at run time.

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

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

   If the string "`@AssumeAssertion(nullness)`" does not appear in the assertion message, then the Nullness Checker treats the assertion as being used for defensive programming, and it warns if the method might throw a nullness-related exception.
   A downside of putting the string in the assertion message is that if the assertion ever fails, then a user might see the string and be confused. But the string should only be used if the programmer has reasoned that the assertion can never fail.
2. Use the `NullnessUtils.castNonNull` method.
   The Nullness Checker considers both the return value, and also the argument, to be non-null after the method call. Therefore, the `castNonNull` method can be used either as a cast expression or as a statement. The Nullness Checker issues no warnings in any of the following code:

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

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

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

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

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

3.4.2 Suppressing warnings on nullness-checking routines and defensive programming

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.5 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 JSR 308 compiler that is distributed with the Checker Framework):

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

Compilation will complete without warnings.

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

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

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

3.5.2 Annotated library

Some libraries that are annotated with nullness qualifiers are:

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

3.6 Tips for getting started

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

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

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

- in Javadoc: add `@Nullable` annotations to method signatures (parameters and return types).
- `return null`: add a `@Nullable` annotation to the return type of the given method.
- *`param`* `== null`: when a formal parameter is compared to `null`, then in most cases you can add a `@Nullable` annotation to the formal parameter's type
- *`TypeName field`* `= null;`: when a field is initialized to `null` in its declaration, then it needs either a `@Nullable` or a `@MonotonicNonNull` annotation. If the field is always set to a non-null value in the constructor, then you can just change the declaration to *`Type field`*`;`, without an initializer, and write no type annotation (because the default is `@NonNull`).

- declarations of `contains`, `containsKey`, `containsValue`, `equals`, `get`, `indexOf`, `lastIndexOf`, and `remove` (with `Object` as the argument type): change the argument type to `@Nullable Object`; for `remove`, also change the return type to `@Nullable Object`.

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

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

Here are some other tips:

- In any file where you write an annotation such as `@Nullable`, don't forget to add `import org.checkerframework.checker.nullness.qual.*;`.
- To indicate an array that can be null, write, for example: `int @Nullable []`.
  By contrast, `@Nullable Object []` means a non-null array that contains possibly-null objects.
- If you know that a particular variable is definitely not null, but the Nullness Checker cannot figure it out, then you can tell it by writing an assertion (see Section 21.2):

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

- To indicate that a routine returns the same value every time it is called, use `@Pure` (see Section 20.4.3).
- To indicate a method precondition (a contract stating the conditions under which a client is allowed to call it), you can use annotations such as `@RequiresNonNull` (see Section 3.2.2).

3.7 Other tools for nullness checking

The Checker Framework's nullness annotations are similar to annotations used in IntelliJ IDEA, FindBugs, JML, the JSR 305 proposal, NetBeans, and other tools. Also see Section 26.5 for a comparison to other tools.

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

If your code is already annotated with a different nullness annotation, you can reuse that effort. The Checker Framework comes with cleanroom re-implementations of annotations from other tools. It treats them exactly as if you had written the corresponding annotation from the Nullness Checker, as described in Figure 3.2.

Alternately, the Checker Framework can process those other annotations (as well as its own, if they also appear in your program). The Checker Framework has its own definition of the annotations on the left side of Figure 3.2, so that they can be used as type qualifiers. The Checker Framework interprets them according to the right side of Figure 3.2.

The Checker Framework may issue more or fewer errors than another tool. This is expected, since each tool uses a different analysis. Remember that the Checker Framework aims at soundness: it aims to never miss a possible null dereference, while at the same time limiting false reports. Also, note FindBugs's non-standard meaning for `@Nullable` (Section 3.7.2).

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

Note that some older tools interpret array and vararg declarations inconsistently with the Java specification. For example, they might interpret `@NonNull Object []` as "non-null array of objects", rather than as "array of non-null objects" which is the correct Java interpretation. Such an interpretation is unfortunate and confusing. See Section 25.5.3 for some more details about this issue.

3.7.1 Which tool is right for you?

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

The Checker Framework has a more powerful nullness analysis; FindBugs misses some real errors. However, FindBugs does not require you to annotate your code as thoroughly as the Checker Framework does. Depending on the importance of your code, you may desire: no nullness checking, the cursory checking of FindBugs, or the thorough

com.sun.istack.NotNull
edu.umd.cs.findbugs.annotations.NonNull
javax.annotation.Nonnull
javax.validation.constraints.NotNull
org.eclipse.jdt.annotation.NonNull
org.jetbrains.annotations.NotNull
org.netbeans.api.annotations.common.NonNull
org.jmlspecs.annotation.NonNull

$\Rightarrow$ org.checkerframework.checker.nullness.qual.NonNull

com.sun.istack.Nullable
edu.umd.cs.findbugs.annotations.Nullable
edu.umd.cs.findbugs.annotations.CheckForNull
edu.umd.cs.findbugs.annotations.UnknownNullness
javax.annotation.Nullable
javax.annotation.CheckForNull
org.eclipse.jdt.annotation.Nullable
org.jetbrains.annotations.Nullable
org.netbeans.api.annotations.common.CheckForNull
org.netbeans.api.annotations.common.NullAllowed
org.netbeans.api.annotations.common.NullUnknown
org.jmlspecs.annotation.Nullable

$\Rightarrow$ org.checkerframework.checker.nullness.qual.Nullable

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

checking of the Checker Framework. You might even want to ensure that both tools run, for example if your coworkers or some other organization are still using FindBugs. If you know that you will eventually want to use the Checker Framework, there is no point using FindBugs first; it is easier to go straight to using the Checker Framework.

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

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

If FindBugs (or any other tool) discovers a nullness error that the Checker Framework does not, please report it to us (see Section 26.2) so that we can enhance the Checker Framework.

3.7.2 Incompatibility note about FindBugs `@Nullable`

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

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

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

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

void myMethod() {
  @Nullable Object o = getObject();
```

```
    // FindBugs issues no warning about calling toString on a possibly-null reference!
    o.toString();
  }
```

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

With FindBugs, you annotate a declaration, which suppresses checking at *all* client uses, even the places that you want to check. It is better to suppress warnings at only the specific client uses where the value is known to be non-null; the Checker Framework supports this, if you write `@SuppressWarnings` at the client uses. The Checker Framework also supports suppressing checking at all client uses, by writing a `@SuppressWarnings` annotation at the declaration site. Thus, the Checker Framework supports both use cases, whereas FindBugs supports only one and gives the programmer less flexibility.

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

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

3.8 Initialization Checker

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

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

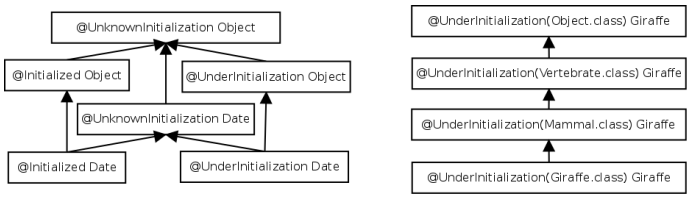

Figure 3.3: Partial type hierarchy for the Initialization type system. `@UnknownInitialization` and `@UnderInitialization` each take an optional parameter indicating how far initialization has proceeded, and the right side of the figure illustrates its type hierarchy in more detail.

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

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

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

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

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

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

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

For compatibility with Java 7 and earlier, you can write the receiver parameter in comments (see Section 21.3.1):

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

3.8.1 Initialization qualifiers

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

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

`@UnknownInitialization` indicates a type that may contain a partially-initialized object. In a partially-initialized object, fields that are annotated as `@NonNull` may be null because the field has not yet been assigned. `@UnknownInitialization` takes a parameter that is the class the object is definitely initialized up to. For instance, the type `@UnknownInitialization(Foo.class)` denotes an object in which every fields declared in

Declarations	Expression	Expression's nullness type, or checker error

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

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

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

`Foo` or its superclasses is initialized, but other fields might not be. Just `@UnknownInitialization` is equivalent to `@UnknownInitialization(Object.class)`.

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

A partially-initialized object (`this` in a constructor) may be passed to a helper method or stored in a variable; if so, the method receiver, or the field, would have to be annotated as `@UnknownInitialization` or as `@UnderInitialization`.

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

The initialization hierarchy is orthogonal to the nullness hierarchy. It is legal for a reference to be `@NonNull @UnderInitialization`, `@Nullable @UnderInitialization`, `@NonNull @Initialized`, or `@Nullable @Initialized`. The nullness hierarchy tells you about the reference itself: might the reference be null? The initialization hierarchy tells you about the `@NonNull` fields in the referred-to object: might those fields be temporarily null in contravention of their type annotation? Figure 3.4 contains some examples.

3.8.2 How an object becomes initialized

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

The Initialization Checker issues an error if the constructor fails to initialize any `@NonNull` field. This ensures that the object is in a legal (initialized) state by the time that the constructor exits. This is different than Java's test for definite assignment (see JLS ch.16), which does not apply to fields (except blank final ones, defined in JLS §4.12.4) because fields have a default value of null.

All `@NonNull` fields must either have a default in the field declaration, or be assigned in the constructor or in a helper method that the constructor calls. If your code initializes (some) fields in a helper method, you will need to annotate the helper method with an annotation such as `@EnsuresNonNull({"field1", "field2"})` for all the fields that the helper method assigns. It's a bit odd, but you use that same annotation, `@EnsuresNonNull`, to indicate that a

primitive field has its value set in a helper method, which is relevant when you supply the `-Alint=uninitialized` command-line option (see Section 2).

3.8.3 Partial initialization

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

Consider a simple example:

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

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

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

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

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

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

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

1. `@UnderInitialization B`

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

3.8.4 How to handle warnings

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

- Declare the field as `@Nullable`. Recall that if you did not write an annotation, the field defaults to `@NonNull`.
- Declare the field as `@MonotonicNonNull`. This is appropriate if the field starts out as `null` but is later set to a non-null value. You may then wish to use the `@EnsuresNonNull` annotation to indicate which methods set the field, and the `@RequiresNonNull` annotation to indicate which methods require the field to be non-null.
- Initialize the field in the constructor or in the field's initializer, if the field should be initialized. (In this case, the Initialization Checker has found a bug!)
  Do *not* initialize the field to an arbitrary non-null value just to eliminate the warning. Doing so degrades your code: it introduces a value that will confuse other programmers, and it converts a clear NullPointerException into a more obscure error.

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

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

The problem is that the current object (`this`) is under initialization, but the receiver formal parameter (Section 25.5.1) of method `initHelper()` is implicitly annotated as `@Initialized`. If `initHelper()` doesn't depend on its receiver being initialized — that is, it's OK to call `x.initHelper` even if `x` is not initialized — then you can indicate that:

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

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

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

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

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

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

the Nullness Checker issues the following warning:

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

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

3.8.5 More details about initialization checking

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

Checking initialization of all fields, not just `@NonNull` ones When the `-Alint=uninitialized` command-line option is provided, then an object is considered uninitialized until *all* its fields are assigned, not just the `@NonNull` ones. See Section 2.

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

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

3.8.6 Rawness Initialization Checker

The Checker Framework supports two different initialization checkers that are integrated with the Nullness Checker. One uses the three annotations `@Initialized`, `@UnknownInitialization`, and `@UnderInitialization`. The other uses the two annotations `@Raw` (which corresponds to `@UnknownInitialization`) and `@NonRaw` (which corresponds to `@Initialized`). The rawness type system is slightly easier to use but slightly less expressive. In practice, you can use whichever one you prefer.

To run the Nullness Checker with the rawness variant of the Initialization Checker, invoke the NullnessRawnessChecker rather than the NullnessChecker; that is, supply the `-processor org.checkerframework.checker.nullness.NullnessRawn` command-line option to javac.

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

```
public class MyClass {
  private @NonNull Object f;
  public MyClass(int x, int y) {
```

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

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

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

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

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

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

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

For compatibility with Java 7 and earlier, you can write the receiver parameter in comments (see Section 21.3.1):

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

Rawness qualifiers

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

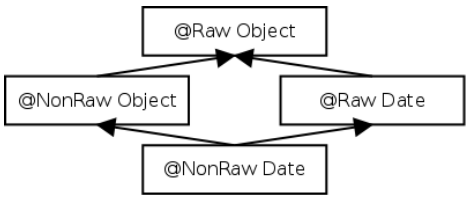

Figure 3.5: Partial type hierarchy for the Rawness Initialization type system.

Declarations	Expression	Expression's nullness type, or checker error

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

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

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

`@Raw` indicates a type that may contain a partially-initialized object. In a partially-initialized object, fields that are annotated as `@NonNull` may be null because the field has not yet been assigned. Within the constructor, `this` has `@Raw` type until all the `@NonNull` fields have been assigned. A partially-initialized object (`this` in a constructor) may be passed to a helper method or stored in a variable; if so, the method receiver, or the field, would have to be annotated as `@Raw`.

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

`@PolyRaw` indicates qualifier polymorphism over rawness (see Section 19.2).

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

The rawness hierarchy is orthogonal to the nullness hierarchy. It is legal for a reference to be `@NonNull @Raw`, `@Nullable @Raw`, `@NonNull @NonRaw`, or `@Nullable @NonRaw`. The nullness hierarchy tells you about the reference itself: might the reference be null? The rawness hierarchy tells you about the `@NonNull` fields in the referred-to object: might those fields be temporarily null in contravention of their type annotation? Figure 3.6 contains some examples.

How an object becomes non-raw

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

The Nullness Checker issues an error if the constructor fails to initialize any `@NonNull` field. This ensures that the object is in a legal (non-raw) state by the time that the constructor exits. This is different than Java's test for definite

assignment (see JLS ch.16), which does not apply to fields (except blank final ones, defined in JLS §4.12.4) because fields have a default value of null.

All `@NonNull` fields must either have a default in the field declaration, or be assigned in the constructor or in a helper method that the constructor calls. If your code initializes (some) fields in a helper method, you will need to annotate the helper method with an annotation such as `@EnsuresNonNull({"field1", "field2"})` for all the fields that the helper method assigns. It's a bit odd, but you use that same annotation, `@EnsuresNonNull`, to indicate that a primitive field has its value set in a helper method, which is relevant when you supply the `-Alint=uninitialized` command-line option (see Section 2).

Partial initialization

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

Consider a simple example:

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

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

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

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

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

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

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

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

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

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

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

Example As another example, consider the following 12 declarations:

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

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

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

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

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

More details about rawness checking

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

Checking initialization of all fields, not just `@NonNull` ones When the `-Alint=uninitialized` command-line option is provided, then an object is considered raw until *all* its fields are assigned, not just the `@NonNull` ones. See Section 2.

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

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

3.9 Map Key Checker

The Map Key Checker uses the `@KeyFor` annotation to declare that a value is a key in a given map.

This is relevant to the Nullness Checker because Java's `Map.get` method always has the possibility to return null, if the key is not in the map. In particular, a call `mymap.get(mykey)` returns non-null 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, the `@KeyFor("mymap")` type qualifier can generally be inferred.

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

Thus, for the Nullness Checker to guarantee that the value returned from `Map.get` is non-null, it is necessary that the map contains only non-null values, *and* the key is in the map. The `@KeyFor` annotation states the latter property.

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

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

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

```
Map<String, Object> m;
Set<@KeyFor("m") String> matchingKeys; // keys that match some criterion
for (@KeyFor("m") String k : matchingKeys) {
  ... m.get(k) ...  // known to be non-null
}
```

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

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

but reordering the two calls this would be OK (no compiler error):

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

Because the `@KeyFor` type hierarchy is independent from the nullness and rawness hierarchies, it uses a different warning suppression key. You can suppress warnings related to map keys with `@SuppressWarnings("keyfor")`.

When you perform 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. For more details, see Section 20.4.

Currently, the set of expressions allowed in @KeyFor is less expressive than the expressions described in Section 20.5. The Checker Framework developers are working to correct this bug.

Chapter 4

Interning Checker

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

Interning is a design pattern in which the same object is used whenever two different objects would be considered equal. Interning is also known as canonicalization or hash-consing, and it is related to the flyweight design pattern. Interning has two benefits: it can save memory, and it can speed up testing for equality by permitting use of ==.

The Interning Checker prevents two types of errors in your code. First, == should be used only on interned values; using == on non-interned values can result in subtle bugs. For example:

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

The Interning Checker helps programmers to prevent such bugs. Second, the Interning Checker also helps to prevent performance problems that result from failure to use interning. (See Section 2.3 for caveats to the checker's guarantees.)

Interning is such an important design pattern that Java builds it in for these types: `String`, `Boolean`, `Byte`, `Character`, `Integer`, `Short`. Every string literal in the program is guaranteed to be interned (JLS §3.10.5), and the `String.intern()` method performs interning for strings that are computed at run time. The `valueOf` methods in wrapper classes always (`Boolean`, `Byte`) or sometimes (`Character`, `Integer`, `Short`) return an interned result (JLS §5.1.7). Users can also write their own interning methods for other types.

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

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

To eliminate Interning Checker errors, you will need to annotate the declarations of any expression used as an argument to ==. Thus, the Interning Checker could also have been called the Reference Equality Checker. In the future, the checker will include annotations that target the non-interning cases above, but for now you need to use `@Interned`, `@UsesObjectEquals` (which handles a surprising number of cases), and/or `@SuppressWarnings`.

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

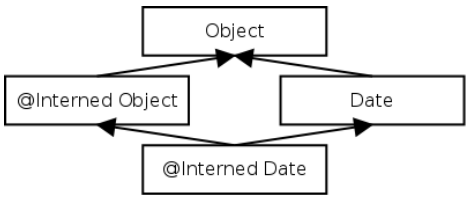

Figure 4.1: Type hierarchy for the Interning type system.

4.1 Interning annotations

These qualifiers are part of the Interning type system:

`@Interned` indicates a type that includes only interned values (no non-interned values).
`@PolyInterned` indicates qualifier polymorphism. For a description of `@PolyInterned`, see Section 19.2.
`@UsesObjectEquals` is a class (not type) annotation that indicates that this class's `equals` method is the same as that of `Object`. In other words, neither this class nor any of its superclasses overrides the `equals` method. Since `Object.equals` uses reference equality, this means that for such a class, == and `equals` are equivalent, and so the Interning Checker does not issue errors or warnings for either one.

4.2 Annotating your code with `@Interned`

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

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

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

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

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

4.2.1 Implicit qualifiers

As described in Section 20.3, the Interning Checker adds implicit qualifiers, reducing the number of annotations that must appear in your code. For example, String literals and the null literal are always considered interned, and object creation expressions (using `new`) are never considered `@Interned` unless they are annotated as such, as in

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

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

4.3 What the Interning Checker checks

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

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

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

Figure 4.2: Correspondence between other interning annotations and the Checker Framework's annotations.

This example shows both sorts of problems:

```
          Date  date;
@Interned Date idate;
...
if (date == idate) { ... }  // error: reference equality test is unsafe
idate = date;               // error: idate's referent may no longer be interned
```

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

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

You can also restrict which types the checker should examine and type-check, using the `-Acheckclass` option. For example, to find only the interning errors related to uses of `String`, you can pass `-Acheckclass=java.lang.String`. The Interning Checker always checks all subclasses and superclasses of the given class.

4.3.1 Limitations of the Interning Checker

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

4.4 Examples

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

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

Compilation will complete without errors or warnings.

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

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

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

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

4.5 Other interning annotations

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

If your code is already annotated with a different interning annotation, you can reuse that effort. The Checker Framework comes with cleanroom re-implementations of annotations from other tools. It treats them exactly as if you had written the corresponding annotation from the Interning Checker, as described in Figure 4.2.

Alternately, the Checker Framework can process those other annotations (as well as its own, if they also appear in your program). The Checker Framework has its own definition of the annotations on the left side of Figure 4.2, so that they can be used as type qualifiers. The Checker Framework interprets them according to the right side of Figure 4.2.

Chapter 5

Lock Checker

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

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

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

5.1 Lock annotations

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

`@GuardedBy` indicates a type whose value may be accessed only when the given lock is held. See the GuardedBy Javadoc for an explanation of the argument and other details. The lock acquisition and the value access may be arbitrarily far in the future; or, if the value is never accessed, the lock never need be held. Figure 5.1 gives the type hierarchy.

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

5.1.1 Examples

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

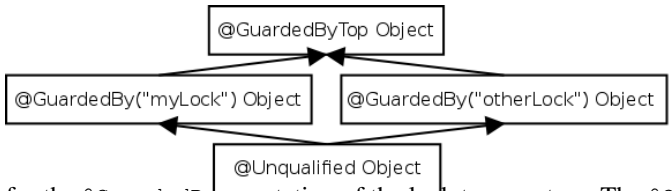

Figure 5.1: Type hierarchy for the `@GuardedBy` annotation of the lock type system. The `@GuardedByTop` annotation is for internal use by the type checker; a programmer cannot write it.

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

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

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

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

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

5.1.2 Discussion of `@Holding`

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

Higher-level synchronization protocol `@Holding` can specify a higher-level synchronization protocol that is not expressible as locks over Java objects. By requiring locks to be held, you can create higher-level protocol primitives

net.jcip.annotations.GuardedBy	⇒ org.checkerframework.checker.lock.qual.GuardedBy

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

without giving up the benefits of the annotations and checking of them.

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

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

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

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

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

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

The advantages of the `@Holding` annotation include:

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

5.2 Other lock annotations

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

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

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

5.2.1 Relationship to annotations in *Java Concurrency in Practice*

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

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

The Lock Checker renames the method annotation to `@Holding`, and it generalizes the `@GuardedBy` annotation into a type qualifier that can apply not just to a field but to an arbitrary type (including the type of a parameter, return value, local variable, generic type parameter, etc.). This makes the annotations more expressive and also more amenable to automated checking. It also accommodates the distinct meanings of the two annotations, and resolves ambiguity when `@GuardedBy` is written in a location that might apply to either the method or the return type.

(The JCIP book gives some rationales for reusing the annotation name for two purposes. One rationale is that there are fewer annotations to learn. Another rationale is that both variables and methods are "members" that can be "accessed"; variables can be accessed by reading or writing them (putfield, getfield), and methods can be accessed by calling them (invokevirtual, invokeinterface): in both cases, `@GuardedBy` creates preconditions for accessing so-annotated members. This informal intuition is inappropriate for a tool that requires precise semantics.)

5.3 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 6

Fake Enum Checker

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

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

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

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

Figure 6.1 shows part of the type hierarchy for the Fenum type system.

6.1 Fake enum annotations

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

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

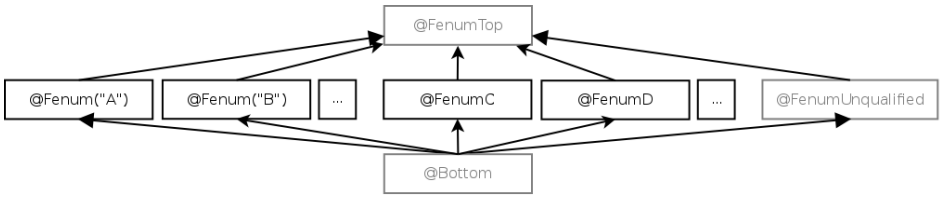

Figure 6.1: Partial type hierarchy for the Fenum type system. There are two forms of fake enumeration annotations — above, illustrated by `@Fenum("A")` and `@FenumC`. See section 6.1 for descriptions of how to introduce both types of fenums. The type qualifiers in gray (`@FenumTop`, `@FenumUnqualified`, and `@Bottom`) should never be written in source code; they are used internally by the type system.

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

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

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

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

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

6.2 What the Fenum Checker checks

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

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

It is the programmer's responsibility to ensure that fields with a fenum type are properly initialized before use. Otherwise, one might observe a `null` reference or zero value in the field of a fenum type. (The Nullness Checker (Chapter 3, page 22) can prevent failure to initialize a reference variable.)

6.3 Running the Fenum Checker

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

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

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

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

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

6.4 Suppressing warnings

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

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

6.5 Example

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

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

  public static final @Fenum("B") int BCONST1 = 4;
  public static final @Fenum("B") int BCONST2 = 5;
}

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

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

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

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

6.6 References

- Java Language Specification on enums:
  `http://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.9`
- Tutorial trail on enums:
  `http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html`
- Typesafe enum pattern:
  `http://www.oracle.com/technetwork/java/page1-139488.html`
- Java Tip 122: Beware of Java typesafe enumerations:
  `http://www.javaworld.com/article/2077487/core-java/java-tip-122-beware-of-java-typesafe-enumerations.h`

Chapter 7

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 org.checkerframework.checker.tainting.TaintingChecker` command-line option to javac.

7.1 Tainting annotations

The Tainting type system uses the following annotations:

- `@Untainted` indicates a type that includes only untainted, trusted values.
- `@Tainted` indicates a type that may include only tainted, untrusted values. `@Tainted` is a supertype of `@Untainted`.
- `@PolyTainted` is a qualifier that is polymorphic over tainting (see Section 19.2).

7.2 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 20.3.1 for overriding this default). This is the safest default, and the one that should be used for all code outside the boundary (for example, code that reads user input). You can set the default qualifier to `@Untainted` in code that may contain sensitive sinks.

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

As an example, suppose that you wish to prevent SQL injection attacks. You would start by annotating the `Statement` class to indicate that the `execute` operations may only operate on untainted queries (Chapter 22 describes how to annotate external libraries):

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

7.3 `@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 17). This lack of hard-coded behavior means that the annotations can serve many different purposes. Here are just a few examples:

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

In each case, you need to annotate the appropriate untainting/sanitization routines. This is similar to the `@Encrypted` annotation (Section 17.2), where the cryptographic functions are beyond the reasoning abilities of the type system. In each case, the type system verifies most of your code, and the `@SuppressWarnings` annotations indicate the few places where human attention is needed.

If you want more specialized semantics, or you want to annotate multiple types of tainting in a single program, then you can copy the definition of the Tainting Checker to create a new annotation and checker with a more specific name and semantics. See Chapter 23 for more details.

Chapter 8

Regex Checker for regular expression syntax

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

A regular expression, or regex, is a pattern for matching certain strings of text. In Java, a programmer writes a regular expression as a string. At run time, the string is "compiled" into an efficient internal form (`Pattern`) that is used for text-matching. Regular expression in Java also have capturing groups, which are delimited by parentheses and allow for extraction from text.

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

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

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

8.1 Regex annotations

These qualifiers make up the Regex type system:

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

`@PolyRegex` indicates qualifier polymorphism. For a description of `@PolyRegex`, see Section 19.2.

The subtyping hierarchy of the Regex Checker's qualifiers is shown in Figure 8.1.

8.2 Annotating your code with `@Regex`

8.2.1 Implicit qualifiers

As described in Section 20.3, the Regex Checker adds implicit qualifiers, reducing the number of annotations that must appear in your code. The checker implicitly adds the `Regex` qualifier with the parameter set to the correct number of capturing groups to any `String` literal that is a valid regex. The Regex Checker allows the `null` literal to be assigned to any type qualified with the `Regex` qualifier.

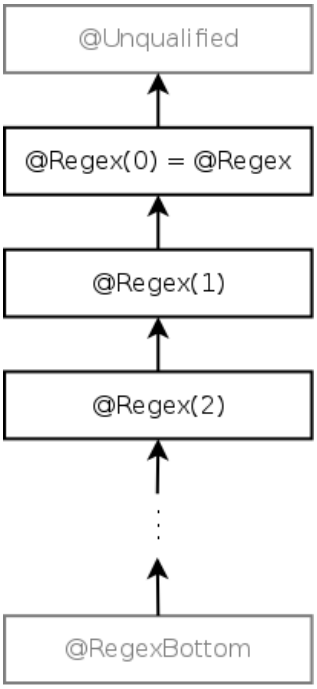

Figure 8.1: The subtyping relationship of the Regex Checker's qualifiers. Because the parameter to a `@Regex` qualifier is at least the number of capturing groups in a regular expression, a `@Regex` qualifier with more capturing groups is a subtype of a `@Regex` qualifier with fewer capturing groups. Qualifiers in gray are used internally by the type system but should never be written by a programmer.

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

Figure 8.2: An example of the Regex Checker's support for concatenation of non-regular-expression Strings to produce valid regular expression Strings.

8.2.2 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 20.4), but you may need to add `@Regex` annotations with a capturing group count to `Pattern` and `Matcher` fields and parameters.

8.2.3 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 8.2.

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

Figure 8.3: Example use of `RegexUtil` methods.

8.2.4 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 8.3 gives an example of the intended use of the `RegexUtil` methods.

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

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

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

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

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

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

8.2.5 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 8.3 should support most cases so the `asRegex` method should be used rarely.

Chapter 9

Format String Checker

The Format String Checker detects and 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 9.3 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.

9.1 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.

9.2 Format String Checker annotations

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

The `@Format` qualifier is parameterized with a list of conversion categories that impose restrictions on the format arguments. Conversion categories are explained in more detail in Section 9.2.1. The type qualifier for `"%d %f"` is for example `@Format({INT, FLOAT})`.

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

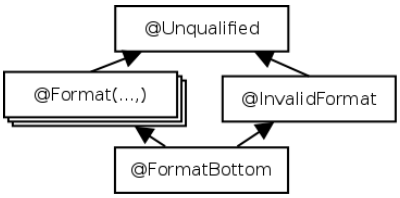

Figure 9.1: The Format String Checker type qualifier hierarchy. The figure does not show the subtyping rules among different `@Format(...)` qualifiers; see Section 9.2.1.

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

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

Figure 9.1 shows all the type qualifiers. The annotations other than `@Format` are only used internally and cannot be written in your code. `@InvalidFormat` indicates an invalid format string — that is, a string that cannot be used as a format string. For example, the type of `"%y"` is `@InvalidFormat String`. `@FormatBottom` is the type of the `null` literal. `@Unqualified` is the default that is applied to strings that are not literals and on which the user has not written a `@Format` annotation.

9.2.1 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 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 9.3.2.) Applicable for conversions c, C.

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

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

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

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

Further, all conversion categories accept `null`.

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

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

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

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

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

NULL is used if no object of any type can be passed as parameter. In this case, the only legal value is `null`. The format string `"%1$f %1$c"`, for example requires that the first format argument be `null`. Passing a value such as `4` or `4.2` would lead to an exception.
CHAR_AND_INT is used if a format argument is restricted by a `CHAR` and a `INT` conversion category (`CHAR` ∩ `INT`).
INT_AND_TIME is used if a format argument is restricted by an `INT` and a `TIME` conversion category (`INT` ∩ `TIME`).

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

Figure 9.2 summarizes the subset relationship among all conversion categories.

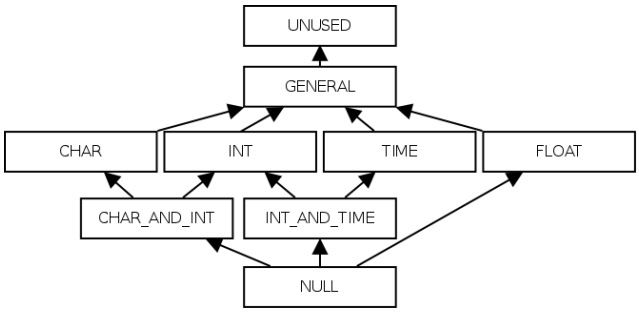

Figure 9.2: The subset relationship among conversion categories.

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

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

The following example shows the subtyping rules in action:

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

f = "%f %d";       // Ok
f = "%s %d";       // OK, %s is weaker than %f
f = "%f";          // warning: last argument is ignored
```

```
f = "%f %d %s";    // error: too many arguments
f = "%d %d";       // error: %d is not weaker than %f

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

9.3 What the Format String Checker checks

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

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

Following are examples for every guarantee:

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

9.3.1 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.

9.3.2 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)
```

9.4 Implicit qualifiers

As described in Section 20.3, the Format String Checker adds implicit qualifiers, reducing the number of annotations that must appear in your code. The checker implicitly adds the `@Format` qualifier with the appropriate conversion categories to any String literal that is a valid format string.

9.5 Testing whether a format string is valid

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

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

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

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

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

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

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

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

Chapter 10

Property File Checker

The Property File Checker ensures that a property file or resource bundle (both of which act like maps from keys to values) is only accessed with valid keys. Accesses without a valid key either return `null` or a default value, which can lead to a `NullPointerException` or hard-to-trace behavior. The Property File Checker (Section 10.1, page 65) ensures that the used keys are found in the corresponding property file or resource bundle.

We also provide two specialized checkers. An Internationalization Checker (Section 10.2, page 66) verifies that code is properly internationalized. A Compiler Message Key Checker (Section 10.3, page 66) 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.

10.1 General Property File Checker

The general Property File Checker ensures that a resource key is located in a specified property file or resource bundle.

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

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

The checker can be invoked by running the following command:

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

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

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

10.2 Internationalization Checker

The Internationalization Checker, or I18n Checker, verifies that your code is properly internationalized. Internationalization is the process of designing software so that it can be adapted to different languages and locales without needing to change the code. Localization is the process of adapting internationalized software to specific languages and locales.

Internationalization is sometimes called i18n (because the word starts with "i", ends with "n", and has 18 characters in between; localization is similarly sometimes abbreviated as l10n).

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

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

The Internationalization Checker verifies these two properties:

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

10.2.1 Internationalization annotations

The Internationalization Checker supports two annotations:

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

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

10.2.2 Running the Internationalization Checker

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

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

You must specify the localization resource, which maps keys to user-visible strings. Like the general Property Key Checker, the Internationalization Checker supports two types of localization resource: `ResourceBundle`s using the `-Abundlenames=`*`resource_name`* option or property files using the `-Apropfiles=`*`prop_file`* option.

10.3 Compiler Message Key Checker

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

The single annotation used by this checker is `@CompilerMessageKey`. The Checker Framework is completely annotated; for example, class `org.checkerframework.framework.source.Result` uses `@CompilerMessageKey` in

methods `failure` and `warning`. For most users of the Checker Framework there will be no need to annotate any `String`s, as the checker looks up all `String` literals and adds annotations as appropriate.

The Compiler Message Key Checker can be invoked by running the following command:

```
javac -processor org.checkerframework.checker.compilermsgs.CompilerMessagesChecker
      -Apropfiles=messages.properties MyFile.java ...
```

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

Chapter 11

Signature Checker for string representations of types

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

Java defines multiple different string representations, 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.

11.1 Signature annotations

Java defines three main formats for the string representation of a type. There is an annotation for each of these representations, plus one more. Figure 11.1 shows how they are related.

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

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

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

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

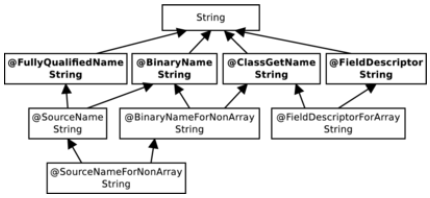

Figure 11.1: Partial type hierarchy for the Signature type system, showing string representations of a Java type. Programmers only need to write the boldfaced qualifiers, in the second row; qualifiers below those are included to improve the internal handling of String literals.

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

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

Here are examples of the supported formats:

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

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

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

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

is

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

11.2 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 12

GUI Effect Checker

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

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

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

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

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

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

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

}
```

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

- effect annotations on methods (Section 12.1),
- class or package annotations controlling the default effect (Section 12.4), and
- *effect-polymorphism*: code that works for both the safe effect and the UI effect (Section 12.5).

12.1 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

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

12.2 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).

12.3 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 ...
```

12.4 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 12.5.2.)
- `@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.

12.5 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;`.

12.5.1 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.

12.5.2 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.

12.5.3 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<Runnable>`.

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.)

12.5.4 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.

12.6 References

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

Chapter 13

Units Checker

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

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

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

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

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

13.1 Units annotations

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

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

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

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

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

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

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

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

13.2 Extending the Units Checker

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

Here is an example of a new unit annotation.

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

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

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

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

13.3 What the Units Checker checks

The Units Checker ensures that unrelated types are not mixed.

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

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

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

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

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

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

13.4 Running the Units Checker

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

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

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

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

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

13.5 Suppressing warnings

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

13.6 References

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

Chapter 14

Linear Checker for preventing aliasing

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

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

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

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

Figure 14.1 gives an example of the Linear Checker's rules.

14.1 Linear annotations

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

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

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

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

This hierarchy makes an assignment like

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

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

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

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

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

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

Figure 14.1: Example of Linear Checker rules.

14.2 Limitations

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

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

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

Chapter 15

IGJ immutability checker

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

IGJ is a Java language extension that helps programmers to avoid mutation errors (unintended side effects). If the IGJ Checker issues no warnings for a given program, then that program will never change objects that should not be changed. This guarantee enables a programmer to detect and prevent mutation-related errors. (See Section 2.3 for caveats to the guarantee.)

To run the IGJ Checker, supply the `-processor org.checkerframework.checker.igj.IGJChecker` command-line option to javac. For examples, see Section 15.7.

15.1 IGJ and Mutability

IGJ [ZPA+07] permits a programmer to express that a particular object should never be modified via any reference (object immutability), or that a reference should never be used to modify its referent (reference immutability). Once a programmer has expressed these facts, an automatic checker analyzes the code to either locate mutability bugs or to guarantee that the code contains no such bugs.

To learn more details of the IGJ language and type system, please see the ESEC/FSE 2007 paper "Object and reference immutability using Java generics" [ZPA+07]. The IGJ Checker supports Annotation IGJ (Section 15.5), which is a slightly different dialect of IGJ than that described in the ESEC/FSE paper.

15.2 IGJ Annotations

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

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

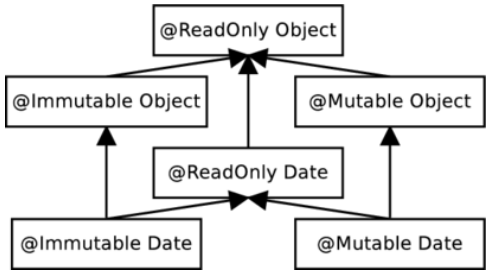

Figure 15.1: Type hierarchy for three of IGJ's type qualifiers.

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

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

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

@AssignsFields is similar to `@Mutable`, but permits only limited mutation — assignment of fields — and is intended for use by constructor helper methods. `@AssignsFields` is assumed to be true of the result of a constructor, so it does not need to be written there.

@I simulates mutability overloading or the template behavior of generics. It can be applied to classes, methods, and parameters. See Section 15.5.3.

For additional details, see [ZPA+07].

15.3 What the IGJ Checker checks

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

15.4 Implicit and default qualifiers

As described in Section 20.3, the IGJ Checker adds implicit qualifiers, reducing the number of annotations that must appear in your code.

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

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

- `@Mutable` for almost all references. This is backward-compatible with Java, since Java permits any reference to be mutated.
- `@Readonly` for local variables. This qualifier may be refined by flow-sensitive local type refinement (see Section 20.4).
- `@Readonly` for type parameter and wildcard bounds. For example,

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

  is defaulted to

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

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

15.5 Annotation IGJ Dialect

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

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

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

15.5.1 Semantic Changes

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

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

  is valid in IGJ, but in Annotation IGJ, only

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

  holds and the other two subtype relations do not hold

  ```
  @ReadOnly Vector<Integer> </:  @ReadOnly Vector<Number>
                            </:  @ReadOnly Vector<Object>
  ```

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

15.5.2 Syntax Changes

- Immutability is specified through type annotations [Ern08] (Section 15.2), not through a combination of generics and annotations. Use of type annotations makes Annotation IGJ backward compatible with Java syntax.
- Templating over Immutability: The annotation `@I(`*`id`*`)` is used to template over immutability. See Section 15.5.3.

15.5.3 Templating Over Immutability: `@I`

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

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

Example:

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

   public @I Date getLastModDate(@ReadOnly FileDescriptor this) { }
}

...
void useFileDescriptor() {
   @Mutable FileDescriptor file =
                    new @Mutable FileDescriptor(...);
   ...
```

```
    @Mutable Data date = file.getLastModDate();

}
```

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

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

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

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

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

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

15.6 Iterators and their abstract state

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

An iterator conceptually has two pieces of state:

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

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

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

15.7 Examples

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

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

The IGJ Checker itself is also annotated with IGJ annotations.

Chapter 16

Javari immutability checker

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

Javari [TE05, QTE08] is a Java language extension that helps programmers to avoid mutation errors that result from unintended side effects. If the Javari Checker issues no warnings for a given program, then that program will never change objects that should not be changed. This guarantee enables a programmer to detect and prevent mutation-related errors. (See Section 2.3 for caveats to the guarantee.) The Javari webpage (`http://types.cs.washington.edu/javari/`) contains papers that explain the Javari language and type system. By contrast to those papers, the Javari Checker uses an annotation-based dialect of the Javari language.

The Javarifier tool infers Javari types for an existing program; see Section 16.2.2.

Also consider the IGJ Checker (Chapter 15). The IGJ type system is more expressive than that of Javari, and the IGJ Checker is a bit more robust. However, IGJ lacks a type inference tool such as Javarifier.

To run the Javari Checker, supply the `-processor org.checkerframework.checker.javari.JavariChecker` command-line option to javac. For examples, see Section 16.5.

16.1 Javari annotations

The following six annotations make up the Javari type system.

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

`@Mutable` indicates a mutable type.

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

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

`@PolyRead` (previously named `@RoMaybe`) specifies polymorphism over mutability; it simulates mutability overloading. It can be applied to methods and parameters. See Section 19.2 and the `@PolyRead` Javadoc for more details.

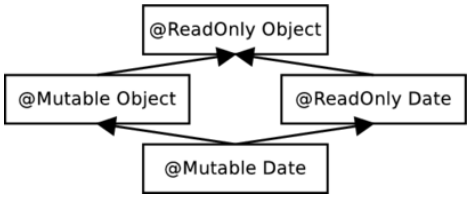

Figure 16.1: Type hierarchy for Javari's ReadOnly type qualifier.

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

16.2 Writing Javari annotations

16.2.1 Implicit qualifiers

As described in Section 20.3, the Javari Checker adds implicit qualifiers, reducing the number of annotations that must appear in your code.

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

16.2.2 Inference of Javari annotations

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

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

16.3 What the Javari Checker checks

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

16.4 Iterators and their abstract state

For an explanation of why the receiver of `Iterator.next()` is annotated as `@ReadOnly`, see Section 15.6.

16.5 Examples

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

```
javac -processor org.checkerframework.checker.javari.JavariChecker tests/javari/*.java
```

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

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

The Javari Checker itself is also annotated with Javari annotations.

Chapter 17

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 23). This includes type introduction rules (implicit annotations, e.g., literals are implicitly considered `@NonNull`) via the `@ImplicitFor` meta-annotation, and other features such as flow-sensitive type qualifier inference (Section 20.4) and qualifier polymorphism (Section 19.2).

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

If you need typestate analysis, then you can extend a typestate checker, much as you would extend the Subtyping Checker if you do not need typestate analysis. For more details (including a definition of "typestate"), see Chapter 18.1. See Section 25.6.2 for a simpler alternative.

For type systems that require special checks (e.g., warning about dereferences of possibly-null values), you will need to write code and extend the framework as discussed in Chapter 23.

17.1 Using the Subtyping Checker

The Subtyping Checker is used in the same way as other checkers (using the `-processor org.checkerframework.common.subtyping.SubtypingChecker` option; see Chapter 2), except that it requires an additional annotation processor argument via the standard "`-A`" switch:

- `-Aquals`: this option specifies a comma-no-space-separated list of the fully-qualified class names of the annotations used as qualifiers in the custom type system. For example,
  ```
  javac -processor org.checkerframework.common.subtyping.SubtypingChecker
        -Aquals=myproject.qual.MyQual,myproject.qual.OtherQual MyFile.java ...
  ```
  It serves the same purpose as the `@TypeQualifiers` annotation used by other checkers (see section 23.6).
  The annotations listed in `-Aquals` must be accessible to the compiler during compilation in the classpath. In other words, they must already be compiled (and, typically, be on the javac bootclasspath) before you run the Subtyping Checker with `javac`. It is not sufficient to supply their source files on the command line.

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

17.2 Subtyping Checker example

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

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

   ```
   package myqual;

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

   import org.checkerframework.framework.qual.*;

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

   Don't forget to compile this class:

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

   The resulting `.class` file should either be on your classpath, or on the processor path (set via the `-processorpath` command-line option to javac).
2. Write `@Encrypted` annotations in your program (`YourProgram.java`):

   ```
   import myqual.Encrypted;

   ...

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

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

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

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

You may also need to add `@SuppressWarnings` annotations to the `encrypt` and `decrypt` methods. Analyzing them is beyond the capability of any realistic type system.

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

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

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

Chapter 18

Third-party checkers

The Checker Framework has been used to build other checkers that are not distributed together with the framework. This chapter mentions just a few of them.

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

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

18.1 Typestate checkers

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

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

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

18.1.1 Comparison to flow-sensitive type refinement

The Checker Framework's flow-sensitive type refinement (Section 20.4) implements a form of typestate analysis. For example, after code that tests a variable against null, the Nullness Checker (Chapter 3) treats the variable's type as `@NonNull` *`T`*, for some *`T`*.

For many type systems, flow-sensitive type refinement is sufficient. But sometimes, you need full typestate analysis. This section compares the two. (Unused variables (Section 20.6) also have similarities with typestate analysis and can occasionally substitute for it. For brevity, this discussion omits them.)

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

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

type — for flow-sensitive type refinement to be effective, `@Closed` would need to be a child of `@Open` in the qualifier hierarchy (and `close()` would need to be treated specially by the checker).

18.2 Units and dimensions checker

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

Unlike the Units Checker that is distributed with the Checker Framework (see Section 13), 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 Type Annotations compiler.

18.3 Thread locality checker

Loci, a checker for thread locality, is available at `http://www.it.uu.se/research/upmarc/loci/`. Developer resources are available at the project page `http://java.net/projects/loci/`.

18.4 Safety-Critical Java checker

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

18.5 Generic Universe Types checker

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

18.6 EnerJ checker

A checker for EnerJ, an extension to Java that exposes hardware faults in a safe, principled manner to save energy with only slight sacrifices to the quality of service, is available from `http://sampa.cs.washington.edu/research/approximation/enerj.html`.

18.7 CheckLT taint checker

CheckLT uses taint tracking to detect illegal information flows, such as unsanitized data that could result in a SQL injection attack. CheckLT is available from `http://checklt.github.io/`.

18.8 JavaUI GUI threading checker

Eclipse and other GUI toolkits require all access to the UI to occur from the UI event loop thread. If a different thread accesses a UI element, the program crashes. JavaUI is an effect system that ensures that you program does not suffer such access errors.

The implementation is available at `https://github.com/csgordon/javaui`. A fork that fixes some compilation errors appears at `https://github.com/Overruler/javaui`. You can also read a technical paper about the effect system: "JavaUI: Effects for Controlling UI Object Access" is available at `http://homes.cs.washington.edu/~mernst/pubs/gui-thread-ecoop2013-abstract.html`.

Chapter 19

Generics and polymorphism

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

19.1 Generics (parametric polymorphism or type polymorphism)

The Checker Framework fully supports type-qualified Java generic types (also known in the research literature as "parametric polymorphism"). When instantiating a generic type, clients supply the qualifier along with the type argument, as in `List<@NonNull String>`.

19.1.1 Raw types

Before running any pluggable type-checker, we recommend that you eliminate raw types from your code (e.g., your code should use `List<...>` as opposed to `List`). Your code should compile without warnings when using the standard Java compiler and the `-Xlint:unchecked -Xlint:rawtypes` command-line options. Using generics helps prevent type errors just as using a pluggable type-checker does, and makes the Checker Framework's warnings easier to understand.

If your code uses raw types, then the Checker Framework will do its best to infer the Java type parameters and the type qualifiers. If it infers imprecise types that lead to type-checking warnings elsewhere, then you have two options. You can convert the raw types such as `List` to parameterized types such as `List<String>`, or you can supply the `-AignoreRawTypeArguments` command-line option. That option causes the Checker Framework to ignore all subtype tests for type arguments that were inferred for a raw type.

19.1.2 Restricting instantiation of a generic class

When you define a generic class in Java, the `extends` or `super` clause of the generic type parameter restricts how the class may be instantiated. For example, given the definition `class G<T extends Number> {...}`, a client can instantiate it as `G<Integer>` but not `G<Date>`. Similarly, type qualifiers on the generic type parameters can restrict on how the class may be instantiated. For example, a generic list class might indicate that it can hold only non-null values. Similarly, a generic map class might indicate it requires an immutable key type, but that it supports both nullable and non-null value types.

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

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

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

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

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

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

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

19.1.3 A qualifier on a type parameter declaration is like two bounds

A type annotation on a generic type parameter declaration can be thought of as syntactic sugar for writing the annotation on both the `extends` and the `super` clauses of the declaration.

Suppose that a type parameter declaration or a wildcard is annotated. Then the annotation applies to all bounds that have no explicit annotation. In other words, the annotation applies to both the upper and lower bounds, except that any explicitly-written bound annotation takes precedence.

Type parameter declaration annotation examples

For example, the following pairs of declarations have the same meaning (except that the second declaration in each pair is not legal syntax).

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

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

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

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

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

Type parameter declaration annotation rationale

It is desirable to be able to constrain both the upper and the lower bound of a type, as in

```
class MyClass<T extends @C MyUpperBound super @D void> { ... }
```

However, doing so is not possible due to two limitations of Java's syntax. First, it is illegal to specify both the upper and the lower bound of a type parameter or wildcard. Second, it is impossible to specify a type annotation for a lower bound without also specifying a type (use of `void` is illegal).

Thus, when you wish to specify both bounds, you write one of them explicitly, and you write the other one in front of the type variable name or `?`. When you wish to specify two identical bounds, you write a single annotation in front of the type variable name or `?`.

19.1.4 Examples of qualifiers on a type parameter

Recall that `@Nullable X` is a supertype of `@NonNull X`, for any *X*. We can see from Section 19.1.3 that almost all of the following types mean different things:

```
class MyList1<@Nullable T> { ... }
class MyList2<@NonNull T> { ... }
class MyList3<T extends @Nullable Object> { ... }
class MyList4<T extends @NonNull Object> { ... } // same as MyList2
```

`MyList1` must be instantiated with a nullable type. The implementation must be able to consume (store) a null value and produce (retrieve) a null value.

`MyList2` and `MyList4` must be instantiated with non-null type. The implementation 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 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`, `MyList3`, and `MyList4` is by comparing what expressions are legal in the implementation of the list — that is, what expressions may appear in the ellipsis, such as inside a method's body. Suppose each class has, in the ellipsis, these declarations:

```
T t;
@Nullable T nble;      // Section "Type annotations on a use of a generic type variable", below,
@NonNull T nn;         // further explains the meaning of "@Nullable T" and "@NonNull T".
void add(T arg) { ... }
T get(int i) { ... }
```

Then the following expressions would be legal, inside a given implementation. (Compilable source code appears as file `checker-framework/checker/tests/nullness/generics/GenericsExample.java`.)

	MyList1	MyList2	MyList3	MyList4
t = null;	OK	error	error	error
t = nble;	OK	error	error	error
nble = null;	OK	OK	OK	OK
nn = null;	error	error	error	error
t = this.get(0);	OK	OK	OK	OK
nble = this.get(0);	OK	OK	OK	OK
nn = this.get(0);	error	OK	error	OK
this.add(t);	OK	OK	OK	OK
this.add(nble);	OK	error	error	error
this.add(nn);	OK	OK	OK	OK

The differences are more significant when the qualifier hierarchy is more complicated than just `@Nullable` and `@NonNull`.

19.1.5 Type annotations on a use of a generic type variable

A type annotation on a use of a generic type variable overrides/ignores any type qualifier (in the same type hierarchy) on the corresponding actual type argument. For example, suppose that `T` is a formal type parameter. Then using `@Nullable T` within the scope of `T` applies the type qualifier `@Nullable` to the (unqualified) Java type of `T`. This feature is only rarely used.

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

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

The type annotation does not restrict how `MyClass2` may be instantiated. In other words, both `MyClass2<@NonNull String>` and `MyClass2<@Nullable String>` are legal, and in both cases `@Nullable T` means `@Nullable String`. In `MyClass2<@Interned String>`, `@Nullable T` means `@Nullable @Interned String`.

19.1.6 Covariant type parameters

Java types are *invariant* in their type parameter. This means that `A<X>` is a subtype of `B<Y>` only if `X` is identical to `Y`. For example, `ArrayList<Number>` is a subtype of `List<Number>`, but neither `ArrayList<Integer>` nor `List<Integer>` is a subtype of `List<Number>`. (If they were, there would be a type hole in the Java type system.) For the same reason, type parameter annotations are treated invariantly. For example, `List<@Nullable String>` is not a subtype of `List<String>`.

When a type parameter is used in a read-only way — that is, when values of that type are read but are never assigned — then it is safe for the type to be *covariant* in the type parameter. Use the `@Covariant` annotation to indicate this. When a type parameter is covariant, two instantiations of the class with different type arguments have the same subtyping relationship as the type arguments do.

For example, consider `Iterator`. Its elements can be read but not written, so `Iterator<@Nullable String>` can be a subtype of `Iterator<String>` without introducing a hole in the type system. Therefore, its type parameter is annotated with `@Covariant`. The first type parameter of `Map.Entry` is also covariant. Another example would be the type parameter of a hypothetical class `ImmutableList`.

The `@Covariant` annotation is trusted but not checked. If you incorrectly specify as covariant a type parameter that that can be written (say, the class performs a `set` operation or some other mutation on an object of that type), then you have created an unsoundness in the type system. For example, it would be incorrect to annotate the type parameter of `ListIterator` as covariant, because `ListIterator` supports a `set` operation.

19.1.7 Method type argument inference and type qualifiers

Sometimes method type argument inference does not interact well with type qualifiers. In such situations, you might need to provide explicit method type arguments, for which the syntax is as follows:

```
Collections.</*@MyTypeAnnotation*/ Object>sort(l, c);
```

This uses Java's existing syntax for specifying a method call's type arguments.

19.2 Qualifier polymorphism

The Checker Framework also supports type *qualifier* polymorphism for methods, which permits a single method to have multiple different qualified type signatures. This is similar to Java's generics, but is used in situations where you cannot use Java generics. If you can use generics, you typically do not need to use a polymorphic qualifier such as `@PolyNull`.

To *use* a polymorphic qualifier, just write it on a type. For example, you can write `@PolyNull` anywhere in a method that you would write `@NonNull` or `@Nullable`. A polymorphic qualifier can be used on a method signature or body. It may not be used on a class or field.

A method written using a polymorphic qualifier conceptually has multiple versions, somewhat like a template in C++ or the generics feature of Java. In each version, each instance of the polymorphic qualifier has been replaced by the same other qualifier from the hierarchy. See the examples below in Section 19.2.1.

The method body must type-check with all signatures. A method call is type-correct if it type-checks under any one of the signatures. If a call matches multiple signatures, then the compiler uses the most specific matching signature for the purpose of type-checking. This is the same as Java's rule for resolving overloaded methods.

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

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

See Section 19.2.5 for a way you can sometimes avoid defining a new polymorphic qualifier.

19.2.1 Examples of using polymorphic qualifiers

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

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

This is like writing:

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

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

As another example, consider

```
// Returns null if either argument is null.
@PolyNull T max(@PolyNull T x, @PolyNull T y);
```

which is like writing

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

At a call site, the most specific applicable signature is selected.

Another way of thinking about which one of the two `max` variants is selected is that the nullness annotations of (the declared types of) both arguments are *unified* to a type that is a supertype of both, also known as the *least upper bound* or lub. If both arguments are `@NonNull`, their unification (lub) is `@NonNull`, and the method return type is `@NonNull`. But if even one of the arguments is `@Nullable`, then the unification (lub) is `@Nullable`, and so is the return type.

19.2.2 Relationship to subtyping and generics

Qualifier polymorphism has the same purpose and plays the same role as Java's generics. If a method is written using generics, it usually does not need qualifier polymorphism. If you have legacy code that is not written generically, and you cannot change it to use generics, then you can use qualifier polymorphism to achieve a similar effect, with respect to type qualifiers only. The base Java types are still treated non-generically.

Why not use ordinary subtyping to handle qualifier polymorphism? Ordinarily, when you want a method to work on multiple types, you can just use Java's subtyping. For example, the `equals` method is declared to take an `Object` as its first formal parameter, but it can be called on a `String` or a `Date` because those are subtypes of `Object`.

In most cases, the same subtyping mechanism works with type qualifiers. `String` is a supertype of `@Interned String`, so a method `toUpperCase` that is declared to take a `String` parameter can also be called on a `@Interned String` argument.

You use qualifier polymorphism in the same cases when you would use Java's generics. (If you can use Java's generics, then that is often better and you don't also need to use qualifier polymorphism.) One example is when you want a method to operate on collections with different types of elements. Another example is when you want two different formal parameters to be of the same type, without constraining them to be one specific type.

19.2.3 Using multiple polymorphic qualifiers in a method signature

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

For example, there is no point to writing

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

which expands to

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

This is no different (in terms of which calls to the method will type-check) than writing just

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

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

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

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

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

19.2.4 Using a single polymorphic qualifier on an element type

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

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

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

If `@PolyNull` were replaced with either `@Nullable` or `@NonNull`, then one of these safe client calls would be rejected:

```
@Nullable Object[] a1;
@NonNull Object[] a2;

indexOf(a1, someObject);
indexOf(a2, someObject);
```

Of course, it would be better style to use a generic method, as in either of these signatures:

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

The examples in this section use arrays, but analogous collection examples exist.

These examples show that use of a single polymorphic qualifier may be necessary in legacy code, but can often be avoided by use of better code style.

19.2.5 The `@PolyAll` qualifier applies to every type system

Ordinarily, you have to create a new polymorphic type qualifier for each type system you write. This can be tedious. More seriously, it can lead to an explosion in the number of type annotations, if some method is qualifier-polymorphic over multiple type qualifier hierarchies.

For example, a method that only performs == on array elements will work no matter what the array's element types are:

```
/** Searches for the first occurrence of the given element in the array,
 *  testing for equality using == (not the equals method). */
public static int indexOfEq(@PolyAll Object[] a, @Nullable Object elt) {
  for (int i=0; i<a.length; i++)
    if (elt == a[i])
      return i;
  return -1;
}
```

The `@PolyAll` qualifier takes an optional argument so that you can specify multiple, independent polymorphic type qualifiers. For example, the method also works no matter what the type argument on the second argument is. This signature is overly restrictive:

```
/** Returns true if the arrays are elementwise equal,
 *  testing for equality using == (not the equals method). */
public static int eltwiseEqualUsingEq(@PolyAll Object[] a, @PolyAll Object elt) {
  for (int i=0; i<a.length; i++)
    if (elt != a[i])
      return false;
  return true;
}
```

That signature requires the element type annotation to be identical for the two arguments. For example, it forbids this invocation:

```
  @Mutable Object[] x;
@Immutable Object   y;
... indexOf(x, y) ...
```

A better signature lets the two arrays' element types vary independently:

```
public static int eltwiseEqualUsingEq(@PolyAll(1) Object[] a, @PolyAll(2) Object elt)
```

Note that in this case, the `@Nullable` annotation on `elt`'s type is no longer necessary, since it is subsumed by `@PolyAll`.

The `@PolyAll` annotation applies to every type qualifier hierarchy for which no explicit qualifier is written. For example, a declaration like `@PolyAll @NonNull Object elt` is polymorphic over every type system *except* the nullness type system, for which the type is fixed at `@NonNull`. That would be the proper declaration for `elt` if the body had used `elt.equals(a[i])` instead of `elt == a[i]`.

Chapter 20

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.

20.1 Invariant array types

Java's type system is unsound with respect to arrays. That is, the Java type-checker approves code that is unsafe and will cause a run-time crash. Technically, the problem is that Java has "covariant array types", such as treating `String[]` as a subtype of `Object[]`. Consider the following example:

```
String[] strings = new String[] {"hello"};
Object[] objects = strings;
objects[0] = new Object();
String myString = strs[0];
```

The above code puts an `Object` in the array `strings` and thence in `myString`, even though `myString = new Object()` should be, and is, rejected by the Java type system. Java prevents corruption of the JVM by doing a costly run-time check at every array assignment; nonetheless, it is undesirable to learn about a type error only via a run-time crash rather than at compile time.

When you pass the `-AinvariantArrays` command-line option, the Checker Framework is stricter than Java, in the sense that it treats arrays invariantly rather than covariantly. This means that a type system built upon the Checker Framework is sound: you get a compile-time guarantee without the need for any run-time checks. But it also means that the Checker Framework rejects code that is similar to what Java unsoundly accepts. The guarantee and the compile-time checks are about your extended type system. The Checker Framework does not reject the example code above, which contains no type annotations.

Java's covariant array typing is sound if the array is used in a read-only fashion: that is, if the array's elements are accessed but the array is not modified. However, fact about read-only usage is not built into any of the type-checkers except those that are specifically about immutability: IGJ (see Chapter 15, page 79) and Javari (see Chapter 16, page 83). Therefore, when using other type systems along with `-AinvariantArrays`, you will need to suppress any warnings that are false positives because the array is treated in a read-only way.

20.2 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 8, page 56), the string `"hello"` is

given type `@Regex String` because it is a legal regular expression (whether it is meant to be used as one or not) and the string `"(foo"` is given the type `@Unqualified String` because it is not a legal regular expression.

Array constructors work differently. When you create an array with the array constructor syntax, such as the right-hand side of this assignment:

```
String[] myStrings = {"hello"};
```

then the expression does not get the most precise possible type, because doing so could cause inconvenience. Rather, its type is determined by the context in which it is used: the left-hand side if it is in an assignment, the declared formal parameter type if it is in a method call, etc.

In particular, if the expression `{"hello"}` were given the type `@Regex String[]`, then the assignment would be illegal! But the Checker Framework gives the type `String[]` based on the assignment context, so the code type-checks.

If you prefer a specific type for a constructed array, you can indicate that either in the context (change the declaration of `myStrings`) or in a `new` construct (change the expression to `new @Regex String[] {"hello"}`).

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

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

Most of this section is applicable only to source code that is being checked by a checker.

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

1. If a type qualifier is present in the source code, that qualifier is used.
2. The type system adds implicit qualifiers. This happens whether or not the programmer has written an explicit type qualifier.
   Here are some examples of implicit qualifiers:
   - In the Nullness type system, `enum` values, string literals, and method receivers are always non-null.
   - In the Interning type system, string literals and `enum` values are always interned.

   If the type has an implicit qualifier, then it is an error to write an explicit qualifier that is equal to (redundant with) or a supertype of (weaker than) the implicit qualifier. A programmer may strengthen (write a subtype of) an implicit qualifier, however.
   Implicit qualifiers arise from two sources:

   built-in Implicit qualifiers can be built into a type system (Section 23.4), in which case the type system's documentation explains all of the type system's implicit qualifiers. Both of the above examples are built into the Nullness type system.

   programmer-declared A programmer may introduce an implicit annotation on each use of class *C* by writing a qualifier on the declaration of class *C*. If `MyClass` is declared as `class @MyAnno MyClass {...}`, then each occurrence of `MyClass` in the source code is treated as if it were `@MyAnno MyClass`.
3. If there is no explicit or implicit qualifier on a type, then a default qualifier may be applied; see Section 20.3.1.

   At this point (after step 3), every type has a qualifier.
4. The type system may refine a qualified type on a local variable — that is, treat it as a subtype of how it was declared or defaulted. This refinement is always sound and has the effect of eliminating false positive error messages. See Section 20.4.

20.3.1 Default qualifier for unannotated types

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

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

- Use the innermost user-written `@DefaultQualifier`, as explained in this section.
- Use the default specified by the type system designer (Section 23.3.3).
- Use `@Unqualified`, which the framework inserts to avoid ambiguity and simplify the programming interface for type system designers. Users do not have to worry about this detail, but type system implementers can rely on the fact that some qualifier is present.

The end-user programmer specifies a default qualifier by writing the `@DefaultQualifier` annotation on a package, class, method, or variable declaration. The argument to `@DefaultQualifier` is the `String` name of an annotation. It may be a short name like `"NonNull"`, if an appropriate import statement exists. Otherwise, it should be fully-qualified, like `"org.checkerframework.checker.nullness.qual.NonNull"`. The optional second argument indicates where the default applies. If the second argument is omitted, the specified annotation is the default in all locations. See the Javadoc of `DefaultQualifier` for details.

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

```
import org.checkerframework.framework.qual.*;        // for DefaultQualifier[s]
import org.checkerframework.checker.nullness.qual.NonNull;

@DefaultQualifier(NonNull.class)
class MyClass {

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

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

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

```
@DefaultQualifiers({
  @DefaultQualifier(NonNull.class),
  @DefaultQualifier(Mutable.class)
})
```

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

Recall that an annotation on a class definition indicates an implicit qualifier (Section 20.3) that can only be strengthened, not weakened. This can lead to unexpected results if the default qualifier applies to a class definition.

Thus, you may want to put explicit qualifiers on class declarations (which prevents the default from taking effect), or exclude class declarations from defaulting.

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

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

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

20.3.2 Defaulting rules and CLIMB-to-top

Each type system defines a default qualifier. For example, the default qualifier for the Nullness Checker is `@NonNull`. That means that when a user writes a type such as `Date`, the Nullness Checker interprets it as `@NonNull Date`.

We recommend that the type system apply that default qualifier to most but not all types. In particular, we recommend the CLIMB-to-top rule. This rule states that the *top* qualifier in the hierarchy is applied to the CLIMB locations: **C**asts, **L**ocals, **I**nstanceof, and i**M**plicit **B**ounds. For example, when the user writes a type such as `Date` in such a location, the Nullness Checker interprets it as `@Nullable Date` (because `@Nullable` is the top qualifier in the hierarchy, see Figure 3.1).

The rest of this section explains the rationale and implementation of CLIMB-to-top.

Casts, local variables (including resource variables in the try-with-resources construct, variables in for statements, etc.), and instanceof should be defaulted to top because they are the locations to which type refinement (Section 20.4) is applied. If they start as the top type, then the Checker Framework chooses the best (most general) possible type for them. As a result, a programmer rarely writes an explicit annotation on any of those locations.

Implicit upper bounds are defaulted to top to allow them to be instantiated in any way. If a user declared `class C<T> { ... }`, then we assume that the user intended to allow any instantiation of the class, and the declaration is interpreted as `class C<T extends @Nullable Object> { ... }` rather than as `class C<T extends @NonNull Object> { ... }`. The latter would forbid instantiations such as `C<@Nullable String>`, or would require rewriting of code. On the other hand, if a user writes an explicit bound such as `class C<T extends D> { ... }`, then the user intends some restriction on instantiation and can write a qualifier on the upper bound as desired.

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 tracks both of them separately and allows the programmer to specify both by using a declaration on a wildcard or type variable name; see Section 19.1.3.

Here is how the CLIMB-to-top rule is expressed for the Nullness Checker:

```
@DefaultQualifierInHierarchy
public @interface NonNull { }

@DefaultFor({ DefaultLocation.LOCAL_VARIABLE, DefaultLocation.RESOURCE_VARIABLE,
    DefaultLocation.IMPLICIT_UPPER_BOUNDS })
public @interface Nullable { }
```

Note that `DefaultLocation.LOCAL_VARIABLE` includes casts and instanceof.

A type system designer does not have to use the CLIMB-to-top rule. In addition, a user may choose a different rule for defaults using the `@DefaultQualifier` annotation; see Section 20.3.1.

20.3.3 Inherited defaults

In certain situations, it would be convenient for an annotation on a superclass member to be automatically inherited by subclasses that override it. This feature would reduce both annotation effort and program comprehensibility. In general,

a program is read more often than it is edited/annotated, so the Checker Framework does not currently support this feature. Here are more detailed justifications:

- Currently, a user can determine the annotation on a parameter or return value by looking at a single file. If annotations could be inherited from supertypes, then a user would have to examine all supertypes to understand the meaning of an unannotated type in a given file.
- Different annotations might be inherited from a supertype and an interface, or from two interfaces. Presumably, the subtype's annotations would be stronger than either (the greatest lower bound in the type system), or an error would be thrown if no such annotations existed.

If these issues can be resolved, then the feature may be added in the future. Or, it may be added optionally, and each type-checker implementation can enable it if desired.

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

In order to reduce your burden of annotating types in your program, the checkers soundly treat certain variables and expressions as having a subtype of their declared or defaulted (Section 20.3.1) type. This functionality eliminates some false positive warnings, but it never introduces unsoundness nor causes an error to be missed.

As an example, suppose you write

```
@Nullable String myVar;
...
myVar = "hello";
myVar.hashCode();
```

The Nullness Checker issues a warning whenever a method such as `hashCode()` is called on a possibly-null value, which may result in a null pointer exception. The Nullness Checker need not issue a warning in this case. In particular, after the assignment, type-checker treats `myVar` as having type `@NonNull String`, which is a subtype of its declared type.

Here is another example:

```
@Nullable String myVar;
...
if (myVar != null) {
  myVar.hashCode();
}
```

Once again, the Nullness Checker need not issue a warning. Within the body of the `if` test, the type of `myVar` is `@NonNull String`, even though `myVar` is declared as `@Nullable String`.

Array element types and generic arguments are never changed by type refinement. Changing these components of a type never yields a subtype of the declared type. For example, `List<Number>` is *not* a subtype of `List<Object>`. Similarly, the Checker Framework does not treat `Number[]` as a subtype of `Object[]`; see Section 20.1 for why.

By default, all checkers, including new checkers that you write, automatically incorporate type refinement. Most of the time, users don't have to think about, and may not even notice, type refinement. The checkers simply do the right thing even when a programmer omits an annotation on a local variable, or when a programmer writes an unnecessarily general type in a declaration.

The functionality has a variety of names: automatic type refinement, flow-sensitive type qualifier inference, local type inference, and sometimes just "flow".

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

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

- starting at the time that it is either assigned a non-null value or checked against null (e.g., via an assertion, `if` statement, or being dereferenced)
- until it might be re-assigned (e.g., via an assignment that might affect this variable, or via a method call that might affect this variable).

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

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

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

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

If you find examples where you think a value should be inferred to have (or not have) a given annotation, but the checker does not do so, please submit a bug report (see Section 26.2) that includes a small piece of Java code that reproduces the problem.

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

Type inference is never performed for method parameters of non-private methods, nor for non-private fields. More generally, the inferred information is never written to the `.class` file as user-written annotations are. If the checker did inference in externally-visible locations and wrote it to the `.class` file, then the resulting `.class` file would be different depending on whether an annotation processor had been run or not. It is a design goal that the same annotations appear in the `.class` file regardless of whether the class is compiled with or without the checker, and this requires that any public signature be fully annotated by the user rather than inferred.

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

20.4.1 Run-time tests and type refinement

Some type systems support a run-time test that the Checker Framework can use to refine types within the scope of a conditional such as `if`, after an `assert` statement, etc.

Whether a type system supports such a run-time test depends on whether the type system is computing properties of data itself, or properties of provenance (the source of the data). An example of a property about data is whether a string is a regular expression. An example of a property about provenance is units of measure: there is no way to look at the representation of a number and determine whether it is intended to represent kilometers or miles.

Type systems that support a run-time test are:

- Nullness Checker for null pointer errors (see Chapter 3, page 22)
- Map Key Checker to track which values are keys in a map (see Chapter 3.9, page 42)
- Regex Checker to prevent use of syntactically invalid regular expressions (see Chapter 8, page 56)

- Format String Checker to ensure that format strings have the right number and type of `%` directives (see Chapter 9, page 59).

Type systems that do not 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 4, page 44)
- Property File Checker to ensure that valid keys are used for property files and resource bundles (see Chapter 10, page 65)
- Internationalization Checker to ensure that code is properly internationalized (see Chapter 10.2, page 66)
- Signature String Checker to ensure that the string representation of a type is properly used, for example in `Class.forName` (see Chapter 11, page 68).

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

- Initialization Checker to ensure all fields are set in the constructor (see Chapter 3.8, page 32)
- Lock Checker for concurrency and lock errors (see Chapter 5, page 47)
- Fake Enum Checker to allow type-safe fake enum patterns (see Chapter 6, page 51)
- Tainting Checker for trust and security errors (see Chapter 7, page 54)
- Units Checker to ensure operations are performed on correct units of measurement (see Chapter 13, page 74)
- Linear Checker to control aliasing and prevent re-use (see Chapter 14, page 77)
- IGJ Checker for mutation errors (incorrect side effects), based on the IGJ type system (see Chapter 15, page 79)
- Javari Checker for mutation errors (incorrect side effects), based on the Javari type system (see Chapter 16, page 83)
- Subtyping Checker for customized checking without writing any code (see Chapter 17, page 85)

20.4.2 Fields and flow-sensitive analysis

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

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

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

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

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

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

  ```
  class DBObject {
    @Nullable Date updatedAt;

    void persistData() {
      ... // write to disk or other non-volatile memory
      updatedAt = null;
    }

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

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

Here the call to `persistData()` invalidates the inferred non-null type of `updatedAt`.
When methods do not modify any object state or have any identity side effects (e.g., `log()` method here), you can annotate these methods as `SideEffectFree` or `Pure` (see Section 20.4.3). When a method is annotated as `SideEffectFree`, the flow analyzer carries the inferred types across the method invocation boundary.

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

As described above, a checker can use a refined type for an expression from the time when the checker infers that the value has that refined type, until the checker can no longer support that inference.

The refined type begins at a test (such as `if (myvar != null) ...`) or an assignment. If the assignment occurs within a method body, write a postcondition annotation such as `@EnsuresNonNull`.

The refined type ends at an assignment or possible assignment. Any method call has the potential to side-effect any field, so calling a method typically causes the checker to discard its knowledge of the refined type. This is undesirable if the method doesn't actually re-assign the field.

There are three annotations, collectively called purity annotations, that you can use to help express what effects a method call does not have. Usually, you only need to use `@SideEffectFree`.

`@SideEffectFree` indicates that the method has no (visible) side effects.
`@Deterministic` indicates that if the method is called multiple times with the same arguments, then it returns the same result.
`@Pure` indicates that the method is both @SideEffectFree and @Deterministic.

The Javadoc of the annotations describes their semantics and how they are checked. This manual section gives examples and supplementary information.

For example, consider the following declarations and uses:

```
@Nullable Object myField;

int computeValue() { ... }

...
if (myField != null) {
  int result = computeValue();
  myField.toString();
}
```

Ordinarily, the Nullness Checker would issue a warning regarding the `toString()` call, because the receiver `myField` might be `null`, according to the `@Nullable` annotation on the declaration of `myField`. Even though the code checked the value of `myField`, the call to `computeValue` might have re-set it to null. If you change the declaration of `computeValue` to

```
@SideEffectFree int computeValue() { ... }
```

then the Nullness Checker issues no warnings, because it can reason that the second occurrence of `myField` has the same (non-null) value as the one in the test.

As a more complex example, consider the following declaration and uses:

```
@Nullable Object getField(Object arg) { ... }

...
if (x.getField(y) != null) {
  x.getField(y).toString();
}
```

Ordinarily, the Nullness Checker would issue a warning regarding the `toString()` call, because the receiver `x.getField(y)` might be `null`, according to the `@Nullable` annotation in the declaration of `getField`. If you change the declaration of `getField` to

```
@Pure @Nullable Object getField(Object arg) { ... }
```

then the Nullness Checker issues no warnings, because it can reason that the two invocations `x.getField(y)` have the same value, and therefore that `x.getField(y)` is non-null within the then branch of the if statement.

If a method is side-effect-free or pure, then it would be legal to annotate its receiver and every parameter as `@ReadOnly`, in the IGJ (Chapter 15) or Javari (Chapter 16) type systems. The reverse is not true, because the method might side-effect a global variable. (Also, for the case of `@Pure`, the method might not be deterministic.)

If you supply the command-line option `-AsuggestPureMethods`, then the Checker Framework will suggest methods that can be marked as `@SideEffectFree`, `@Deterministic`, or `@Pure`.

Currently, purity annotations are trusted. Purity annotations on called methods affect type-checking of client code. However, you can make a mistake by writing `@SideEffectFree` on the declaration of a method that is not actually side-effect-free or by writing `@Deterministic` on the declaration of a method that is not actually deterministic. To enable checking of the annotations, supply the command-line option `-AenablePurity`. It is not enabled by default because of a high false positive rate. In the future, after a new purity-checking analysis is implemented, the Checker Framework will default to checking purity annotations.

It can be tedious to annotate library methods with purity annotations such as `@SideEffectFree`. If you supply the command-line option `-AassumeSideEffectFree`, then the Checker Framework will unsoundly assume that every called method is side-effect-free. This can make flow-sensitive type refinement much more effective, since method calls will not cause the analysis to discard information that it has learned. However, this option can mask real errors. It is most appropriate when you are starting out annotating a project, or if you are using the Checker Framework to find some bugs but not to give a guarantee that no more errors exist of the given type.

A common error is:

```
MyClass.java:1465: error: int hashCode() in MyClass cannot override int hashCode(Object this) in java.lang
    public int hashCode() {
               ^
  found   : []
  required: [SIDE_EFFECT_FREE, DETERMINISTIC]
```

The reason for the error is that the `Object` class is annotated as:

```
class Object {
  ...
  @Pure int hashCode() { ... }
}
```

(where `@Pure` means both `@SideEffectFree` and `@Deterministic`). Every overriding definition, including those in your program, must use be at least as strong a specification; in particular, every overriding definition must be annotated as `@Pure`.

You can fix the definition by adding `@Pure` to your method definition. Alternately, you can suppress the warning. You can suppress each such warning individually using `@SuppressWarnings("purity.invalid.overriding")`, or you can use the `-AsuppressWarnings=purity.invalid.overriding` command-line argument to suppress all such warnings. In the future, the Checker Framework will support inheriting annotations from superclass definitions.

20.4.4 Assertions

If your code contains an `assert` statement, then your code could behave in two different ways at run time, depending on whether you compile with or without the `-ea` command-line option to javac.

By default, the Checker Framework outputs warnings about any error that could happen at run time, whether assertions are enabled or disabled.

If you supply the `-AassumeAssertionsAreEnabled` command-line option, then the Checker Framework assumes assertions are enabled. If you supply the `-AassumeAssertionsAreEnabled` command-line option, then the Checker Framework assumes assertions are disabled. You may not supply both command-line options.

20.5 Writing Java expressions as annotation arguments

Sometimes, it is necessary to write a Java expression as the argument to an annotation. The annotations that take a Java expression as an argument are:

- `@RequiresQualifier`
- `@EnsuresQualifier`
- `@EnsuresQualifierIf`
- `@RequiresNonNull`
- `@EnsuresNonNull`
- `@EnsuresNonNullIf`

The expression is a subset of legal Java expressions:

- the receiver object, `this`.
- the receiver object as seen from the superclass, `super`. This can be used to refer to fields shadowed in the subclass (although shadowing fields is discouraged).
- a formal parameter. Write `#` followed by the **one-based** parameter index. For example: `#1`, `#3`. It is not permitted to write `#0` to refer to the receiver object; use `this` instead.
- a static variable. Write the class name and the variable, as in `System.out`.
- a field of any expression. For example: `next`, `this.next`, `#1.next`.
- an array access. For example: `this.myArray[i]`, `vals[#1]`.
- literals: string, integer, long, null.
- a method invocation on any expression This even works for overloaded methods and methods with type parameters. For example: `m1(x, y.z, #2)`, `a.m2("hello")`.

You may optionally omit a leading "`this.`", just as in Java. Thus, `this.next` and `next` are equivalent.

One unusual feature is that the method call is allowed to have side effects. If a specification is going to be checked at run time via assertions, then the specification must not use methods with side effects. But, our checking is at compile time, so we allow it. Also, the current implementation will never able to prove such a contract, but it is able to use the information (when checking the method body with preconditions, or when checking the callers code with postconditions). This can be useful to annotate trusted methods precisely (e.g., `java.io.BufferedReader.ready()`).

(A side note: The formal parameter syntax `#1` may seem less convenient than writing the formal parameter name. This syntax is necessary because in the `.class` file, no parameter name information is available. Running the compiler without a checker should create legal annotations in the `.class` file, so we cannot rely on the checker to translate names to indices.)

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

- Some literals: floats, doubles, chars, and class literals.
- String concatenation expressions.
- Mathematical operators (plus, minus, division, ...).
- Comparisons (equality, less than, etc.).
- Quantification over all array components (e.g. to express that all array elements are non-null).

20.6 Unused fields

In an inheritance hierarchy, subclasses often introduce new methods and fields. For example, a `Marsupial` (and its subclasses such as `Kangaroo`) might have a variable `pouchSize` indicating the size of the animal's pouch. The field does not exist in superclasses such as `Mammal` and `Animal`, so Java issues a compile-time error if a program tries to access `myMammal.pouchSize`.

If you cannot use subtypes in your program, you can enforce similar requirements using type qualifiers. Section 20.6.1 describes the `@Unused` annotation, which enforces that a field or method may only be accessed from a receiver expression with a given annotation (or one of its subtypes).

Also see the discussion of typestate checkers, in Chapter 18.1.

20.6.1 `@Unused` annotation

A Java subtype can have more fields than its supertype. For example:

```
class Animal { }
class Mammal extends Animal { ... }
class Marsupial extends Mammal {
  int pouchSize;  // pouch capacity, in cubic centimeters
  ...
}
```

You can simulate the same effect for type qualifiers: the `@Unused` annotation on a field declares that the field may *not* be accessed via a receiver of the given qualified type (or any *super*type). For example:

```
class Animal {
  @Unused(when=Mammal.class)
  int pouchSize;  // pouch capacity, in cubic centimeters
  ...
}
@interface Mammal { }
@interface Marsupial { }

@Marsupial Animal joey = ...;
... joey.pouchSize ...    // OK
@Mammal Animal mae = ...;
... mae.pouchSize ...    // compile-time error
```

The above class declaration is like writing

```
class @Mammal-Animal { ... }
class @Marsupial-Animal {
  int pouchSize;  // pouch capacity, in cubic centimeters
  ...
}
```

Chapter 21

Handling warnings and legacy code

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

Also recall that you can convert checker errors into warnings via the `-Awarns` command-line option; see Section 2.2.1.

21.1 Checking partially-annotated programs: handling unannotated code

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

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

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

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

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

1. Annotate `unannotated_method` in full. This approach provides the strongest guarantees, but may require you to annotate additional methods that `unannotated_method` calls. See Chapter 22 for a discussion of how to annotate libraries for which you have no source code.
2. Annotate only the signature of `unannotated_method`, and suppress warnings in its body. Two ways to suppress the warnings are via a `@SuppressWarnings` annotation or by not running the checker on that file (see Section 21.2).
3. Suppress all warnings related to uses of `unannotated_method` via the `skipUses` processor option (see Section 21.2.4). Since this can suppress more warnings than you may expect, it is usually better to annotate at least the method's signature. If you choose the boundary between the annotated and unannotated code wisely, then you only have to annotate the signatures of a limited number of classes/methods (e.g., the public interface to a library or package).

Chapter 22 discusses adding annotations to signatures when you do not have source code available. Section 21.2 discusses suppressing warnings.

If you annotate a third-party library, please share it with us so that we can distribute the annotations with the Checker Framework; see Section 26.2.

21.2 Suppressing warnings

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

1. the `@SuppressWarnings` annotation (Section 21.2.1),
2. the `-AsuppressWarnings` command-line option (Section 21.2.2),
3. the `@AssumeAssertion` string in an `assert` message (Section 21.2.3),
4. the `-AskipUses` and `-AonlyUses` command-line options (Section 21.2.4),
5. the `-AskipDefs` and `-AonlyDefs` command-line options (Section 21.2.5),
6. the `-Alint` command-line option (Section 21.2.6),
7. not using the `-processor` command-line option (Section 21.2.7), or
8. checker-specific mechanisms (Section 21.2.8).

We now explain these mechanisms in turn.

21.2.1 `@SuppressWarnings` annotation

You can suppress specific errors and warnings by use of the `@SuppressWarnings("`*`checkername`*`")` annotation, for example `@SuppressWarnings("interning")` or `@SuppressWarnings("nullness")`. The argument *checkername* is in lower case and is derived from the way you invoke the checker; for example, if you invoke a checker as `javac -processor MyNiftyChecker ...`, then you would suppress its error messages with `@SuppressWarnings("mynifty")`. (An exception is the Subtyping Checker, for which you use the annotation name; see Section 17.1).

A `@SuppressWarnings` annotation may be placed on program declarations such as a local variable declaration, a method, or a class. It suppresses all warnings related to the given checker, for that program element. `@SuppressWarnings` is a declaration annotation, and it cannot be used on statements, expressions, or types.

For instance, one common use is to suppress warnings at a cast that you know is safe. Here is an example that uses the Tainting Checker (Section 7); assume that `expr` has type `@Tainted String`:

```
@SuppressWarnings("tainting") // Explain why the suppression is sound.
@Untainted String myvar = expr;
```

It would have been illegal to write

```
@Untainted String myvar;
@SuppressWarnings("tainting") // Explain why the suppression is sound.
myvar = expr;
```

because Java does not permit annotations (such as `@SuppressWarnings`) on assignments or other statements or expressions.

Each warning from the compiler also issues the most concrete suppression key that can be used to suppress that warning. Additionally, the `-AshowSuppressWarningKeys` command-line option can be used to show all applicable suppression keys.

Good practices when suppressing warnings

Suppress warnings in the smallest possible scope If a particular expression causes a false positive warning, you should extract that expression into a local variable and place a `@SuppressWarnings` annotation on the variable declaration, rather than suppressing warnings for a larger expression or an entire method body.

Justify why the warning is a false positive An `@SuppressWarnings` annotation asserts that the code is actually correct, even though the type system is unable to prove that the code is correct.

Whenever you write a `@SuppressWarnings` annotation, you should also write, typically on the same line or on the previous line, a code comment explaining why the code is actually correct. In some cases you might also justify why the code cannot be rewritten in a simpler way that would be amenable to type-checking.

This documentation will help you and others to understand the reason for the `@SuppressWarnings` annotation. It will also help if you decide to audit your code to verify all the warning suppressions.

Use a specific argument to `@SuppressWarnings` The `@SuppressWarnings` argument string can be of the form *checkername* or or *checkername:messagekey*. The *checkername* part is as described above. The *messagekey* part suppresses only errors/warnings relating to the given message key. For example, `cast.unsafe` is the key for warnings about an unsafe cast, and `cast.redundant` to the key for warnings about a redundant cast.

Thus, the above example could have been written as any one of the following, which would have suppressed the specific error:

```
@SuppressWarnings("tainting")              // suppresses all tainting-related warnings
@SuppressWarnings("tainting:cast.unsafe")  // suppresses tainting warnings about unsafe casts
@SuppressWarnings("tainting:cast")         // suppresses tainting warnings about casts
```

For a list of the message keys, see the `messages.properties` files in `checker-framework/checker/src/org/checkerframework/checker/`*`checkername`*`/messages.properties`. Each checker is built on the `basetype` checker and inherits its properties. Thus, to find all the error keys for a checker, you usually need to examine its own `messages.properties` file and that of `basetype`.

If a checker produces a warning/error and you want to determine its message key, you can re-run the checker, passing the the `-Anomsgtext` command-line option (Section 23.8).

21.2.2 `-AsuppressWarnings` command-line option

Supplying the `-AsuppressWarnings` command-line option is equivalent to writing a `@SuppressWarnings` annotation on every class that the compiler type-checks. The argument to `-AsuppressWarnings` is a comma-separated list of warning suppression keys, as in `-AsuppressWarnings=purity,uninitialized`.

When possible, it is better to write a `@SuppressWarnings` annotation with a smaller scope, rather than using the `-AsuppressWarnings` command-line option.

21.2.3 `@AssumeAssertion` string in an `assert` message

You can suppress a warning by `assert`ing that some property is true, and placing the string `@AssumeAssertion(`*`warningkey`*`)` in the assertion message.

For example, in this code:

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

the Nullness Checker assumes that `x` is non-null from the `assert` statement forward, and so the expression `x.f` cannot throw a null pointer exception.

The assert expression must be an expression that would affect flow-sensitive type qualifier refinement (Section 20.4), if the expression appeared in a conditional test. Each type system has its own rules about what type refinement it performs, if any.

The warning key is exactly as in the `@SuppressWarnings` annotation (Section 21.2.1). The same good practices apply as for a `@SuppressWarnings` annotations, such as writing a comment justifying why the assumption is safe.

If the string `@AssumeAssertion(`*`warningkey`*`)` does not appear in the assertion message, then the Checker Framework treats the assertion as being used for defensive programming. That is, the programmer believes that the assertion might fail at run time, so the Checker Framework should not make any inference, which would not be justified.

A downside of putting the string in the assertion message is that if the assertion ever fails, then a user might see the string and be confused. But the string should only be used if the programmer has reasoned that the assertion can never fail.

21.2.4 `-AskipUses` and `-AonlyUses` command-line options

You can suppress all errors and warnings at all *uses* of a given class, or suppress all errors and warnings except those at uses of a given class. (The class itself is still type-checked, unless you also use the `-AskipDefs` or `-AonlyDefs` command-line option, see 21.2.5).

Set the `-AskipUses` command-line option to a regular expression that matches class names (not file names) for which warnings and errors should be suppressed. Or, set the `-AonlyUses` command-line option to a regular expression that matches class names (not file names) for which warnings and errors should be emitted; warnings about uses of all other classes will be suppressed.

For example, suppose that you use "`-AskipUses=^java\.`" on the command line (with appropriate quoting) when invoking `javac`. Then the checkers will suppress all warnings related to classes whose fully-qualified name starts with `java.`, such as all warnings relating to invalid arguments and all warnings relating to incorrect use of the return value.

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

You can supply both `-AskipUses` and `-AonlyUses`, in which case the `-AskipUses` argument takes precedence, and `-AonlyUses` does further filtering but does not add anything that `-AskipUses` removed.

Warning: Use the `-AonlyUses` command-line option with care, because it can have unexpected results. For example, if the given regular expression does not match classes in the JDK, then the Checker Framework will suppress every warning that involves a JDK class such as `Object` or `String`. The meaning of `-AonlyUses` may be refined in the future. Oftentimes `-AskipUses` is more useful.

21.2.5 `-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 21.2.4).

Set the `-AskipDefs` command-line option to a regular expression that matches class names (not file names) in whose definition warnings and errors should be suppressed. Or, set the `-AonlyDefs` command-line option to a regular expression that matches class names (not file names) whose definitions should be type-checked.

For example, if you use "`-AskipDefs=^mypackage\.`" on the command line (with appropriate quoting) when invoking `javac`, then the definitions of classes whose fully-qualified name starts with `mypackage.` will not be checked.

If you supply both `-AskipDefs` and `-AonlyDefs`, then `-AskipDefs` takes precedence.

Another way not to type-check a file is not to pass it on the compiler command-line: the Checker Framework type-checks only files that are passed to the compiler on the command line, and does not type-check any file that is not passed to the compiler. The `-AskipDefs` and `-AonlyDefs` command-line options are intended for situations in which the build system is hard to understand or change. In such a situation, a programmer may find it easier to supply an extra command-line argument, than to change the set of files that is compiled.

A common scenario for using the arguments is when you are starting out by type-checking only part of a legacy codebase. After you have verified the most important parts, you can incrementally check more classes until you are type-checking the whole thing.

21.2.6 `-Alint` command-line option

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

- `cast:unsafe` (default: on) warn about unsafe casts that are not checked at run time, as in `((@NonNull String) myref)`. Such casts are generally not necessary when flow-sensitive local type refinement is enabled.
- `cast:redundant` (default: on) warn about redundant casts that are guaranteed to succeed at run time, as in `((@NonNull String) "m")`. Such casts are not necessary, because the target expression of the cast already has the given type qualifier.
- `cast` Enable or disable all cast-related warnings.
- `all` Enable or disable all lint warnings, including checker-specific ones if any. Examples include `redundantNullComparison` for the Nullness Checker (see Section 1) and `dotequals` for the Interning Checker (see Section 4.3). This option does not enable/disable the checker's standard checks, just its optional ones.
- `none` The inverse of `all`: disable or enable all lint warnings, including checker-specific ones if any.

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

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`.

21.2.7 No `-processor` command-line option

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

21.2.8 Checker-specific mechanisms

Finally, some checkers have special rules. For example, the Nullness checker (Chapter 3) uses `assert` statements that contain null checks, and the special `castNonNull` method, to suppress warnings (Section 3.4.1). This manual also explains special mechanisms for suppressing warnings issued by the Fenum Checker (Section 6.4) and the Units Checker (Section 13.5).

21.3 Backward compatibility with earlier versions of Java

Sometimes, your code needs to be compiled by people who are not using a compiler that supports type annotations. Sections 21.3.1–21.3.3 discuss this situation, which you can handle by writing annotations in comments.

In other cases, your code needs to be run by people who are not using a Java 8 JVM. Section 21.3.4 discusses this situation, which you can handle by passing the `-target 5` command-line argument.

(**Note:** These are features of the Type Annotations compiler that is distributed along with the Checker Framework. They are *not* supported by the mainline OpenJDK compiler. These features are the key difference between the Type Annotations compiler and the OpenJDK compiler on which it is built.)

21.3.1 Annotations in comments

A Java 4 compiler does not permit use of annotations, and a Java 5/6/7 compiler only permits annotations on declarations (but not on generic arguments, casts, `extends` clauses, method receiver, etc.).

So that your code can be compiled by any Java compiler (for any version of the Java language), you may write any single annotation inside a `/*...*/` Java comment, as in `List</*@NonNull*/ String>`. The Type Annotations compiler treats the code exactly as if you had not written the `/*` and `*/`. In other words, the Type Annotations compiler will recognize the annotation, but your code will still compile with any other Java compiler.

This feature only works if you provide no `-source` command-line argument to `javac`, or if the `-source` argument is `1.8` or `8`.

In a single program, you may write some annotations in comments, and others outside of comments.

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

Compiler flag `-XDTA:noannotationsincomments` causes the compiler to ignore annotation comments. With this compiler flag the Type Annotations compiler behaves like a standard Java 8 compiler that does not support annotations in comments. If your code already contains comments of the form `/*@...*/` that look like type annotations, and you want the Type Annotations compiler not to try to interpret them, then you can either selectively add spaces to the comments or use `-XDTA:noannotationsincomments` to turn off all annotation comments.

There is a more powerful mechanism that permits arbitrary code to be written in a comment that is formatted as "`/*»>...*/`", with the first three characters of the comment being greater-than signs. As with annotations in comments, the commented code is ignored by ordinary compilers but is treated like ordinary code by the UW Type Annotations compiler.

This mechanism is intended for two purposes. First, it supports the receiver (`this` parameter) syntax, as in

```
public boolean getResult(/*>>> @ReadOnly MyClass this*/) { ... }
public boolean getResult(/*>>> @ReadOnly MyClass this, */ String argument) { ... }
```

for a method that does not modify its receiver.

Second, it can be used for import statements:

```
/*>>>
import org.checkerframework.checker.nullness.qual.*;
import org.checkerframework.checker.regex.qual.*;
*/
```

Such a use avoids dependences on qualifiers. It also eliminates Eclipse warnings about unused import statements, if all uses of the imported qualifier are themselves in comments and thus invisible to Eclipse.

A third use is for adding multiple annotations inside one comment. However, it is better style to write multiple annotations each inside its own comment, as in `/*@NonNull*/ /*@Interned*/ String s;`.

It would be possible to abuse the `/*»>...*/` mechanism to inject code only when using the Type Annotations compiler. Doing so is not a sanctioned use of the mechanism.

21.3.2 Implicit import statements

When writing source code with annotations, it is more convenient to write a short form such as `@NonNull` instead of `@org.checkerframework.checker.nullness.qual.NonNull`. Here are ways to achieve this goal.

- The traditional way to do this is to write an import statement like "`import org.checkerframework.checker.nullness.qual.*;`". This works, but everyone who compiles the code (no matter what compiler they use, and even if the annotations are in comments) must have the annotation definitions (e.g., the `checker.jar` or `checker-qual.jar` file) on their classpath. The reason is that a Java compiler issues an error if an imported package is not on the classpath. See Section 2.1.1.
- Write an import statement in a comment, just as for annotations in comments:

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

- An alternative is to set the shell environment variable `jsr308_imports` when you compile the code. The Type Annotations compiler treats this as if the given packages/classes were imported, but other compilers ignore the `jsr308_imports` environment variable — they do not need it, since they do not support annotations in comments. Thus, your code can compile whether or not the Type Annotations compiler is being used.
  You can specify multiple packages/classes separated by the classpath separator (same as the file path separator: `;` for Windows, and `:` for Unix and Mac). For example, to implicitly import the Nullness, Interning, and dataflow qualifiers, set `jsr308_imports` to `org.checkerframework.checker.nullness.qual.*:org.checkerframework.checker.interning.qual.*:org.checker`
  Three ways to set an environment variable are:
  - Set the environment variable in your shell. For example, in bash, you could do `export jsr308_imports='org.checkerframework.checker.nullness.qual.*:org.checkerframework.dataflow.qual.*`
    This takes effect for all subsequent commands in that shell. To take effect for all shells that you run, put the command in your `~/.bashrc` file.
  - Set the environment variable for a single command. For example, in bash prefix the `javac` command by `jsr308_imports='org.checkerframework.checker.nullness.qual.*:dataflow.qual.*'`.
  - Set the environment variable for a single command, via a `javac` argument. Use the `javac` command-line argument `-J-Djsr308_imports='org.checkerframework.checker.nullness.qual.*:dataflow.qual.*'`.

  If you issue the javac command from the command line or in a Makefile, you may need to add quotes (as shown above), to prevent your shell from expanding the `*` character. If you supply the `-J-Djsr308_imports` argument via an Ant buildfile, you do not need the extra quoting.
- If it is not possible to set the environment variable, you can instead use a different javac command-line argument: `-jsr308_imports ...` or `-Djsr308.imports=...` (they have the same effect). The same syntax for the packages/classes, and the same warnings about quoting from the command line, apply as for the `jsr308_imports` environment variable.

21.3.3 Migrating away from annotations in comments

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

[TODO: This doesn't handle the »> comments. Adapt it to do so.]

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

You can customize this command:

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

If your code used implicit import statements (Section 21.3.2), then after uncommenting the annotations, you may also need to introduce explicit import statements into your code.

21.3.4 Annotations in Java 5 `.class` files

If you supply the `-target 5` command-line argument along with no `-source` argument (or `-source 8`, which is equivalent), then the Type Annotations compiler creates a `.class` file that can be run on a Java 5 JVM, but that contains the type annotations. (It does not matter whether the type annotations were written in comments or not.) The fact that the `.class` file contains the type annotations is useful when type-checking client code. If you try to type-check client code against a library that lacks type annotations, then spurious warnings can result. So, use of `-target 5` gives backward compatibility with earlier JVMs while still permitting pluggable type-checking.

Ordinary Java compilers do not let you use a `-target` command-line argument with a value less than the `-source` argument.

Use of the `-source 5` command-line argument produces a `.class` file that does not contain type annotations. One reason you might want to periodically compile with the `-source 5` argument is to ensure that your code does not use any Java 8 features other than type annotations in comments.

Chapter 22

Annotating libraries

When annotated code uses an unannotated library, a checker may issue warnings. As described in Section 21.1, the best way to correct this problem is to add annotations to the library. (Alternately, you can instead suppress all warnings related to an unannotated library by use of the `-AskipUses` or `-AonlyUses` command-line option; see Section 21.2.4.) If you have source code for the library, you can easily add the annotations. This chapter tells you how to add annotations to a library for which you have no source code, because the library is distributed only in binary form (as `.class` files, possibly packaged in a `.jar` file). This chapter is also useful if you do not wish to edit the library's source code.

Note that this chapter is about annotating libraries, not analyzing them. The Checker Framework analyzes all, and only, the source code that is passed to it. The Checker Framework is a plug-in to the javac compiler, and it never analyzes code that is not being compiled, though it does look up annotations for code that is not being compiled.

You can make the library's annotations known to the checkers in two ways.

- You can write annotations in a "stub file" containing classes with no method bodies. Section 22.2 describes how to create and use stub files.
- You can insert annotations in the compiled `.class` files of the library. You would express the annotations textually, typically as an annotation index file, and then insert them in the library by using the Annotation File Utilities (`http://types.cs.washington.edu/annotation-file-utilities/`). See the Annotation File Utilities documentation for full details.

The Checker Framework distribution contains annotations for popular libraries, such as the JDK6 and JDK7. It uses both of the above mechanisms. The Nullness, Javari, IGJ, and Interning Checkers use the annotated JDKs (Section 22.3), and all other checkers use stub files (Section 22.2).

If you annotate additional libraries, please share them with us so that we can distribute the annotations with the Checker Framework; see Section 26.2. You can determine the correct annotations for a library either automatically by running an inference tool, or manually by reading the documentation. Presently, type inference tools are available for the Nullness (Section 3.3.7) and Javari (Section 16.2.2) type systems.

22.1 Choosing between stub files and annotated `.class` files

A checker can read annotations either from a stub file or from a library's `.class` files. This section helps you choose between the two alternatives.

Once created, a stub file can be used directly; this makes it an easy way to get started with library annotations. When provided by the author of the checker, a stub file is used automatically, with no need for the user to supply a command-line option.

Inserting annotations in a library's `.class` files takes an extra step using an external tool, the Annotation File Utilities (`http://types.cs.washington.edu/annotation-file-utilities/`). However, this process does not suffer the limitations of stub files (Section 22.2.5).

22.2 Using stub classes

A stub file contains "stub classes" that contain annotated signatures, but no method bodies. A checker uses the annotated signatures at compile time, instead of or in addition to annotations that appear in the library.

Section 22.2.3 describes how to create stub classes. Section 22.2.1 describes how to use stub classes. These sections illustrate stub classes via the example of creating a `@Interned`-annotated version of `java.lang.String`. You don't need to repeat these steps to handle `java.lang.String` for the Interning Checker, but you might do something similar for a different class and/or checker.

22.2.1 Using a stub file

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

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

Each stub path entry is a file or a directory; specifying a directory is equivalent to specifying every file in it whose name ends with `.astub`. The stub path entries are delimited by `File.pathSeparator` (`:` for Linux and Mac, `;` for Windows).

A checker automatically reads the stub file `jdk.astub`. (The checker author should place it in the same directory as the Checker class, i.e., the subclass of `BaseTypeVisitor`.) Programmers should only use the `-Astubs` argument for additional stub files they create themselves.

If a method appears in more than one stub file (or twice in the same stub file), then the annotations are merged. If any of the methods have different annotations from the same hierarchy on the same type, then the annotation from the last declaration is used.

22.2.2 Stub file format

Every Java file is a valid stub file. However, you can omit information that is not relevant to pluggable type-checking; this makes the stub file smaller and easier for people to read and write.

As an illustration, a stub file for the Interning type system (Chapter 4) could be:

```
import org.checkerframework.checker.interning.qual.Interned;
package java.lang;
@Interned class Class<T> { }
class String {
  @Interned String intern();
}
```

Note, annotations in comments are ignored.

The stub file format is allowed to differ from Java source code in the following ways:

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

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

Declaration specifiers: Declaration specifiers (e.g., `public`, `final`, `volatile`) may be omitted.

Return types: The return type of a method does not need to match the real method. In particular, it is valid to use `java.lang.Object` for every method. This simplifies the creation of stub files.

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

Enum constants in annotations need to be either fully qualified or imported. For example, one has to either write the enum constant `ANY` in fully-qualified form:

```
@Source(sparta.checkers.quals.FlowPermission.ANY)
```

or correctly import the enum class:

```
import sparta.checkers.quals.FlowPermission;
...
@Source(FlowPermission.ANY)
```

or statically import the enum constants:

```
import static sparta.checkers.quals.FlowPermission.*;
...
@Source(ANY)
```

Importing all packages from a class (`import my.package.*;`) only considers annotations from that package; enum types need to be explicitly imported.

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

22.2.3 Creating a stub file

If you have access to the Java source code

Every Java file is a stub file. If you have access to the Java file, then you can use the Java file as the stub file, without removing any of the parts that the stub file format permits you to. Just add annotations to the signatures, leaving the method bodies unchanged. Optionally (but highly recommended!), run the type-checker to verify that your annotations are correct. When you run the type-checker on your annotations, there should not be any stub file that also contains annotations for the class. In particular, if you are type-checking the JDK itself, then you should use the `-Aignorejdkastub` command-line option.

This approach retains the original documentation and source code, making it easier for a programmer to double-check the annotations. It also enables creation of diffs, easing the process of upgrading when a library adds new methods. And, the annotations are in a format that the library maintainers can even incorporate.

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

If you do not have access to the Java source code

If you do not have access to the library source code, then you can create a stub file from the class file (Section 22.2.3), and then annotate it. The rest of this section describes this approach.

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

   ```
   cd nullness-stub
   java org.checkerframework.framework.stub.StubGenerator java.lang.String > String.astub
   ```

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

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

   The stub file parser silently ignores any annotations that it cannot resolve to a type, so don't forget the import statement. Use the `-AstubWarnIfNotFound` command-line option to see warnings if an entry could not be found.
3. Add annotations to the stub class. For example, you might annotate the `String.intern()` method as follows:

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

   You may also remove irrelevant parts of the stub file; see Section 22.2.2.

Two command-line options can be used to debug the behavior of stub files: `-AstubWarnIfNotFound` warns if a stub file entry could not be found. Annotations on unknown classes and methods are silently ignored. Use this option to ensure that all stub file entries could be resolved. `-AstubDebug` outputs debug messages while parsing stub files.

22.2.4 Troubleshooting stub libraries

An error is issued if a stub file has a typo or the API method does not exist.

Fix this error by removing the extra L in the method name:

```
StubParser: Method isLLowerCase(char) not found in type java.lang.Character
```

Fix this error by removing the method `enableForgroundNdefPush(...)` from the stub file, because it is not defined in class `android.nfc.NfcAdapter` in the version of the library you are using:

```
StubParser: Method enableForegroundNdefPush(Activity,NdefPushCallback)
      not found in type android.nfc.NfcAdapter
```

22.2.5 Limitations

The stub file reader has several limitations. We will fix these in a future release.

- The receiver is written after the method parameter list, instead of as an explicit first parameter. That is, instead of

  ```
  returntype methodname(@Annotations C this, params);
  ```

  in a stub file one has to write

  ```
  returntype methodname(params) @Annotations;
  ```

- The stub file reader does not handle nested classes. To work around this, it permits a top-level class to be written with a `$` in its name, and applies the annotations to the appropriate nested class.
- Annotations must be written before the package name on a fully qualified types rather than directly on the type it qualifies. However, it is usually not necessary to write the fully qualified name.

  ```
  void init(@Nullable java.security.SecureRandom random);
  ```

- Annotations can only use string or boolean literals; other literals are not yet supported.

If these limitations are a problem, then you should insert annotations in the library's `.class` files instead.

22.3 Using distributed annotated JDKs

The Checker Framework distribution contains two annotated JDKs at the paths `checker/bin/jdk7.jar` and `checker/dist/jdk8.jar`. The `javac` that is distributed with the Checker Framework and the command `java -jar $CHECKERFRAMEWORK/checker/dist/checker` both use the appropriate jdk6.jar or jdk7.jar based on the version of Java used to run them.

The annotated JDKs should *not* be in your classpath at run time, only at compile time.

22.4 Troubleshooting/debugging annotated libraries

Sometimes, it may seem that a checker is treating a library as unannotated even though the library has annotations. The compiler has two flags that may help you in determining whether library files are read, and if they are read whether the library's annotations are parsed.

`-verbose` Outputs info about compile phases — when the compiler reads/parses/attributes/writes any file. Also outputs the classpath and sourcepath paths.

`-XDTA:parser` (which is equivalent to `-XDTA:reader` plus `-XDTA:writer`) Sets the internal `debugJSR308` flag, which output information about reading and writing.

Chapter 23

How to create a new checker

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

Warning: Due to recent improvements and refactorings, a few parts of this chapter are out of date as of December 2013. The Checker Framework developers are working to update it. If you notice inaccuracies or can make suggestions to improve this chapter, please do so. Thanks!

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

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

This checker is so short because it builds on the Subtyping Checker (Chapter 17). See Section 17.2 for more details about this particular checker. When you wish to create a new checker, it is often easiest to begin by building it declaratively on top of the Subtyping Checker, and then return to this chapter when you need more expressiveness or power than the Subtyping Checker affords.

You can also create your own checker by customizing a different existing checker. Specific checkers that are designed for extension (besides the Subtyping Checker) include the Fake Enumeration Checker (Chapter 6, page 51), the Units Checker (Chapter 13, page 74), and the typestate checkers (Chapter 18.1, page 88). Or, you can copy and then modify a different existing checker — whether one distributed with the Checker Framework or a third-party one.

You can place your checker's source files wherever you like. When you compile your checker, `$CHECKERFRAMEWORK/framework/dist/framework.jar` and `$CHECKERFRAMEWORK/framework/dist/javac.jar` should be on your classpath. (If you wish to modify an existing checker in place, or to place the source code for your new checker in your own private copy of the Checker Framework source code, then you need to be able to re-compile the Checker Framework, as described in Section 26.3.)

The rest of this chapter contains many details for people who want to write more powerful checkers. You do not need all of the details, at least at first. In addition to reading this chapter of the manual, you may find it helpful to examine the implementations of the checkers that are distributed with the Checker Framework. You can even create your checker by modifying one of those. The Javadoc documentation of the framework and the checkers is in the distribution and is also available online at `http://types.cs.washington.edu/checker-framework/current/api/`.

If you write a new checker and wish to advertise it to the world, let us know so we can mention it in the Checker Framework Manual, link to it from the webpages, or include it in the Checker Framework distribution. For examples, see Chapters 18.1 and 18.

23.1 Relationship of the Checker Framework to other tools

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

<table>
<tr><td>Subtyping Checker</td><td>Nullness Checker</td><td>Mutation 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">Type Annotations syntax and classfile format (“JSR 308”)
(no built-in semantics)</td></tr>
</table>

The Base Checker enforces the standard subtyping rules on extended types. The Subtyping Checker is a simple use of the Base Checker that supports providing type qualifiers on the command line. You usually want to build your checker on the Base Checker.

23.2 The parts of a checker

The Checker Framework provides abstract base classes (default implementations), and a specific checker overrides as little or as much of the default implementations as necessary. Sections 23.3–23.6 describe the components of a type system as written using the Checker Framework:

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

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

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

- Subtyping rules related to the type hierarchy, such as that every assignment and pseudo-assignment satisfies a subtyping relationship. Your checker automatically inherits these subtyping rules from the Base Checker (Chapter 17).
- Additional rules that are specific to your particular checker. For example, in the Nullness type system, only references with a `@NonNull` type may be dereferenced. You write these additional rules yourself.

23.6 **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.

23.3 Annotations: Type qualifiers and hierarchy

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

Type qualifiers are defined as Java annotations [Dar06]. In Java, an annotation is defined using the Java `@interface` keyword. For example:

```
// Define an annotation for the @NonNull type qualifier.
@TypeQualifier
@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})
public @interface NonNull { }
```

Write the `@TypeQualifier` meta-annotation on the annotation definition to indicate that the annotation represents a type qualifier and should be processed by the checker. Also write a `@Target` meta-annotation to indicate where the annotation may be written. (An annotation that is written on an annotation definition, such as `@TypeQualifier`, is called a *meta-annotation*.)

The type hierarchy induced by the qualifiers can be defined either declaratively via meta-annotations (Section 23.3.1), or procedurally through subclassing `QualifierHierarchy` or `TypeHierarchy` (Section 23.3.2).

23.3.1 Declaratively defining the qualifier and type hierarchy

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

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

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

  `@SubtypeOf` accepts multiple annotation classes as an argument, permitting the type hierarchy to be an arbitrary DAG. For example, in the IGJ type system (Section 15.2), `@Mutable` and `@Immutable` induce two mutually exclusive subtypes of the `@ReadOnly` qualifier.
  All type qualifiers, except for polymorphic qualifiers (see below and also Section 19.2), need to be properly annotated with `SubtypeOf`.
  The top qualifier is annotated with `@SubtypeOf( { } )`. The top qualifier is the qualifier that is a supertype of all other qualifiers. For example, `@Nullable` is the top qualifier of the Nullness type system, hence is defined as:

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

  If the top qualifier of the hierarchy is the unqualified type, then its children will use `@SubtypeOf(Unqualified.class)`, but no `@SubtypeOf( { } )` annotation on the top qualifier is necessary. For an example, see the `Encrypted` type system of Section 17.2.
- `@PolymorphicQualifier` denotes that a qualifier is a polymorphic qualifier. For example:

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

  For a description of polymorphic qualifiers, see Section 19.2. A polymorphic qualifier needs no `@SubtypeOf` meta-annotation and need not be mentioned in any other `@SubtypeOf` meta-annotation.

The declarative and procedural mechanisms for specifying the hierarchy can be used together. In particular, when using the `@SubtypeOf` meta-annotation, further customizations may be performed procedurally (Section 23.3.2) by overriding the `isSubtype` method in the checker class (Section 23.6). However, the declarative mechanism is sufficient for most type systems.

23.3.2 Procedurally defining the qualifier and type hierarchy

While the declarative syntax suffices for many cases, more complex type hierarchies can be expressed by overriding, in `BaseTypeVisitor`, either `createQualifierHierarchy` or `createTypeHierarchy` (typically only one of these

needs to be overridden). For more details, see the Javadoc of those methods and of the classes `QualifierHierarchy` and `TypeHierarchy`.

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

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

23.3.3 Defining a default annotation

A type system designer may set a default annotation. A user may override the default; see Section 20.3.1.

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

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

Recall that defaults are distinct from implicit annotations; see Sections 20.3 and 23.4.

23.3.4 Completeness of the type hierarchy

When you define a type system, its type hierarchy must be a complete lattice — that is, there must be a top type that is a supertype of all other types, and there must be a bottom type that is a subtype of all other types. Furthermore, it is best if the top type and bottom type are defined explicitly for the type system, rather than (say) reusing a qualifier from the Checker Framework such as `@Unqualified`.

It is possible that a single type-checker checks multiple type hierarchies. An example is the Nullness Checker that has separate type hierarchies for nullness, initialization, and map keys. In this case, each type hierarchy would have its own top qualifier and its own bottom qualifier; they don't all have to share a single top qualifier or a single bottom qualifier.

Bottom qualifier Your type hierarchy must have a bottom qualifier — a qualifier that is a (direct or indirect) subtype of every other qualifier.

Your type system must give `null` the bottom type. (The only exception is if the type system has special treatment for `null` values, as the Nullness Checker does.) This legal code will not type-check unless `null` has the bottom type:

```
<T> T f() {
    return null;
```

You don't necessarily have to define a new bottom qualifier. But, You can use `org.checkerframework.framework.qual.Bottom` if your type system does not already have an appropriate bottom qualifier.

If your type system has a special bottom type that is used *only* for the `null` value, then users should never write the bottom qualifier explicitly. To ensure this, write `@Target({})` on the definition of the bottom qualifier.

The hierarchy shown in Figure 15.1 lacks a bottom qualifier, because there is no qualifier that is a subtype of both `@Immutable` and `@Mutable`. The actual IGJ hierarchy does contain a (non-user-visible) bottom qualifier, defined like this:

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

Top qualifier Your type hierarchy must have a top qualifier — a qualifier that is a (direct or indirect) supertype of every other qualifier. The default type for local variables is the top qualifier (that type is then flow-sensitively refined depending on what values are stored in the local variable). If there is no single top qualifier, then there is no unambiguous choice to make for local variables.

Furthermore, it is most convenient to users if the top qualifier is defined by the type system. An example of a type system that does *not* do that is the `@Encrypted` type system of Section 17.2. It lacks its own explicit top qualifier and instead uses `@Unqualified`, which is shared across multiple type systems:

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

It can be convenient to use `@Unqualified` as the top type to avoid having to define your own top type. The disadvantage is that users lose flexibility in expressing defaults: there is no way for a user to change the default qualifier for just that type system. If a user specifies `@DefaultQualifier(Unqualified.class)`, then the default would apply to every type system that uses `@Unqualified`, which is unlikely to be desired.

It is best if a type system as an explicit qualifier for every possible meaning. For example, the Nullness type system has both `@Nullable` and `@NonNull`. Because it has no built-in meaning for unannotated types; a user may specify a default qualifier. This greater flexibility for the user is usually preferable.

There are reasons to not explicitly define the top qualifier, but to reuse `@Unqualified`. The ability to omit the top qualifier is a convenience when writing a type system, because it reduces the number of qualifiers that must be defined; this is especially convenient when using the Subtyping Checker (Chapter 17). More importantly, omitting the top qualifier restricts the user in ways that the type system designer may have intended.

23.4 Type factory: Implicit annotations

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

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

23.4.1 Declaratively specifying implicit annotations

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

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

- `trees`: an array of `com.sun.source.tree.Tree.Kind`, e.g., `NEW_ARRAY` or `METHOD_INVOCATION`
- `types`: an array of `TypeKind`, e.g., `ARRAY` or `BOOLEAN`
- `treeClasses`: an array of class literals for classes implementing `Tree`, e.g., `LiteralTree.class` or `ExpressionTree.class`
- `typeClasses`: an array of class literals for classes implementing `javax.lang.model.type.TypeMirror`, e.g., `javax.lang.model.type.PrimitiveType`. Often you should use a subclass of `AnnotatedTypeMirror`.

- `stringPatterns`: an array of regular expressions that will be matched against string literals, e.g., `"[01]+"` for a binary number. Useful for annotations that indicate the format of a string.

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

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


@TypeQualifier
@SubtypeOf({})
@ImplicitFor(trees={Tree.Kind.NULL_LITERAL})
@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})
public @interface Nullable { }
```

For more details, see the Javadoc for the `ImplicitFor` annotation, and the Javadoc for the javac classes that are linked from it. You only need to understand a small amount about the javac AST, such as the `Tree.Kind` and `TypeKind` enums. All the information you need is in the Javadoc, and Section 23.9 can help you get started.

23.4.2 Procedurally specifying implicit annotations

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

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

23.4.3 Flow-sensitive type qualifier inference

The Checker Framework provides automatic type refinement as described in Section 20.4.

Class `BaseAnnotatedTypeFactory` provides a 2 parameter constructor that allows subclasses to disable flow inference. By default the 1 parameter constructor performs flow inference. To disable flow inference, call `super(checker, root, false);` in your subtype of `BaseAnnotatedTypeFactory`.

23.5 Visitor: Type rules

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

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

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

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

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

- invalid assignment (type.incompatible) for an assignment from an expression type to an incompatible type. The assignment may be a simple assignment, or pseudo-assignment like return expressions or argument passing in a method invocation
  In particular, in every assignment and pseudo-assignment, the left-hand side of the assignment is a supertype of (or the same type as) the right-hand side. For example, this assignment is not permitted:

  ```
  @Nullable Object myObject;
  @NonNull Object myNonNullObject;
  ...
  myNonNullObject = myObject;  // invalid assignment
  ```
- invalid generic argument (type.argument.type.incompatible) when a type is bound to an incompatible generic type variable
- invalid method invocation (method.invocation.invalid) when a method is invoked on an object whose type is incompatible with the method receiver type
- invalid overriding parameter type (override.parameter.invalid) when a parameter in a method declaration is incompatible with that parameter in the overridden method's declaration
- invalid overriding return type (override.return.invalid) when a parameter in a method declaration is incompatible with that parameter in the overridden method's declaration
- invalid overriding receiver type (override.receiver.invalid) when a receiver in a method declaration is incompatible with that receiver in the overridden method's declaration

23.5.1 AST traversal

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

Because there is no standard API for the AST of Java code[1], the Checker Framework uses the javac implementation. This is why the Checker Framework is not deeply integrated with Eclipse, but runs as an external tool (see Section 24.6).

23.5.2 Avoid hardcoding

It may be tempting to write a type-checking rule for method invocation, where your rule checks the name of the method being called and then treats the method in a special way. This is usually the wrong approach. It is better to write annotations, in a stub file (Chapter 22), and leave the work to the standard type-checking rules.

[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.

23.6 The checker class: Compiler interface

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

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

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

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

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

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

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

A checker can customize the default error messages through a `Properties`-loadable text file named `messages.properties` that appears in the same directory as the checker class. The property file keys are the strings passed to `report` (like `type.incompatible`) and the values are the strings to be printed (`"cannot assign ..."`). The `messages.properties` file only need to mention the new messages that the checker defines. It is also allowed to override messages defined in superclasses, but this is rarely needed. For more details about message keys, see Section 21.2.1 (page 110).

23.6.1 Bundling multiple checkers

To run a checker, a user supplies the `-processor` command-line option. There are two ways to run multiple related checkers as a unit.

1. A user can pass multiple `-processor` command-line options, like:

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

   This is verbose, and it is also error-prone, since a user might omit one of several related checkers that are designed to be run together.
2. You can define an aggregate checker class that combines multiple checkers. Extend `AggregateChecker` and override the `getSupportedTypeCheckers` method, like the following:

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

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

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

23.6.2 Providing command-line options

A checker can provide two kinds of command-line options: boolean flags and named string values (the standard annotation processor options).

Boolean flags

To specify a simple boolean flag, add:

```
@SupportedLintOptions({"flag"})
```

to your checker subclass. The value of the flag can be queried using

```
checker.getLintOption("flag", false)
```

The second argument sets the default value that should be returned.
To pass a flag on the command line, call javac as follows:

```
javac -processor Mine -Alint=flag
```

Named string values

For more complicated options, one can use the standard annotation processing `@SupportedOptions` annotation on the checker, as in:

```
@SupportedOptions({"info"})
```

The value of the option can be queried using

```
checker.getOption("info")
```

To pass an option on the command line, call javac as follows:

```
javac -processor Mine -Ainfo=p1,p2
```

The value is returned as a single string and you have to perform the required parsing of the option.

23.7 Testing framework

The Checker Framework comes with a testing framework that is used for testing the distributed checkers. It is easy to use this testing framework to ensure correctness of your checker!

You first need to provide a subclass of `ParameterizedCheckerTest` that determines the checker to use and all command-line options that should be provided. This class can be run as a JUnit test runner. Note that you always need to use the `-Anomsgtext` option to suppress the substitution of message keys by human-readable values. See the test setup classes in directory `tests/src/tests/` for examples.

Locate all your test cases in a subdirectory of the `tests` directory. The individual test cases are normal Java files that use stylized comments to indicate expected error messages. For example, consider this test case from the Nullness Checker:

```
//:: error: (dereference.of.nullable)
s.toString();
```

An expected error message is introduced by the `//::` comment. The next token is either `error:` or `warning:`, distinguishing what kind of message is expected. Finally, the message key for the expected message is given.

Multiple expected messages can be given using the "//:: A :: B :: C" syntax. This example expects both an error and a warning from the same line of code:

```
@Regex String s1 = null;
//:: error: (assignment.type.incompatible) :: warning: (cast.unsafe)
@Regex(3) String s2 = (@Regex(2) String) s;
```

As an alternative, expected errors can be specified in a separate file using the `.out` file extension. These files are of the following format:

```
:19: error: (dereference.of.nullable)
```

The number between the colons is the line number of the expected error message. This format is a lot harder to maintain and we suggest using the in-line comment format.

23.8 Debugging options

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

23.8.1 Amount of detail in messages

- `-AprintAllQualifiers`: print all type qualifiers, including qualifiers like `@Unqualified` which are usually not shown. (Use the `@InvisibleQualifier` meta-annotation on a qualifier to hide it.)
- `-Adetailedmsgtext`: print a more detailed error message in addition to the full message text when reporting errors or warnings. This is useful for tools that wish to parse the Checker Framework output.
- `-AprintErrorStack`: print a stack trace whenever an internal Checker Framework error occurs.
- `-Anomsgtext`: use message keys (such as "`type.invalid`") rather than full message text when reporting errors or warnings. This is used by the Checker Framework's own tests, so they do not need to be changed if the English message is updated.

23.8.2 Stub and JDK libraries

- `-Aignorejdkastub`: ignore the `jdk.astub` file in the checker directory. Files passed through the `-Astubs` option are still processed. This is useful when experimenting with an alternative stub file.
- `-Anocheckjdk`: don't issue an error if no annotated JDK can be found.
- `-AstubDebug`: Print debugging messages while processing stub files.

23.8.3 Progress tracing

- `-Afilenames`: print the name of each file before type-checking it.
- `-Ashowchecks`: print debugging information for each pseudo-assignment check (as performed by `BaseTypeVisitor`; see Section 23.5.

23.8.4 Miscellaneous debugging options

- `-Aflowdotdir`: Directory for .dot files that visualize the control flow graph of all the methods and code fragments analyzed by the dataflow analysis. The graph also contains information about flow-sensitively refined types of various expressions at many program points.
- `-AresourceStats`: Whether to output resource statistics at JVM shutdown.

23.8.5 Examples

The following example demonstrates how these options are used:

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

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

You can use any standard debugger to observe the execution of your checker. Set the execution main class to `com.sun.tools.javac.Main`, and insert the JSR 308 javac.jar (resides in `.../jsr308-langtools/dist/lib/javac.jar`). If using an IDE, it is recommended that you add `.../jsr308-langtools` as a project, so you can step into its source code if needed.

You can also set up remote (or local) debugging using the following command as a template:

```
java -jar $CHECKERFRAMEWORK/framework/dist/framework.jar \
    -J-Xdebug -J-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005 \
    -processor org.checkerframework.checker.nullness.NullnessChecker \
    src/sandbox/FileToCheck.java
```

23.9 javac implementation survival guide

Since this section of the manual was written, the useful "The Hitchhiker's Guide to javac" has become available at `http://openjdk.java.net/groups/compiler/doc/hhgtjavac/index.html`. See it first, and then refer to this section. (This section of the manual should be revised, or parts eliminated, in light of that document.)

A checker built using the Checker Framework makes use of a few interfaces from the underlying compiler (Oracle's OpenJDK javac). This section describes those interfaces.

23.9.1 Checker access to compiler information

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

Types - Java Language Model API

A `TypeMirror` represents a Java type.

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

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

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

Elements - Java Language Model API

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

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

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

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

Trees - Compiler Tree API

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

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

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

Using the APIs

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

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

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

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

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

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

The most important methods of `AbstractTypeProcessor` are `typeProcess` and `getSupportedSourceVersion`. The former class is where you would insert any sort of method call to walk the AST, and the latter just returns a constant indicating that we are targeting version 8 of the compiler. Implementing these two methods should be enough for a basic plugin; see the Javadoc for the class for other methods that you may find useful later on.

The Checker Framework uses Oracle's Tree API to access a program's AST. The Tree API is specific to the Oracle OpenJDK, so the Checker Framework only works with the OpenJDK javac, not with Eclipse's compiler ecj or with gcj. This also limits the tightness of the integration of the Checker Framework into other IDEs such as IntelliJ IDEA. An implementation-neutral API would be preferable. In the future, the Checker Framework can be migrated to use the Java Model AST of JSR 198 (Extension API for Integrated Development Environments) [Cro06], which gives access to the source code of a method. But, at present no tools implement JSR 198. Also see Section 23.5.1.

Learning more about javac

Sun's javac compiler interfaces can be daunting to a newcomer, and its documentation is a bit sparse. The Checker Framework aims to abstract a lot of these complexities. You do not have to understand the implementation of javac

to build powerful and useful checkers. Beyond this document, other useful resources include the Java Infrastructure Developer's guide at http://wiki.netbeans.org/Java_DevelopersGuide and the compiler mailing list archives at http://news.gmane.org/gmane.comp.java.openjdk.compiler.devel (subscribe at http://mail.openjdk.java.net/mailman/listinfo/compiler-dev).

Chapter 24

Integration with external tools

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

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

This chapter also discusses type inference tools (see Section 24.8).

All examples in this chapter are in the public domain, with no copyright nor licensing restrictions.

24.1 Javac Compiler

If you use `javac` compiler from the command line, then you can instead use the Checker Framework compiler that is bundled with the Checker Framework. The bundled `javac` is a variant of the OpenJDK `javac` that recognizes type annotations. Eventually, the OpenJDK `javac` will recognize type annotations. The bundled `javac` also supports annotations in comments (see Section 21.3.1), which the OpenJDK `javac` will not.

This section describes how you can install and use the bundled `javac`, using either Unix/Linux/MacOS (see Section 24.1.1) or Windows (see Section 24.1.2). The instructions are identical to those of Section 1.3, but are given as commands that you can cut and paste into your command shell.

24.1.1 Unix/Linux/MacOS installation

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

1. Download the latest Checker Framework distribution and unzip it. You can put it anywhere you like by changing the definition of environment variable `JSR308` below; a standard place is in a new directory named `jsr308`.

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

   The output of the last command should be:

   ```
   javac 1.7.0-jsr308-1.8.0
   ```

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

```
export JSR308=$HOME/jsr308
export CHECKERFRAMEWORK=$JSR308/checker-framework
export PATH=$CHECKERFRAMEWORK/checker/bin:${PATH}
```

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

```
javac -version
```

The output should be:

```
javac 1.7.0-jsr308-1.8.0
```

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

24.1.2 Windows installation

1. Download the latest Checker Framework distribution and unzip it to create a `checkers` directory. You can put it anywhere you like; a standard place is in a new directory under `C:\Program Files`.
   (a) Save the file `http://types.cs.washington.edu/checker-framework/current/checker-framework.zip` to your Desktop.
   (b) Double-click the `checker-framework.zip` file on your computer. Click on the `checkers` directory, then Select `Extract all files`, and use `C:\Program Files` as the destination. You will obtain a new `C:\Program Files\checker-framework` folder.
   (c) Verify that the installation works. From a Windows command prompt, run:
   ```
   set CHECKERFRAMEWORK = C:\Program Files\checker-framework\
   java -jar C:%CHECKERFRAMEWORK%\checker\dist\checker.jar -version
   ```
   The output should be:
   ```
   javac 1.7.0-jsr308-1.8.0
   ```
2. In order to use the updated compiler when you type `javac`, add the directory `C:\Program Files\checker-framework\checkers\binary` to the beginning of your path variable. Also set a `CHECKERFRAMEWORK` variable.
   To set an environment variable, you have two options: make the change temporarily or permanently.
   - To make the change **temporarily**, type at the command shell prompt:
     `path = newdir;%PATH%`
     For example:
     ```
     set CHECKERFRAMEWORK = C:\Program Files\checker-framework
     path = %CHECKERFRAMEWORK%\checker\bin;%PATH%
     ```
     This is a temporary change that endures until the window is closed, and you must re-do it every time you start a new command shell.
   - To make the change **permanently**, Right-click the `My Computer` icon and select `Properties`. Select the `Advanced` tab and click the `Environment Variables` button. You can set the variable as a "System Variable" (visible to all users) or as a "User Variable" (visible to just this user). Both work; the instructions below show how to set as a "System Variable". In the `System Variables` pane, select `Path` from the list and click `Edit`. In the `Edit System Variable` dialog box, move the cursor to the beginning of the string in the `Variable Value` field and type the full directory name (not using the `%CHECKERFRAMEWORK%` environment variable) followed by a semicolon (`;`).
     Similarly, set the `CHECKERFRAMEWORK` variable.
     This is a permanent change that only needs to be done once ever.

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

   ```
   javac -version
   ```

   The output should be:

   ```
   javac 1.7.0-jsr308-1.8.0
   ```

24.2 Ant task

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

1. Set the `jsr308javac` property:

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

<property name="checkerframework" value="${env.CHECKERFRAMEWORK}" />

<!-- On Mac/Linux, use the javac shell script; on Windows, use javac.bat -->
<condition property="cfJavac" value="javac.bat" else="javac">
    <os family="windows" />
</condition>

<presetdef name="jsr308.javac">
  <javac fork="yes" executable="${checkerframework}/checker/bin/${cfJavac}" >
    <!-- JSR-308-related compiler arguments -->
    <compilerarg value="-version"/>
    <!-- optional, so .class files work with older JVMs: <compilerarg line="-target 5"/> -->
    <compilerarg value="-implicit:class"/>
  </javac>
</presetdef>
```

2. **Duplicate** the compilation target, then **modify** it slightly as indicated in this example:

```
<target name="check-nullness"
        description="Check for null pointer dereferences"
        depends="clean,...">
  <!-- use jsr308.javac instead of javac -->
  <jsr308.javac ... >
    <compilerarg line="-processor org.checkerframework.checker.nullness.NullnessChecker"/>
    <!-- optional, for implicit imports: <compilerarg value="-J-Djsr308_imports=org.checkerframework.checker.nullness.qual.*:
    <!-- optional, to not check uses of library methods: <compilerarg value="-AskipUses=^(java\.awt\.|javax\.swing\.)"/> -->
    <compilerarg line="-Xmaxerrs 10000"/>
    ...
  </jsr308.javac>
</target>
```

   Fill in each ellipsis (...) from the original compilation target. But, don't include any `-source` argument with value other than `1.8` or `8`. Doing so will disable the annotations in comments feature (see Section 21.3.1, page 112).
   In the example, the target is named `check-nullness`, but you can name it whatever you like.

24.2.1 Explanation

This section explains each part of the Ant task.

1. Definition of `jsr308.javac`:
   The `fork` field of the `javac` task ensures that an external javac program is called. Otherwise, Ant will run javac via a Java method call, and there is no guarantee that it will get the JSR 308 version that is distributed with the Checker Framework.
   The `-version` compiler argument is just for debugging; you may omit it.
   The `-target 5` compiler argument is optional, if you use Java 5 in ordinary compilation when not performing pluggable type-checking (see Section 21.3.4, page 114).
   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.

24.3 Maven plugin

If you use the Maven tool, then you can specify pluggable type-checking as part of your build process.

1. Add the repositories to your `pom.xml` file:

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

   As an alternative to the repository, you can find the Checker Framework in Maven Central (MC): `http://search.maven.org/#search|ga|1|a%3A%22checker-framework%22`
2. Declare a dependency on the type qualifier annotations. Find the existing `<dependencies>` section and add a new `<dependencies>` item:

```
<dependencies>
    ... existing <dependency> items ...

    <!-- annotations from the Checker Framework: nullness, interning, locking, ... -->
    <dependency>
        <groupId>org.checkerframework</groupId>
        <artifactId>checker-qual</artifactId>
        <version>1.8.0</version>
    </dependency>

</dependencies>
```

3. Attach the plugin to your build lifecycle by adding a new `<plugin>` item within `<plugins>`, or a new `<plugins>` item if necessary.

```
<build>
    <plugins>
        ... existing <plugin> items ...

        <plugin>
            <groupId>org.checkerframework</groupId>
            <artifactId>checkerframework-maven-plugin</artifactId>
            <version>1.8.0</version>
            <executions>
                <execution>
                    <!-- run the checkers after compilation; this can also be any later phase -->
```

```
                    <phase>process-classes</phase>
                    <goals>
                        <goal>check</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <!-- required configuration options -->
                <!-- a list of processors to run -->
                <processors>
                    <processor>org.checkerframework.checker.nullness.NullnessChecker</processor>
                    <processor>org.checkerframework.checker.interning.InterningChecker</processor>
                </processors>
                <!-- optional configuration options go here -->
            </configuration>
        </plugin>
    </plugins>
</build>
```

The above example attaches the "check" goal to the "process-lifecycle" phase which occurs directly after the "compile" phase (see `http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html`). This means that any maven target that exercises that phase will also run the Checker Framework (e.g., `mvn package`, `mvn verify`).

To run only checking: `mvn checkerframework:check`

4. Add optional parameters

   Within the configuration element you can add a number of parameters.

   ```
   <procOnly>false</procOnly>
   ```

   By default, the Checker Framework Maven plugin will only check source files but will not produce class files for the source files it checks (see `-proc:only` at `http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/javac.html`). Setting `procOnly` to false will cause the Checker Framework compiler to generate class files. Note: If your Maven project's output directory has not been created then the plugin will attempt to create it and all required directories preceding it. The check task will fail in the event that these directories cannot be created.

   You may want to check only a subset of source files in your project. To do so you can use the `includes` and `excludes` elements using the standard Maven syntax. The semantics of including/excluding files are also described by the DirectoryScanner documentation found at: `http://plexus.codehaus.org/plexus-utils/apidocs/org/codehaus/plexus/util/DirectoryScanner.html` (see setIncludes/setExcludes)

   ```
   <!-- a list of patterns to include, in the standard maven syntax; defaults to **/*.java -->
   <includes>
       <include>org/company/important/**/*.java</include>
   </includes>
   <!-- a list of patterns to exclude, in the standard maven syntax; defaults to an empty list -->
   <excludes>
       <exclude>org/company/notimportant/**/*.java</exclude>
   </excludes>
   ```

   By default, an error reported by the Checker Framework Maven plugin will cause your build to fail. Maven also occasionally causes a build to fail when it encounters a message it does not recognize, even if that message is not an error or a warning. To prevent a build from failing and still produce error and warning output, set `failOnError` to false.

   ```
   <failOnError>false</failOnError>
   ```

Errors reported by the Checker Framework Maven plugin use a similar format to the Maven compiler plugin. If you want to see the exact output of the Checker Framework compiler instead, use the following option:

```
<useJavacOutput>true</useJavacOutput>
```

You may want to run the Checker Framework using a Java version that differs from the version running Maven. The `executable` element specifies a Java executable to use when running the Checker Framework.

```
<executable>/usr/bin/java</executable>
```

The version of the Checker Framework run by the Maven plugin will share the same version as the Maven plugin by default. That is, if you are using the Checker Framework Maven plugin version 1.8.0 then it will run the Checker Framework version 1.8.0 when checking source code. However, you can specify an earlier or later version of the Checker Framework using the `checkerFrameworkVersion` element. Note: The Checker Framework Maven plugin will only work with Checker Framework versions 1.5.0 and higher.

```
<checkerFrameworkVersion>1.6.2-johndoe</checkerFrameworkVersion>
```

To disable the Checker Framework check without removing it from your pom file you can use the `skip` option.

```
<skip>true</skip>
```

Finally, the Checker Framework compiler is an extension of the javac compiler and therefore excepts any parameter that can be provided to javac or the JVM. To provide javac parameters you can use the `javacParams` element. To provide JVM arguments you can use the `javaParams` element.

```
<!-- additional parameters passed to the Java compiler -->
<javacParams>-Alint</javacParams>
<!-- additional parameters to pass to the forked JVM -->
<javaParams>-Xdebug</javaParams>
```

You can find an example Maven project that uses the Checker Framework Maven plugin at: `http://types.cs.washington.edu/checker-framework/current/mvn-examples.zip`

The plugin was contributed by Adam Warski.

24.4 Gradle

If you fork the compilation task, Gradle lets you specify the executable to compile java programs.

To specify the appropriate executable, set `options.fork = true` and `compile.options.fork.executable = "$CHECKERFRAMEWORK/checker/bin/javac"`

To specify command-line arguments, set `compile.options.compilerArgs`. Here is a possible example:

```
allprojects {
  tasks.withType(JavaCompile).all { JavaCompile compile ->
    compile.options.debug = true
    compile.options.compilerArgs = [
      '-version',
      '-implicit:class',
      '-processor', 'org.checkerframework.checker.nullness.NullnessChecker'
    ]
    options.fork = true
    options.forkOptions.executable = "$CHECKERFRAMEWORK/checker/bin/javac"
  }
}
```

24.5 IntelliJ IDEA

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

24.6 Eclipse

Eclipse supports type annotations. It does not enable running the Checker Framework, nor is it necessary for running the Checker Framework.

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

No matter what method you choose, we suggest that all Checker Framework annotations be written in the comments if you are using a version of Eclipse that does not support Java 8. This will avoid many text highlighting errors with versions of Eclipse that don't support Java 8 and the new JSR 308 syntax changes.

Even in a version of Eclipse that supports Java 8's type annotations, you still need to run the Checker Framework via Ant or via the plug-in, rather than by supplying the `-processor` command-line option to the ejc compiler. The reason is that the Checker Framework is built upon javac, and ejc represents the Java program differently. (If both javac and ejc implemented JSR 198 [Cro06], then it would be possible to build type-checking plug-ins that work with both compilers.)

Using an Ant task Add an Ant target as described in Section 24.2. You can run the Ant target by executing the following steps (instructions copied from `http://www.eclipse.org/documentation/?topic=/org.eclipse.platform.doc.user/gettingStarted/qs-84_run_ant.htm`):

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

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

24.7 tIDE

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

24.8 Type inference tools

24.8.1 Varieties of type inference

There are two different tasks that are commonly called "type inference".

1. Type inference during type-checking (Section 20.4): During type-checking, if certain variables have no type qualifier, the type-checker determines whether there is some type qualifier that would permit the program to type-check. If so, the type-checker uses that type qualifier, but never tells the programmer what it was. Each time the type-checker runs, it re-infers the type qualifier for that variable. If no type qualifier exists that permits the program to type-check, the type-checker issues a type warning.

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

- If the type qualifier is obvious to the programmer, then omitting it can reduce annotation clutter in the program.
- The type inference can take advantage of only the code currently being compiled, rather than having to be correct for all possible calls. Additionally, if the code changes, then there is no old annotation to update.

2. Type inference to annotate a program (Section 24.8.2): As a separate step before type-checking, a type inference tool takes the program as input, and outputs a set of type qualifiers that would type-check. These qualifiers are inserted into the source code or the class file. They can be viewed and adjusted by the programmer, and can be used by tools such as the type-checker.
   This variety of type inference must be provided by a separate tool. It is not built into the Checker Framework.
   Advantages of this variety of type inference include:

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

Advantages of both varieties of inference include:

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

Each variety of type inference has its place. When using the Checker Framework, type inference during type-checking is performed only *within* a method (Section 20.4). Every method signature (arguments and return values) and field must be explicitly annotated, either by the programmer or by a separate type-checking tool (Section 24.8.2). This choice reduces programmer effort (typically, a programmer does not have to write any qualifiers inside the body of a method) while still retaining modular checking and documentation benefits.

24.8.2 Type inference to annotate a program

This section lists tools that take a program and output a set of annotations for it.
Section 3.3.7 lists several tools that infer annotations for the Nullness Checker.
Section 16.2.2 lists a tool that infers annotations for the Javari Checker, which detects mutation errors.

Chapter 25

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:

25.1 Motivation for pluggable type-checking

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

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

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

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

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

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

25.2 Getting started

25.2.1 How do I get started annotating an existing program?

See Section 2.4.1.

25.2.2 Which checker should I start with?

You should start with a property that matters to you. Think about what aspects of your code cause the most errors, or cost the most time during maintenance, or are the most common to be incorrectly-documented. Focusing on what you care about will give you the best benefits.

When you first start out with the Checker Framework, it's usually best to get experience with an existing type-checker before you write your own new checker.

Many users are tempted to start with the Nullness Checker (see Chapter 3, page 22), 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.

The Linear Checker (see Chapter 14, page 77) has not been extensively tested. The IGJ Checker (see Chapter 15, page 79), Javari Checker (see Chapter 16, page 83), and typestate checkers (see Chapter 18.1, page 88) have known bugs that limit their usability. (Report the ones that affect you, and the Checker Framework developers will prioritize fixing them.)

25.3 Usability of pluggable type-checking

25.3.1 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.

25.3.2 Will my code become cluttered with type annotations?

In summary: annotations do not clutter code; they are used much less frequently than generic types, which Java programmers find acceptable; and they reduce the overall volume of documentation that a codebase needs.

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

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

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

Each annotation that a programmer writes replaces a sentence or phrase of English descriptive text that would otherwise have been written in the Javadoc. So, use of annotations actually reduces the overall size of the documentation, at the same time as making it machine-processable and less ambiguous.

25.3.3 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 Type Annotations compiler emits the identical bytecodes as the Java 8 compiler and so there is no run-time effect. Because there is no run-time representation of type qualifiers, there is no way to use reflection to query the qualifier on a given object, though you can use reflection to examine a class/method/field declaration.

Using the Checker Framework does increase compilation time. In theory it should only add a few percent overhead, but our current implementation can double the compilation time — or more, if you run many pluggable type-checkers at once. This is especially true if you run pluggable type-checking on every file (as we recommend) instead of just on the ones that have recently changed. Nonetheless, compilation with pluggable type-checking still feels like compilation, and you can do it as part of your normal development process.

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

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

25.4 How handle warnings and errors

25.4.1 What should I do if a checker issues a warning about my code?

For a discussion of this issue, see Section 2.4.7.

25.4.2 What does a certain Checker Framework warning message mean?

Search through this manual for the text of the warning message. Oftentimes the manual explains it. If not, ask on the mailing list.

25.4.3 Can a pluggable type-checker guarantee that my code is correct?

Each checker looks for certain errors. You can use multiple checkers to detect more errors in your code, but you will never have a guarantee that your code is completely bug-free.

If the type-checker issues no warning, then you have a guarantee that your code is free of some particular error. There are some limitations to the guarantee.

Most importantly, if you run a pluggable checker on only part of a program, then you only get a guarantee that those parts of the program are error-free. For example, suppose you have type-checked a framework that clients are intended to extend. You should recommend that clients run the pluggable checker. There is no way to force users to do so, so you may want to retain dynamic checks or use other mechanisms to detect errors.

Section 2.3 states other limitations to a checker's guarantee, such as regarding concurrency. Java's type system is also unsound in certain situations, such as for arrays and casts (however, the Checker Framework is sound for arrays and casts). Java uses dynamic checks is some places it is unsound, so that errors are thrown at run time. The pluggable type-checkers do not currently have built-in dynamic checkers to check for the places they are unsound. Writing dynamic checkers would be an interesting and valuable project.

Other types of dynamism in a Java application do not jeopardize the guarantee, because the type-checker is conservative. For example, at a method call, dynamic dispatch chooses some implementation of the method, but it is impossible to know at compile time which one it will be. The type-checker gives a guarantee no matter what implementation of the method is invoked.

Even if a pluggable checker cannot give an ironclad guarantee of correctness, it is still useful. It can find errors, exclude certain types of possible problems (e.g., restricting the possible class of problems), improve documentation, and increase confidence in your software.

25.4.4 What guarantee does the Checker Framework give for concurrent code?

The Lock Checker (see Chapter 5) 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 warn about a possible null dereference. For example, the Nullness Checker issues no warning for this code:

```
if (myobject.myfield != null) {
  myobject.myfield.toString();
}
```

This code is safe when run on its own. However, in the presence of multithreading, the call to `toString` may fail because another thread may set `myobject.myfield` to `null` after the nullness check in the `if` condition, but before the `if` body is executed.

If you supply the `-AconcurrentSemantics` command-line option, then the Checker Framework assumes that any field can be changed at any time. This limits the amount of flow-sensitive type qualifier refinement (Section 20.4) that the Checker Framework can do.

25.4.5 How do I make compilation succeed even if a checker issues errors?

Section 2.2 describes the `-Awarns` command-line option that turns checker errors into warnings, so type-checking errors will not cause `javac` to exit with a failure status.

25.4.6 Why does the checker always say there are 100 errors or warnings?

By default, javac only reports the first 100 errors or warnings. Furthermore, once javac encounters an error, it doesn't try compiling any more files (but does complete compilation of all the ones that it has started so far).

To see more than 100 errors or warnings, use the javac options `-Xmaxerrs` and `-Xmaxwarns`. To convert Checker Framework errors into warnings so that javac will process all your source files, use the option `-Awarns`. See Section 2.2 for more details.

25.4.7 Why does the Checker Framework report an error regarding a type I have not written in my program?

Sometimes, a Checker Framework warning message will mention a type you have not written in your program. This is typically because a default has been applied where you did not write a type; see Section 20.3.1. In other cases, this is because flow-sensitive type refinement has given an expression a more specific type than you wrote or than was defaulted; see Section 20.4.

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

Some properties are not checked statically (see Chapter 21 for reasons that code might not be statically checked). In such cases, it would be desirable to check the property dynamically, at run time. Currently, the Checker Framework has no support for adding code to perform run-time checking.

Adding such support would be an interesting and valuable project. An example would be an option that causes the Checker Framework to automatically insert a run-time check anywhere that static checking is suppressed. If you are able to add run-time verification functionality, we would gladly welcome it as a contribution to the Checker Framework.

Some checkers have library methods that you can explicitly insert in your source code. Examples include the Nullness Checker's `NullnessUtils.castNonNull` method (see Section 3.4.1) and the Regex Checker's `RegexUtil` class (see Section 8.2.4). But, it would be better to have more general support that does not require the user to explicitly insert method calls.

25.5 Syntax of type annotations

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

25.5.1 What is a "receiver"?

The *receiver* of a method is the `this` formal parameter, sometimes also called the "current object". Within the method declaration, `this` is used to refer to the receiver formal parameter. At a method call, the receiver actual argument is written before the method name.

The method `compareTo` takes *two* formal parameters. At a call site like `x.compareTo(y)`, the two arguments are `x` and `y`. It is desirable to be able to annotate the types of both of the formal parameters, and doing so is supported by both Java's type annotations syntax and by the Checker Framework.

A type annotation on the receiver is treated exactly like a type annotation on any other formal parameter. At each call site, the type of the argument must be a consistent with (a subtype of or equal to) the declaration of the corresponding formal parameter. If not, the type-checker issues a warning.

Here is an example. Suppose that `@A Object` is a supertype of `@B Object` in the following declaration:

```
class MyClass {
  void requiresA(@A MyClass this) { ... }
  void requiresB(@B MyClass this) { ... }
}
```

Then the behavior of four different invocations is as follows:

```
@A MyClass myA = ...;
@B MyClass myB = ...;

myA.requiresA()    // OK
myA.requiresB()    // compile-time error
myB.requiresA()    // OK
myB.requiresB()    // OK
```

The invocation `myA.requiresB()` does not type-check because the actual argument's type is not a subtype of the formal parameter's type.

A top-level constructor does not have a receiver. An inner class constructor does have a receiver, whose type is the same as the containing outer class. The receiver is distinct from the object being constructed. In a method of a top-level class, the receiver is named `this`. In a constructor of an inner class, the receiver is named `Outer.this` and the result is named `this`.

25.5.2 What is the meaning of an annotation after a type, such as `@NonNull Object @Nullable`?

In a type such as `@NonNull Object @Nullable []`, it may appear that the `@Nullable` annotation is written *after* the type `Object`. In fact, `@Nullable` modifies `[]`. See the next FAQ, about array annotations (Section 25.5.3).

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

You should parse this as: (**`@NonNull Object`**) (**`@Nullable []`**). Each annotation precedes the component of the type that it qualifies.

Thus, `@NonNull Object @Nullable []` is a possibly-null array of non-null objects. Note that the first token in the type, "`@NonNull`", applies to the element type `Object`, not to the array type as a whole. The annotation `@Nullable` applies to the array (`[]`).

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

Some older tools interpret a declaration like `@NonEmpty String[] var` as "non-empty array of strings". This is in conflict with the Java type annotations specification, which defines it as meaning "array of non-empty strings". If you use one of these older tools, you will find this incompatibility confusing. You will have to live with it until the older tool is updated to conform to the Java specification, or until you transition to a newer tool that conforms to the Java specification.

25.5.4 What is the meaning of a type qualifier at a class declaration?

Writing an annotation on a class declaration makes that annotation implicit for all uses of the class (see Section 20.3). If you write `class @MyQual MyClass { ... }`, then every unannotated use of `MyClass` is `@MyQual MyClass`. A user is permitted to strengthen the type by writing a more restrictive annotation on a use of MyClass, such as `@MyMoreRestrictiveQual MyClass`.

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

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

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

> "Each `Widget` type may have a `@Version` annotation. I wish to prove that versions of widgets don't get assigned to incompatible variables, and that older code does not call newer code (to avoid problems when backporting).
>
> A `@Version` annotation could be written like so:
>
> ```
> @Version("2.0") Widget createWidget(String value) { ... }
> ```
>
> `@Version("2.0")` on the method could mean that the `createWidget` method only appears in the 2.0 version. `@Version("2.0")` on the return type could mean that the returned `Widget` should only be used by code that uses the 2.0 API of `Widget`. It should be possible to specify these independently, such as a 2.0 method that returns a value that allows the 1.0 API method invocations."

Both of these are type properties and should be specified with type annotations. No method annotation is necessary or desirable. The best way to require that the receiver has a certain property is to use a type annotation on the receiver of the method. (Slightly more formally, the property being checked is compatibility between the annotation on the type of the formal parameter receiver and the annotation on the type of the actual receiver.) If you do not know what "receiver" means, see the next question.

Another example of a type-and-declaration annotation that represents poor design is JCIP's `@GuardedBy` annotation [GPB+06]. As discussed in Section 5.2.1, it means two different things when applied to a field or a method. To reduce confusion and increase expressiveness, the Lock Checker (see Chapter 5) uses the `@Holding` annotation for one of these meanings, rather than overloading `@GuardedBy` with two distinct meanings.

25.6 Semantics of type annotations

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

25.6.2 How can I handle typestate, or phases of my program with different data properties?

Sometimes, your program works in phases that have different behavior. For example, you might have a field that starts out null and becomes non-null at some point during execution, such as after a method is called. You can express this property as follows:

1. Annotate the field type as `@MonotonicNonNull`.
2. Annotate the method that sets the field as `@EnsuresNonNull("myFieldName")`. (If method `m1` calls method `m2`, which actually sets the field, then you would probably write this annotation on both `m1` and `m2`.)
3. Annotate any method that depends on the field being non-null as `@RequiresNonNull("myFieldName")`. The type-checker will verify that such a method is only called when the field isn't null — that is, the method is only called after the setting method.

You can also use a typestate checker (see Chapter 18.1, page 88), but they have not been as extensively tested.

25.6.3 Why are explicit and implicit bounds defaulted differently?

The following two bits of code have the same semantics under Java, but are treated differently by the Checker Framework's CLIMB-to-top defaulting rules (Section 20.3.2):

```
class MyClass<T> { ... }
class MyClass<T extends Object> { ... }
```

The difference is the annotation on the upper bound of the type argument `T`. They are treated in the following.

```
class MyClass<T>  ==  class MyClass<T extends @TOPTYPEANNO Object> { ... }
class MyClass<T extends Object>  ==  class MyClass<T extends @DEFAULTANNO Object>
```

`@TOPTYPEANNO` is the top annotation in the type qualifier hierarchy. For example, for the nullness type system, the top type annotation is `@Nullable`; as shown in Figure 3.1. `@DEFAULTANNO` is the default annotation for the type system. For example, for the nullness type system, the default type annotation is `@NonNull`.

In some type systems, the top qualifier and the default are the same. For such type systems, the two code snippets shown above are treated the same. An example is the regular expression type system; see Figure 8.1.

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 consistenly with all other types in this location and all other occurrences of `Object` in the program.

It is uncommon for a user to write `Object` as an upper bound with no type qualifier: `class C<T extends Object> { ... }`. It is better style to write no upper bound or to write an explicit type annotation on `Object`.

25.7 Creating a new checker

25.7.1 How do I create a new checker?

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

Chapter 23 gives complete details regarding how to write a checker. It also suggests places to look for more help, such as the Checker Framework API documentation (Javadoc) and the source code of the distributed checkers.

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

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

Section 17.2 explains this checker and tells you how to run it.

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

A type system implementer can declaratively specify the type qualifier hierarchy (Section 23.3.1) and the type introduction rules (Section 23.4.1). However, the Checker Framework uses a procedural syntax for specifying type-checking rules (Section 23.5). A declarative syntax might be more concise, more readable, and more verifiable than a procedural syntax.

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

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

25.8 Relationship to other tools

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

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

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

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

Each tool is useful in different circumstances, depending on how important your code is and your desired level of confidence in your code. For more details on the comparison, see section 26.5. For a case study that compared the nullness analysis of FindBugs, Jlint, PMD, and the Checker Framework, see section 6 of the paper "Practical pluggable types for Java" [PAC+08].

25.8.2 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.

25.8.3 Is the Checker Framework an official part of Java?

The Checker Framework is not an official part of Java. The Checker Framework relies on type annotations, which are part of Java 8. See the Type Annotations (JSR 308) FAQ for more details.

25.8.4 What is the relationship between the Checker Framework and JSR 305?

JSR 305 aimed to define official Java names for some annotations, such as `@NonNull` and `@Nullable`. However, it did not aim to precisely define the semantics of those annotations nor to provide a reference implementation of an annotation processor that validated their use.

By contrast, the Checker Framework precisely defines the meaning of a set of annotations and provides powerful type-checkers that validate them. However, the Checker Framework is not an official part of the Java language; it chooses one set of names, but another tool might choose other names.

JSR 305 has been abandoned; there has been no activity by its expert group since 2009. In the future, the Java Community Process might standardize the names and meanings of specific annotations, after there is more experience with their use in practice.

The Checker Framework defines annotations `@NonNull` and `@Nullable` that are compatible with annotations defined by JSR 305, FindBugs, IntelliJ, and other tools; see Section 3.7.

25.8.5 What is the relationship between the Checker Framework and JSR 308?

JSR 308, also known as the Type Annotations specification, dictates the syntax of type annotations in Java SE 8: how they are expressed in the Java language.

JSR 308 does not define any type annotations such as `@NonNull`, and it does not specify the semantics of any annotations. Those tasks are left to third-party tools. The Checker Framework is one such tool.

The Checker Framework makes use of Java SE 8's type annotation syntax, but the Checker Framework can be used with previous versions of the Java language via the annotations-in-comments feature (Section 21.3.1).

Chapter 26

Troubleshooting and getting help

The manual might already answer your question, so first please look for your answer in the manual, including this chapter and the FAQ (Chapter 25). If not, you can use the mailing list, `checker-framework-discuss@googlegroups.com`, to ask other users for help. For archives and to subscribe, see `http://groups.google.com/group/checker-framework-discuss`. To report bugs, please see Section 26.2. If you want to help out, you can give feedback (including on the documentation), choose a bug and fix it, or select a project from the ideas list at `http://code.google.com/p/checker-framework/wiki/Ideas`.

26.1 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.

26.1.1 Unable to run the checker, or checker crashes

If you are unable to run the checker, or if the checker or the compiler terminates with an error, then the problem may be a problem with your environment. (If the checker or the compiler crashes, that is a bug in the Checker Framework; please report it. See Section 26.2.) This section describes some possible problems and solutions.

- If you get the error

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

  then you are using the source installation and file `tools.jar` is not on your classpath. See the installation instructions (Section 1.3).
- If you get an error such as

  ```
  package org.checkerframework.checker.nullness.qual does not exist
  ```

  despite no apparent use of `import org.checkerframework.checker.nullness.qual.*;` in the source code, then perhaps `jsr308_imports` is set as a Java system property, a shell environment variable, or a command-line option (see Section 21.3.2). You can solve this by unsetting the variable/option, or by ensuring that the `checker.jar` file is on your classpath.
  If the error is

  ```
  package org.checkerframework.checker.nullness.qual does not exist
  ```

  (note the extra apostrophe!), then you have probably misused quoting when supplying the `jsr308_imports` environment variable.
- If you get an error like one of the following,

  ```
  ...\build.xml:59: Error running ${env.CHECKERFRAMEWORK}\checker\bin\javac.bat compiler
  ```

```
.../bin/javac: Command not found
```

then the problem may be that you have not set the `CHECKERFRAMEWORK` environment variable, as described in Section 24.1. Or, maybe you made it a user variable instead of a system variable.

- If you get one of these errors:

```
The hierarchy of the type ClassName is inconsistent

The type com.sun.source.util.AbstractTypeProcessor cannot be resolved.
  It is indirectly referenced from required .class files
```

then you are likely NOT using the Checker Framework compiler. Use either `$CHECKERFRAMEWORK/checker/bin/javac` or "`java -jar $CHECKERFRAMEWORK/checker/dist/checker.jar`".

- If you get the error

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

If you get an error such as

```
java.lang.NoClassDefFoundError: java/util/Objects
```

then you are trying to run the compiler using a JDK 6 or earlier JVM. Install and use a Java 7 or 8 JDK, at least for running the Checker Framework.
then an annotation is not present at run time that was present at compile time. For example, maybe when you compiled the code, the `@Nullable` annotation was available, but it was not available at run time. You can use JDK 8 at run time, or compile with a Java 7 compiler that will ignore the annotations in comments.

- A "class file for ... not found" error, especially for an inner class in the JDK, is probably due to a JDK version mismatch.
  In general, Java issues a "class file for ... not found" error when your classpath contains code that was compiled with some library, but your classpath does not contain that library itself.
  For example, suppose that when you run the compiler, you are using JDK 8, but some library on your classpath was compiled against JDK 7, and the compiled library refers to a class that only appears in JDK 7. (If only one version of Java existed, or the Checker Framework didn't try to support multiple different versions of Java, this would not be a problem.)
  Examples of classes that were in JDK 7 but were removed in JDK 8 include:

```
class file for java.util.TimeZone$DisplayNames not found
```

Examples of classes that were in JDK 6 but were removed in JDK 7 include:

```
class file for java.io.File$LazyInitialization not found
class file for java.util.Hashtable$EmptyIterator not found
java.lang.NoClassDefFoundError: java/util/Hashtable$EmptyEnumerator
```

Examples of classes that were not in JDK 7 but were introduced in JDK 8 include:

```
The type java.lang.Class$ReflectionData cannot be resolved
```

Examples of classes that were not in JDK 6 but were introduced in JDK 7 include:

```
class file for java.util.Vector$Itr not found
```

You may be able to solve the problem by running

```
cd checker
ant jdk.jar bindist
```

to re-generate files `checker/jdk/jdk{7,8}.jar` and `checker/bin/jdk{7,8}.jar`.

- A `NoSuchFieldError` such as this:

```
java.lang.NoSuchFieldError: NATIVE_HEADER_OUTPUT
```

Field `NATIVE_HEADER_OUTPUT` was added in JDK 8. The error message suggests that you're not executing with the right bootclasspath: some classes were compiled with the JDK 8 version and expect the field, but you're executing the compiler on a JDK without the field.

One possibility is that you are not running the Checker Framework compiler — use `javac -version` to check this, then use the right one. (Maybe the Checker Framework javac is at the end rather than the beginning of your path.)
If you are using Ant, then one possibility is that the javac compiler is using the same JDK as Ant is using. You can correct this by being sure to use `fork="yes"` (see Section 24.2) and/or setting the `build.compiler` property to `extJavac`.
If you are building from source, you might need to rebuild the Annotation File Utilities before recompiling or using the Checker Framework.

- If you get an error that contains lines like these:

```
Caused by: java.util.zip.ZipException: error in opening zip file
        at java.util.zip.ZipFile.open(Native Method)
        at java.util.zip.ZipFile.<init>(ZipFile.java:131)
```

  then one possibility is that you have installed the Checker Framework in a directory that contains special characters that Java's ZipFile implementation cannot handle. For instance, if the directory name contains "+", then Java 1.6 throws a ZipException, and Java 1.7 throws a FileNotFoundException and prints out the directory name with "+" replaced by blanks.
- If you get an error

```
error: scoping construct for static nested type cannot be annotated
```

  then you have probably written something like `@Nullable java.util.List`. The correct syntax is `java.util.@Nullable List`. But, it's usually better to add `import java.util.List` to your source file, so that you can just write `@Nullable List`. Likewise, you must write `Outer.@Nullable StaticNestedClass` rather than `@Nullable Outer.StaticNestedClass`.
  Java 8 requires that a type qualifier be written directly on the type that it qualifies, rather than on a scoping mechanism that assists in resolving the name. Examples of scoping mechanisms are package names and outer classes of static nested classes.
  The reason for the Java 8 syntax is to avoid syntactic irregularity. When writing a member nested class (also known as an inner class), it is possible to write annotations on both the outer and the inner class: `@A1 Outer.@A2 Inner`. Therefore, when writing a static nested class, the annotations should go on the same place: `Outer.@A3 StaticNested` (rather than `@ConfusingAnnotation Outer.Nested` where `@ConfusingAnnotation` applies to `Outer` if `Nested` is a member class and applies to `Nested` if `Nested` is a static class). It's not legal to write an annotation on the outer class of a static nested class, because neither annotations nor instantiations of the outer class affect the static nested class.
  Similar arguments apply when annotating `package.Outer.Nested`.

26.1.2 Unexpected type-checking results

This section describes possible problems that can lead the type-checker to give unexpected results.

- If the Checker Framework is unable to verify a property that you know is true, then it is helpful to formulate an argument about why the property is true. Recall that the Checker Framework does modular verification, one procedure at a time; it observes the specifications, but not the implementations, of other methods.
  If any aspects of your argument are not expressed as annotations, then you may need to write more annotations.
  If any aspects of your argument are not expressible as annotations, then you may need to extend the type-checker.
- If a checker seems to be ignoring the annotation on a method, then it is possible that the checker is reading the method's signature from its `.class` file, but the `.class` file was not created by the JSR 308 compiler. You can check whether the annotations actually appear in the `.class` file by using the `javap` tool.
  If the annotations do not appear in the `.class` file, here are two ways to solve the problem:
  - Re-compile the method's class with the Type Annotations compiler. This will ensure that the type annotations are written to the class file, even if no type-checking happens during that execution.
  - Pass the method's file explicitly on the command line when type-checking, so that the compiler reads its source code instead of its `.class` file.

- If the compiler reports that it cannot find a method from the JDK or another external library, then maybe the stub/skeleton file for that class is incomplete. You can edit it to add the missing method. The libraries appear, for example, at `checker/jdk/nullness/src/` for the Nullness Checker.
  The error might take one of these forms:

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

- If you get an error related to a bounded type parameter and a literal such as `null`, the problem may be missing defaulting. Here is an example:

  ```
  mypackage/MyClass.java:2044: warning: incompatible types in assignment.
        T retval = null;
                   ^
    found   : null
    required: T extends @MyQualifier Object
  ```

  A value that can be assigned to a variable of type `T extends @MyQualifier Object` only if that value is of the bottom type, since the bottom type is the only one that is a subtype of every subtype of `T extends @MyQualifier Object`. The value `null` satisfies this for the Java type system, and it must be made to satisfy it for the pluggable type system as well. The typical way to address this is to write the meta-annotation `@ImplicitFor(trees=Tree.Kind.NULL_LITERAL)` on the definition of the bottom type qualifier.
- An error such as

  ```
  MyFile.java:123: error: incompatible types in argument.
                          myModel.addElement("Scanning directories...");
                                             ^
    found   : String
    required: ? extends Object
  ```

  may stem from use of raw types. ("`String`" might be a different type and might have type annotations.) If your declaration was

  ```
  DefaultListModel myModel;
  ```

  then it should be

  ```
  DefaultListModel<String> myModel;
  ```

  Running the regular Java compiler with the `-Xlint:unchecked` command-line option will help you to find and fix problems such as raw types.

26.1.3 Unable to build the checker, or to run programs

An error like this

```
Unsupported major.minor version 51.0
```

means that you have compiled some files into the Java 7 format (version 51.0), but you are trying to run them with Java 6. (Likewise, "Unsupported major.minor version 52.0" means that you have compiled some files into the Java 8 format (version 52.0), but you are trying to run them with Java 7 or earlier.) Run `java -version` to determine the version of Java you are using and use a newer version, and/or use the `-target` command-line option to `javac` to specify the version of the class files that are created, such as `javac -target 7 ...`.

26.2 How to report problems (bug reporting)

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

Alternately (especially if your communication is not a bug report), you can send mail to checker-framework-dev@googlegroups.com. We welcome suggestions, annotated libraries, bug fixes, new features, new checker plugins, and other improvements.

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

- Add `-version -verbose -AprintErrorStack -AprintAllQualifiers` to the javac options. This causes the compiler to output debugging information, including its version number.
- Indicate exactly what you did. Don't skip any steps, and don't merely describe your actions in words. Show the exact commands by attaching a file or using cut-and-paste from your command shell;
- Include all files that are necessary to reproduce the problem. This includes every file that is used by any of the commands you reported, and possibly other files as well.
- Indicate exactly what the result was by attaching a file or using cut-and-paste from your command shell (don't merely describe it in words). Also indicate what you expected the result to be — remember, a bug is a difference between desired and actual outcomes.

A particularly useful format for a test case is as a new file, or a diff to an existing file, for the existing Checker Framework test suite. For instance, for the Nullness Checker, see directory `checker-framework/checker/tests/nullness/`. But, please report your bug even if you do not report it in this format.

26.3 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.

26.3.1 Obtain the source

Obtain the latest source code from the version control repository:

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

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

26.3.2 Build the Type Annotations compiler

1. Set the `JAVA_HOME` environment variable to the location of your JDK 7 installation (not the JRE installation, and not JDK 6). This needs to be an Oracle JDK. (The `JAVA_HOME` environment variable might already be set, because it is needed for Ant to work.)
   In the bash shell, the following command *sometimes* works (it might not because `java` might be the version in the JDK or in the JRE):

   ```
   export JAVA_HOME=${JAVA_HOME:-$(dirname $(dirname $(dirname $(readlink -f $(/usr/bin/which java)))))
   ```

2. Compile the Type Annotations tools:

   ```
   cd $JSR308/jsr308-langtools/make
   ant clean-and-build-all-tools
   ```

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

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

   You may wish to later put the Checker Framework `javac` even earlier on your path. The Checker Framework's `javac` ensures that the Checker Framework is on your classpath and bootclasspath, but is otherwise identical to the Type Annotations compiler.

26.3.3 Build the Annotation File Utilities

This is simply done by:

```
cd $JSR308/annotation-tools
ant
```

You do not need to add the Annotation File Utilities to the path, as the Checker Framework build finds it using relative paths.

26.3.4 Build the Checker Framework

1. Run `ant` to create `checker.jar`:

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

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

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

3. Test that everything works:
   - Run `ant all-tests` in the `checker` directory:

     ```
     cd $JSR308/checker-framework/checker
     ant all-tests
     ```

   - Run the Nullness Checker examples (see Section 3.5, page 29).

26.3.5 Build the Checker Framework Manual (this document)

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

26.4 Learning more

The technical paper "Practical pluggable types for Java" [PAC+08] (`http://homes.cs.washington.edu/~mernst/pubs/pluggable-checkers-issta2008.pdf`) gives more technical detail about many aspects of the Checker Framework and its implementation. The technical paper also describes case studies in which each of the checkers found previously-unknown errors in real software.

The paper "Building and using pluggable type-checkers" [DDE+11] (`http://homes.cs.washington.edu/~mernst/pubs/pluggable-checkers-icse2011.pdf`) discusses further experience with the Checker Framework, increasing the number of lines of verified code to 3 million.

In addition to these papers that discuss use the Checker Framework directly, other academic papers use the Checker Framework in their implementation or evaluation. Most educational use of the Checker Framework is never published, and most commercial use of the Checker Framework is never discussed publicly.

26.5 Comparison to other tools

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

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

- A type-checker aims to find *all* errors. Thus, it can verify the *absence* of errors: if the type-checker says there are no null pointer errors in your code, then there are none. (This guarantee only holds for the code it checks, of course; see Section 2.3.)
  A bug detector aims to find *some* of the most obvious errors. Even if it reports no errors, then there may still be errors in your code.
  Both types of tools may issue false alarms, also known as false positive warnings; see Section 21.2.
- A type-checker requires you to annotate your code with type qualifiers, or to run an inference tool that does so for you. A bug detector may not require annotations. This means that it may be easier to get started running a bug detector.
- A type-checker may use a more sophisticated and complete analysis. A bug detector typically does a more lightweight analysis, coupled with heuristics to suppress false positives.
  As one example, a type-checker can take advantage of annotations on generic type parameters, such as `List<@NonNull String>`, permitting it to be much more precise for code that uses generics.

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

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

26.6 Credits, changelog, and license

The key developers of the Checker Framework are Mahmood Ali, Telmo Correa, Werner M. Dietl, Michael D. Ernst, and Matthew M. Papi. Many other developers have also contributed, for example by writing the checkers that are distributed with the Checker Framework. Many, many users to list have provided valuable feedback, for which we are grateful.

Differences from previous versions of the checkers and framework can be found in the `changelog-checkers.txt` file. This file is included in the Checker Framework distribution and is also available on the web at `http://types.cs.washington.edu/checker-framework/current/changelog-checkerframework.txt`.

Two different licenses apply to different parts of the Checker Framework.

- The Checker Framework itself is licensed under the GNU General Public License (GPL), version 2. The GPL is the same license that OpenJDK is licensed under. That means that type-checking your code using the Checker Framework is no more dangerous (from an intellectual property point of view) than compiling your code using javac.
- The more permissive MIT License applies to code that you might want to include in your own program, such as the annotations.

For details, see file `LICENSE.txt`.

Bibliography

[Art01] Cyrille Artho. Finding faults in multi-threaded programs. Master's thesis, Swiss Federal Institute of Technology, March 15, 2001.

[Cop05] Tom Copeland. *PMD Applied.* Centennial Books, November 2005.

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

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

[DDE+11] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kıvanç Muşlu, and Todd Schiller. Building and using pluggable type-checkers. In *ICSE'11, Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690, Waikiki, Hawaii, USA, May 25–27, 2011.

[Ern08] Michael D. Ernst. Type Annotations specification (JSR 308). `http://types.cs.washington.edu/jsr308/`, September 12, 2008.

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

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

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

[Goe06] Brian Goetz. The pseudo-typedef antipattern: Extension is not type definition. `http://www.ibm.com/developerworks/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.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. In *Companion to Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 132–136, Vancouver, BC, Canada, October 26–28, 2004.

[HSP05] David Hovemeyer, Jaime Spacco, and William Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2005)*, pages 13–19, Lisbon, Portugal, September 5–6, 2005.

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

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

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

[SDE12] Eric Spishak, Werner Dietl, and Michael D. Ernst. A type system for regular expressions. In *FTfJP 2012: 14th Workshop on Formal Techniques for Java-like Programs*, Beijing, China, June 12, 2012.

[SM11] Alexander J. Summers and Peter Müller. Freedom before commitment: A lightweight type system for object initialisation. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2011)*, pages 1013–1032, Portland, OR, USA, October 25-27, 2011.

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

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