The Checker Framework Manual

MIT Program Analysis Group
http://groups.csail.mit.edu/pag/jsr308/

June 9, 2008

1 Introduction

The Checker Framework supports adding pluggable type systems to the Java language in a backward-compatible way. A type system designer defines 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 manual also documents 5 checkers that are built using the Checker Framework and are distributed with it. These checkers find errors or verify their absence.

1. the Nullness checker for null pointer errors (see Section 3)
2. Interned checker for equality testing and interning errors (see Section 4)
3. the Javari checker for mutation errors (incorrect side effects), based on the Javari type system (see Section 5)
4. the IGJ checker for mutation errors (incorrect side effects), based on the IGJ type system (see Section 6)
5. the Basic checker, which can check the type hierarchy for any annotation, without writing any code (see Section 7)

This document is organized as follows.

- Section 1.1 describes how to install the Checker Framework.
- Section 2 describes how to use a checker.
- The next sections give specific details for the Nullness (Section 3), Interned (Section 4), Javari (Section 5), IGJ (Section 6), and Basic (Section 7) checkers.
- Section 8 describes an approach for annotating external libraries.
- Section 9 describes how to write a new checker using the Checker Framework.

The paper "Practical pluggable types for Java" [PAC+08] (`http://people.csail.mit.edu/mernst/pubs/pluggable-checkers-issta2008.pdf`) introduces the Checker Framework at a high level. It also describes case studies in which each of the checkers found previously-unknown errors in real software.

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

1.1 Installation

To install the Checker Framework, simply place the `checkers.jar` file on your classpath. (You must have previously installed the JSR 308 `javac` compiler.)

The following instructions give detailed steps for installing the Checker Framework.

1. Download and install the JSR 308 implementation; follow the instructions at `http://groups.csail.mit.edu/pag/jsr308/current/README-jsr308.html#installing`. This creates a `langtools` directory.

2. Download the Checker Framework distribution zipfile from `http://groups.csail.mit.edu/pag/jsr308/current/jsr308-checkers.zip`, and unzip it to create a `checkers` directory. We recommend that the `checkers` directory and the `langtools` directory be siblings. Example commands:

   ```
   cd ~/jsr308
   wget http://groups.csail.mit.edu/pag/jsr308/current/jsr308-checkers.zip
   unzip jsr308-checkers.zip
   ```

3. Edit property `compiler.lib` in `checkers/build.properties`.
4. Add to your classpath: `$HOME/jsr308/jdk1.7.0/lib/tools.jar` and `$HOME/jsr308/checkers/checkers.jar`. (If you do not do this, you will have to supply the `-cp` option whenever you run `javac` and use a checker plugin.) Example commands:

   ```
   export CLASSPATH=${CLASSPATH}:$HOME/jsr308/jdk1.7.0/lib/tools.jar:$HOME/jsr308/checkers/checkers.jar
   ```

5. Test that everything works:
   - Run `ant all-tests` in the `checkers` directory:
     `ant all-tests`
   - Run the Nullness checker examples (see Section 3.5).

JSR 308 extends the Java language to permit annotations to appear on types, as in `List<@NonNull String>`. This change is planned to be part of the Java 7 language.) We recommend that you write annotations in comments, as in `List</*@NonNull*/ String>` (see Section 2.1). The JSR 308 compiler still reads such annotations, but this syntax permits you to use a compiler other than the JSR 308 compiler. For example, you can use a checker as an external tool in an IDE such as Eclipse.

1.1.1 Building from source

Building (compiling) the checkers and framework from source creates the `checkers.jar` file. A pre-compiled `checkers.jar` is included in the distribution, so building it is optional. It is mostly useful for people who are developing compiler plug-ins (type-checkers). If you only want to *use* the compiler and existing plug-ins, it is sufficient to use the pre-compiled version.

1. Edit `checkers/build.properties` file so that the `compiler.lib` property specifies the location of the JSR 308 `javac.jar` library. (If you also installed the JSR 308 compiler from source, and you made the `checkers` and `langtools` directories siblings, then you don't need to edit `checkers/build.properties`.)
2. Run `ant` in the `checkers` directory:

   ```
   cd checkers
   ant
   ```

2 Using a checker

Finding bugs with a checker plugin is a two-step process:

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.)
2. The checker reports whether the program contains any erroneous code — that is, code that is inconsistent with the annotations.

2.1 Writing annotations

The syntax of type qualifier annotations is specified by JSR 308 [Ern07]. Ordinary Java permits annotations on declarations. JSR 308 permits annotations anywhere that you would write a type, including generics and casts. You can also write annotations to indicate type qualifiers for array levels and receivers. Here are a few examples:

```
@Interned String intern() { ... }             // return value
int compareTo(@NonNull String other) { ... }  // parameter
String toString() @ReadOnly { ... }           // receiver ("this" parameter)
@NonNull List<@Interned String> messages;     // generics:  non-null list of interned Strings
@NonNull String[@Interned] messages;          // arrays:  non-null array of interned Strings
myDate = (@ReadOnly Date) readonlyObject;     // cast
```

2.1.1 Writing annotations in comments for backwarrd compatibility

Sometimes, your code needs to be compilable by people who are not using the JSR 308 compiler.

A Java 4 compiler does not permit use of annotations, and a Java 5 compiler only permits annotations on declarations (but not on generic arguments, casts, etc.). For backward compatibility, you may write any annotation inside a `/*...*/` Java comment, as in `List</*@NonNull*/ String>`. The JSR 308 compiler will recognize such an annotation, but your code will still compile with pre-JSR-308 compilers.

The compiler ignores any comment that does not appear to contain exactly one annotation. The compiler ignores any comment that contains spaces at the beginning or end, or between the `@` and the annotation name. Compiler flag `-Xspacesincomments` causes the compiler to parse annotation comments even when they contain spaces.

When writing source code with annotations, it is more convenient to write a short form such as `@NonNull` instead of `@checkers.nonnull.quals.NonNull`. There are two ways to do this.

- Write an import statement like: `import checkers.nonnull.quals.*;`
  A potential disadvantage of this is that everyone who compiles the code (even using a non-JSR-308 compiler) must have the annotation definitions (e.g., the `checkers.jar` file) on their classpath. The reason is that a Java compiler issues an error if an imported package is not on the classpath.
- When you compile the code, set the shell environment variable `jsr308_imports`. This permits your code to compile whether or not the JSR 308 compiler is being used.
  In bash, you could write `export jsr308_imports='checkers.nonnull.quals.*'`, or prefix the `javac` command by `jsr308_imports='checkers.nonnull.quals.*'` . Alternately, you can set the system variable via the javac command line argument `-J-Djsr308_imports="checkers.nonnull.quals.*"`.

2.2 Running a checker

To run a checker plugin, run the JSR 308 compiler `javac` as usual, but pass the `-typeprocessor` *plugin_class* command-line option. Two concrete examples (using the Nullness checker) are:

```
javac -typeprocessor checkers.nonnull.NonNullChecker MyFile.java
javac -typeprocessor checkers.nonnull.NonNullChecker -sourcepath checkers/jdk/nonnull/src MyFile.java
```

For a discussion of the `-sourcepath` argument, see Section 8.1.2.

You can always compile the code without the `-typeprocessor` command-line option, but in that case no checking of the type annotations is performed.

2.3 Checking against unannotated code

A checker plugin reads annotations from the source code or `.class` files of classes that are used by the code being compiled and checked. If annotated code uses unannotated code (e.g., libraries or the JDK), then the checker may issue warnings. For example, the Nullness checker (Section 3) will warn whenever an unannotated library call result is used in a non-null context:

```
@NonNull myvar = library_call();   // WARNING: library_call may return a null value
```

If the library call can return null, you should fix the bug in your program by removing the `@NonNull` annotation. If the library call never returns null, there are two general ways to prevent compiler warnings: add the missing annotations (Section 2.3.1), or suppress the warnings (Section 2.4).

2.3.1 Adding library annotations

You may be able to obtain a version of the library that contains the annotations, or a set of external annotations that describe the library. For example, the Checker Framework distribution contains annotations for popular libraries, such as the JDK. Section 8.1.2 describes how to use them.

Otherwise, you will need to annotate the library, using one of these techniques:

- If source code is available, you can annotate the source code and re-compile the library.
- If no source code is available, or if you do not want to edit and recompile the library, you can use the skeleton class generation tool; see Section 8.
- You can annotate the compiled `.jar` or `.class` files using the annotation file utilities (`http://groups.csail.mit.edu/pag/jsr308/annotation-file-utilities/`). First, express the annotations textually as an annotation index file, and then the tools insert them in the compiled library class files.

If you annotate additional libraries, please share them with us so that we can distribute the annotations with the Checker Framework; see Section 2.8.

2.4 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

- the javac `-Alint` command-line option,
- the `@SuppressWarnings` annotation, or
- the `checkers.skipClasses` Java property.

You can suppress an entire class of warnings via javac's `-Alint` command-line option. Following `-Alint=`, write a list of option names. If the option name is preceded by a hyphen (-), that disables the option; otherwise it enables it. For example: `-Alint=-dotequals` causes the Interned checker (Section 4) not to output advice about when `a.equals(b)` could be replaced by `a==b`.

You can suppress specific errors and warnings by use of the `@SuppressWarnings("`*annotationname*`")` annotation, for example `@SuppressWarnings("interned")`. This may be placed on program elements such as a class, method, or local variable declaration. It is good practice to suppress warnings in the smallest possible scope.

You can suppress errors and warnings pertaining to un-annotated (or other) classes by setting the `checkers.skipClasses` Java property to a regular expression that matches classes for which warnings and errors should be suppressed. For example, if you use "`-Dcheckers.skipClasses=^java\.`" on the command line when invoking `javac`, then the checkers will suppress warnings relating to uses of classes in the `java` package. (Note that if your `javac` is a script rather than a binary, it may not support JVM flags such as `-D`; in that case, you may need to edit `javac` script itself to pass the `-D` flag. This is a flaw in the OpenJDK build process, which we will try to correct in a future release.)

You can also compile parts of your code without use of the `-typeprocessor` switch to `javac`. No checking is done during such compilations.

Finally, some checkers have special rules. For example, the Nullness checker (Section 3) uses `assert` statements that contain null checks to suppress warnings.

2.5 Implicitly annotated types (flow-sensitive type qualifier inference)

In order to reduce the burden of annotating types in your program, the checkers treat certain variables and expressions as being annotated, even if you have not annotated them. For instance, the Nullness checker (Section 3) can automatically determine that certain variables are non-null, without you having to annotate them. By default, all checkers, including new checkers that you write, take advantage of this functionality.

For example, a variable or expression can be treated as `@NonNull` from 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 explicitly non-null types, without compiler warnings.

For example, consider this code, along with comments indicating whether the Nullness checker 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(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 instances 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 2.8) that includes a small piece of Java code that reproduces the problem.

Type inference is never performed for method parameters of non-private methods and for non-private fields, because unknown client code could use them in arbitrary ways. The inferred information is never written to the `.class` file as user-written annotations are.

The inference indicates when a variable can be treated as having a subtype of its declared type — for instance, when an otherwise nullable type can be treated as a `@NonNull` one. The inference never treats a variable as a supertype of its declared type (e.g., an expression of `@NonNull` type is never inferred to be treated as possibly-null).

2.6 What the checker guarantees

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

- Native methods and reflection can behave in a manner that is impossible for a compiler plugin to check. Such constructs they may violate the property being checked. Similarly, deserialization and cloning can create objects that could not result from normal constructor calls, and that therefore may violate the property being checked.
- 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 the `-typeprocessor` switch or with the `checkers.skipClasses` property (in other words, without running the checker), or if you use the `@SuppressWarnings` annotation to suppress some errors or warnings, then there is no guarantee that the entire program satisfies the property being checked. An analogous situation is using an external library that was compiled without being checked by the compiler plugin.
- The Checker Framework does not yet support annotations on intersection types (see JLS §4.9). As a result, checkers cannot provide guarantees about intersection types.

- Specific checkers may have other limitations; see their documentation for details.

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

2.7 Troubleshooting

If you get the error

```
com.sun.tools.javac.code.Symbol$CompletionFailure: class file for com.sun.source.tree.Tree not found
```

then file `tools.jar` is not on your classpath; see the installation instructions (Section 1.1).

If you get an error such as

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

despite no apparent use 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 2.1.1). You can solve this by unsetting the variable/option, or by ensuring that the `checkers.jar` file is on your classpath.

2.7.1 Known problems

- The framework currently does not honor annotated type variables (e.g., `@NonNull T`).

2.8 How to report problems

If you have any problems with any checker, or with the Checker Framework, please let us know at `jsr308-bugs@lists.csail.mit.ed`
In addition to bug reports, 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:

- Indicate exactly what you did. Show the exact commands (don't merely describe them in words). Don't skip any steps.
- 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 (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.
- Indicate which version of the JSR 308 compiler and Checker Framework you are using. You can determine the JSR 308 version by running `javac -version`.

2.9 Credits and changelog

The Checker Framework distribution was developed in the MIT Program Analysis Group. The Checker Framework was implemented by Matthew M. Papi and Mahmood Ali. The non-null checker was implemented by Matthew M. Papi. The interned checker was implemented by Matthew M. Papi. The Javari checker was implemented by Telmo Correa. The IGJ checker was implemented by Mahmood Ali. The basic checker was implemented by Matthew M. Papi. Many users have provided valuable feedback.

Differences from previous versions of the checkers and framework can be found in the `changelog-checkers.txt` file. This file is included in the checkers distribution and is also available on the web at `http://groups.csail.mit.edu/pag/jsr308/current/changelog-checkers.txt`.

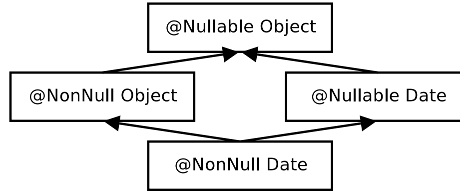

Figure 1: Type hierarchy for the Nullness type system. Java's `Object` is expressed as `@Nullable Object`. Programmers can omit most type qualifiers, because the default annotation (Section 3.2) is usually correct.

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 his program is run. See Section 3.6 for caveats to the guarantee.

3.1 Annotating your code with `@NonNull` and `@Nullable`

In order to perform checking, you must annotate your code. You can write the `@NonNull` type annotation, which indicates a type that does not include the null value, or the `@Nullable` type annotation, which indicates a type that does include null. Unannotated references are treated as if they had a default annotation; see Section 3.2.

A variable of type `Boolean` always has one of the values `TRUE`, `FALSE`, or `null`. By contrast, a variable of type `@NonNull Boolean` always has one of the values `TRUE` or `FALSE` — never `null`. Dereferencing an expression of type `@NonNull Boolean` can never cause a null pointer exception.

The checker issues a warning in two cases:

1. When an expression of non-`@NonNull` type is dereferenced, because it might cause a null pointer exception.
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.

This example shows both sorts of problems:

```
         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
```

Parameter passing and return values are checked analogously to assignments.

You can control the behavior of the Nullness checker via the `-Alint` options `flow`, `cast`, and `cast:redundant`.

3.2 Default annotations

As noted in Section 3.1, you can write `@NonNull` and `@Nullable` type annotations. Unannotated references are treated as if they had a default annotation.

There are three possible defaults:

- `@Nullable`: Unannotated types are regarded as possibly-null, or nullable. This default is backward-compatible with Java, which permits any reference to be null. You can activate this default by writing a `@Default("checkers.nonnull.quals.Nullable")` annotation on a class or method declaration. If you write no `@Default` annotation, then the checker currently uses this default.
- `@NonNull`: Unannotated types are treated as non-null. You can activate this default via the `@Default("checkers.nonnull.quals` annotation.

- Non-null except locals (NNEL): Unannotated types are treated as `@NonNull`, *except* that the unannotated raw type of a local variable is treated as `@Nullable`. (Any generic arguments to a local variable still default to `@NonNull`.) You can activate this default via the `@Default(value="checkers.nonnull.quals.NonNull", types={DefaultLocation.ALL_EXCEPT_LOCALS})` annotation.
  The NNEL default leads to the smallest number of explicit annotations in your code. It is what we recommend, and will become the default default in a future release.

The `@Default` annotation has an argument for the fully qualified `String` name of an annotation, and an optional second argument indicating where the default applies. If the second argument is omitted, th specified annotation is the default in all locations.

This example illustrates the use of the `@Default` annotation:

```
@Default("checkers.nonnull.quals.NonNull")
public boolean compile(File file) {  // file has type "@NonNull File"
  if (!file.exists())    // no warning: file cannot be null
    return false;

  @Nullable File srcPath = ...;    // must annotate to specify "@Nullable File"
  // ...
  if (srcPath.exists())  // warning: srcPath might be null
    // ...
}
```

3.3 `@Raw` annotation for partially-initialized objects

During execution of a constructor, every field of non-primitive type starts out with the value `null`. If the field has `@NonNull` type, the value `null` violates the type. If the constructor makes a method call (passing `this` as a parameter or the receiver), then the called method could observe the object in an illegal state.

The `@Raw` type annotation represents a partially-initialized object. If a reference has `@Raw` type, then all fields fields are treated as `@Nullable`. Within the constructor, `this` has `@Raw` type and can only be passed to methods when the corresponding parameter is annotated with `@Raw`. Similar restrictions apply to assigning `this` to a field.

The name "raw" comes from a research paper that proposed this approach [FL03]. The `@Raw` annotation has nothing to do with the raw types of Java Generics.

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

It can be tedious to write annotations in your code. Two tools exist that can automatically infer annotations and insert them in your program.

- JastAdd (`http://jastadd.org`) can infer `@NonNull` annotations.
- Daikon (`http://pag.csail.mit.edu/daikon/`) can infer `@Nullable` annotations.

You only need one of these tools. Daikon is primarily useful if you are using a default annotation (Section 3.2) of `@NonNull` or non-null-except-locals.

3.5 Examples

3.5.1 Tiny examples

To try the `@NonNull` checker on a source file that uses the `@NonNull` qualifier, use the following command (where `javac` is the JSR 308 compiler):

```
javac -typeprocessor checkers.nonnull.NonNullChecker examples/NonNullExample.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 -typeprocessor checkers.nonnull.NonNullChecker examples/NonNullExampleWithWarnings.java
```

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

3.5.2 Annotated library

The Nullness checker itself is annotated with `@NonNull`.

In addition, you can run the Nullness checker on the annotation scene library, another library that has been fully annotated with `@NonNull`. To run the Nullness checker on the annotation scene library, first download the scene library suite (which includes build dependencies for the scene library as well as its source code) and extract it into your checkers installation. The checker can then be run on the annotation scene library with Apache Ant using the following commands:

```
cd checkers
ant -f scene-lib-test.xml
```

You can view the annotated source code, which contains `@NonNull` annotations, in the `checkers/scene-lib-test/src/annotations/` directory.

3.6 Caveats to the guarantee of no null pointer errors

The Nullness checker prevents null pointer errors in your code. In addition to the caveats for any checker (Section 2.6), there is one additional caveat:

- The Nullness checker assumes that assertions are enabled, so that no null pointer exception can occur in code such as

  ```
  assert x != null;
  ... x.f ...
  ```

  If the JVM is run with assertions disabled, then a null pointer exception could occur.

3.7 Related work

The Checker Framework `@NonNull` annotation is similar, but not identical, to the `@NotNull` annotation of IntelliJ IDEA, the `@NonNull` annotation of FindBugs, the `non_null` modifier of JML, and annotations proposed by JSR 305, among others.

4 Interned checker

If the Interned checker issues no warnings for a given program, then all reference equality tests (i.e., "`==`") in that program operate on interned types. Interning can save memory and can speed up testing for equality by permitting use of `==`; however, use of `==` 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 Interned checker helps programmers to prevent such bugs. The Interned checker also helps to prevent performance problems that result from failure to use interning. (See Section 2.6 for caveats to the checker's guarantees.)

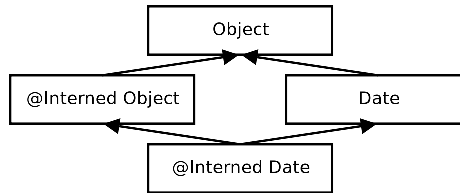

Figure 2: Type hierarchy for the Interning type system.

4.1 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 What the Interned checker checks

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

The checker issues a warning in two cases:

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

This example shows both sorts of problems:

```
          Object  obj;
@Interned Object iobj;
...
if (obj == iobj) { ... }  // checker warning: reference equality test is unsafe
iobj = obj;               // checker warning: iobj's referent may no longer be interned
```

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
```

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`.

4.3 Examples

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

```
javac -typeprocessor checkers.interned.InternedChecker examples/InternedExample.java
```

Compilation will complete without warnings.

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

```
javac -typeprocessor checkers.interned.InternedChecker examples/InternedExampleWithWarnings.java
```

The compiler will issue a warning regarding violation of the semantics of `@Interned`.

The Daikon invariant detector (`http://groups.csail.mit.edu/pag/daikon/`) is also annotated with `@Interned`.

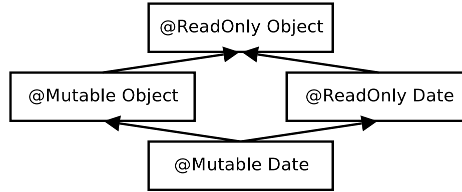

Figure 3: Type hierarchy for Javari's ReadOnly type qualifier.

5 Javari checker

Javari [TE05, QTE08] is a Java language extension that helps programmers to avoid mutation errors that result from unintended side effects. If the Javari checker issues no warnings for a given program, then that program will never change objects that should not be changed. This guarantee enables a programmer to detect and prevent mutation-related errors. (See Section 2.6 for caveats to the guarantee.) The Javari webpage (`http://groups.csail.mit.edu/pag/javari/`) gives pointers to papers that explain the Javari language and type system.

The Javari webpage also contains a separate program, the Javarifier (`http://groups.csail.mit.edu/pag/javari/javarifier/`), which infers Javari types for an existing program. The Javarifier inserts Javari annotations in a 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. (Annotation of libraries is not as critical for other type systems such as the Nullness checker (Section 3) and the Interned checker (Section 4).)

5.1 Annotation Javari dialect

The Javari checker uses an annotation-based dialect of the Javari language.

The supported annotations are `@ReadOnly`, `@Mutable`, `@Assignable`, `@QReadOnly` and `@PolyRead`, that correspond to the Javari keywords `readonly`, `mutable`, `assignable`, `? readonly`, and `romaybe`, respectively.

The `@ReadOnly` type annotation indicates that a reference provides only read-only access. 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.

The `@Mutable` annotation ensures that a reference is mutable, no matter the inherited mutability.

The `@QReadOnly` annotation is a mutability wildcard that can be applied to types (for example, `List<@QReadOnly Date>`). As such, it allows only the operations which are allowed for both readonly and mutable types.

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

5.2 Inference of Javari annotations

It can be tedious to write annotations in your code. The Javarifier tool (`http://groups.csail.mit.edu/pag/javari/javarifier/`) can automatically infer Javari's annotations and insert them in your program.

5.3 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, or specify just one of the test files.

```
javac -typeprocessor checkers.javari.JavariChecker tests/javari/*.java
```

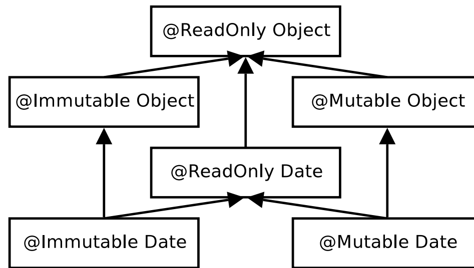

Figure 4: Type hierarchy for three of IGJ's type qualifiers.

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.

6 IGJ checker

IGJ is a Java language extension that helps programmers to avoid mutation errors that result from 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 2.6 for caveats to the guarantee.)

6.1 IGJ and Mutability

IGJ 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 the 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 6.3), which is slightly different dialect of IGJ than that described in the ESEC/FSE paper.

6.2 Supported Annotations

The supported annotations are `@ReadOnly`, `@Mutable`, `@Immutable`, `@Assignable`, and `@AssignsFields`, as specified in the IGJ paper. The `@I(`*`string`*`)` annotation is added to mimic the template behavior of generics.

The `@ReadOnly` type annotation indicates that a reference provides only read-only access. 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.

The `@Mutable` annotation ensures that a reference is mutable, no matter the inherited mutability. `@AssignsFields` similar, but permits only limited mutation — assignment of fields — and is for use by constructor helper methods.

The `@Immutable` annotation ensures that a reference is to an immutable object.

The `@I` annotation simulates mutability overloading. It can be applied to classes, methods and parameters. See Section 6.3.3.

6.3 Annotation IGJ Dialect

The IGJ checker supports the Annotation IGJ dialect of IGJ. The syntax of Annotation IGJ is based on JSR 308 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.

6.3.1 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>
```

- 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.

6.3.2 Syntax Changes

- Immutability is specified through JSR 308 [Ern07] annotations (Section 6.2), not through a combination of generics and annotations. Use of JSR 308 annotations makes Annotation IGJ backward compatible with Java syntax.
- Templating over Immutability: The annotation @I(*id*) is used to template over immutability. See Section 6.3.3.

6.3.3 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 annotated with @I could be declared with any IGJ Immutability annotation. The actual immutability that @I is resolved to dictates the immutability type for all the non-static appearances of @I with the same value as the class declaration.

Example:

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

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

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

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

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

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

For example, 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, 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) { ... }
```

6.4 Examples

To try the IGJ checker on a source file that uses the IGJ qualifier, use the following command, where `javac` is the JSR 308 compiler.

```
javac -typeprocessor checkers.igj.IGJChecker examples/IGJExample.java
```

The IGJ checker itself is also annotated with IGJ annotations.

7 The Basic checker

The Basic checker enforces only subtyping rules. It operates over annotations specified by a user on the command line. Thus, users can create a simple type checker without writing any code beyond definitions of the type qualifier annotations.

The Basic checker can accommodate all of the type system enhancements that can be declaratively specified (see Section 9). This includes type introduction rules (implicit annotations, e.g., literals are implicitly considered `@NonNull`) via the `@ImplicitFor` meta-annotation, and other features such as qualifier polymorphism.

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

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 Section 9.

7.1 Using the Basic checker

The Basic checker is used in the same way as other checkers (using the `-processor` option; see Section 2), except that it requires an additional annotation processor argument via the standard "`-A`" switch:

- `-Aquals`: this option specifies a comma-no-space-separated list of the fully-qualified class names of the annotations used as qualifiers in the custom type system. It serves the same purpose as the `@TypeQualifiers` annotation used by other checkers (see section 9.4).

The annotations listed in `-Aquals` must be accessible to the compiler during compilation, either on the classpath or sourcepath or as one of the `.java` files passed to the compiler.

7.2 Basic checker example

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

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

```
package myquals;

/**
 * Denotes that the representation of an object is encrypted.
 * ...
 */
@TypeQualifier
public @interface Encrypted {}
```

2. Write `@Encrypted` annotations in your program:

```
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);
}
```

3. Invoke the compiler with the Basic checker, specifying the `@Encrypted` annotation using the `-Aquals` option:

```
\$ javac -processor checkers.util.BasicChecker -Aquals=myquals.Encrypted YourProgram.java

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

8 Annotating libraries with the skeleton class generator

When annotated code uses unannotated code (e.g., libraries such as the JDK), a checker may issue warnings (see Section 2.3). As described in Section 2.3.1, the best way to correct this problem is to add annotations to the library.

One way to do so is to annotate a "skeleton class" version of the library and use it during compilation (only). A skeleton class has trivial method bodies that always throw an exception.

8.1 Creating and using a skeleton class

There are two steps to creating, and two steps to using, a skeleton class. We illustrate them via the example of creating a `@NonNull`-annotated version of `java.lang.Set`. (You don't need to repeat these steps, since such a

skeleton class is already included in the Checker Framework distribution.)

First, you must install the skeleton class generator (Section 8.2).

8.1.1 Creating a skeleton class

1. Create a skeleton class by running the skeleton class generator.

   ```
   cd checkers/jdk/nonnull/src
   java checkers.util.skel.Skeleton java.util.Set > java/util/Set.java
   ```

   Supply it with the fully-qualified name of the class for which you wish to generate a skeleton class. The skeleton class generator prints the skeleton class to standard out, so you may wish to redirect its output to a file. See Section 8.2 for installation instructions for the skeleton class generator.
2. Add annotations to the skeleton class. For example, you might annotate the `Set.iterator()` method as follows:

   ```
   public abstract @NonNull java.util.Iterator<E> iterator();
   ```

8.1.2 Using a skeleton class

1. Use `javac`'s `-sourcepath` argument to indicate where to find the skeleton classes. The checker will read annotations from the annotated skeleton class instead of the unannotated original library class.

   ```
   javac -typeprocessor checkers.nonnull.NonNullChecker -sourcepath checkers/jdk/nonnull/src my_source_files
   ```

2. When you run the compiled code, do *not* include the skeleton files on the classpath. If a skeleton method is called instead of the true library method, then your program will throw a RuntimeException.

8.2 Installing the skeleton class generator

Source code for the skeleton class generator tool is included in the Checker Framework distribution, but because the tool has additional dependencies, the provided build script does not build the tool by default.

Follow these steps to install the skeleton class generator:

1. Install the annotation file utilities, using the instructions at `http://groups.csail.mit.edu/pag/jsr308/annotation-file-utilities/`. Per those instructions, the `annotation-file-utilities.jar` file should be on your classpath.
2. Update the `build.properties` file in the Checker Framework distribution so that the `annotation-utils.lib` property specifies the location of the `annotation-file-utilities.jar` library.
3. Build the skeleton class generator tool by running `ant skeleton-util dist` in the `checkers` directory. This updates the `checkers.jar` file to contain the skeleton class generator. `checkers.jar` should already be on your classpath (see Section 1.1).

8.3 Known problems

The skeleton class generator has several limitations that require you to edit its output before using it. We are working to correct these bugs.

- It does not handle `enum`s.
- It does not add a `super()` call in constructors.
- It does not add type variable declarations in static methods.

9 How to create a new checker plugin

This section describes how to extend the Checker Framework to create a checker — a type-checking compiler plugin that detects bugs or verifies their absence. After a programmer annotates a program, the checker

plugin verifies that the code is consistent with the annotations. If you only want to *use* a checker, you do not need to read this section.

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 9.1–9.4 describe the components of a type system as written using the Checker Framework:

9.1 **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`).

9.2 **Type introduction rules.** For some types and expressions, a qualifier should be treated as present even if a programmer did not explicitly write it. For example, every literal (other than `null`) has a `@NonNull` type.

9.3 **Type rules.** You specify the the type system semantics (type rules), violation of which yields a type error. For example, in the Nullness type system, only references with a `@NonNull` type may be dereferenced. In every type system, every assignment and pseudo-assignment must satisfy a subtyping relationship; your checker automatically inherits such rules.

9.4 **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.

In addition to reading this section of the manual, you may find it helpful to examine the implementations of the checkers that are distributed with the Checker Framework, or to create your checker by modifying another one.

9.1 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] — that is, using the Java `@interface` keyword. Write the `@TypeQualifier` meta-annotation on the annotation definition to indicate that the annotation represents a type qualifier (e.g., `@NonNull` or `@Interned`). This distinguishes it from an ordinary annotation that applies to a declaration (e.g., `@Deprecated` or `@Override`). The framework ignores any annotation whose declaration does not bear the `@TypeQualifier` meta-annotation (with minor exceptions, such as `@SuppressWarnings`).

The type hierarchy induced by the qualifiers can be defined either declaratively (via meta-annotations) or procedurally (through sub-classing `QualifierHierarchy` or `TypeHierarchy`).

9.1.1 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 the subtype of another qualifier or qualifiers, specified as an array of class literals. For instance, for the Nullness checker, the annotation `@NonNull` is annotated with `@SubtypeOf({Nullable.class})`, since for some type T, `@NonNull` T is a subtype of `@Nullable` T.
  `@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 6.2), `@Mutable` and `@Immutable` induce two mutually exclusive subtypes of the `@ReadOnly` qualifier.
- `@QualifierRoot` denotes that a qualifier is the supertype of all other qualifiers in the type hierarchy. For instance, for the Nullness checker, the definition of `Nullable` is annotated with `@QualifierRoot`, and for the Javari checker, the definition of `ReadOnly` is annotated with `@QualifierRoot`.

When using the meta-annotation facility, further customizations may be added by overriding the `isSubtype` method in the checker class; for most type systems, however, simply writing these meta-annotations on the checker class and the annotation declarations for these qualifiers is sufficient.

9.1.2 Procedurally defining the qualifier and type hierarchy

While the declarative syntax suffices for many cases, more complex type hierarchies can be expressed by overriding, in `BaseTypeChecker`, either `createQualifierHierarchy` or `createTypeHierarchy` (typically only one of these needs to be overridden). For more details, see the Javadoc of those methods and of the classes `QualifierHierarchy` and `TypeHierarchy`.

The `QualifierHierarchy` class represents the qualifier hierarchy (not the type hierarchy), e.g., `Mutable` is a subtype of `ReadOnly`. A type-system designer may subclass `QualifierHierarchy` to express customized qualifier relationships (e.g., relationships based on annotation arguments).

The `TypeHierarchy` class represents 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 wild-cards 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, Java language specifications specifies that two generic types are subtypes only if their type arguments are invariants. The Javari type system overrides this behavior to allow type arguments to change covariantly in a type-safe manner (e.g., `List<@Mutable Date>` is a subtype of `List<@QReadOnly Date>`).

9.2 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.

9.2.1 Declaratively specifying implicit annotations

The `@ImplicitFor` meta-annotation indicates implicit annotations. When written on a qualifier, `@ImplicitFor` specifies the trees and types for which the framework should automatically add that qualifier. These types and trees can be specified via any combination of four fields:

- `trees`: an array of `com.sun.source.tree.Tree.Kind`, e.g., `NEW_ARRAY` or `METHOD_INVOCATION`
- `types`: an array of `javax.lang.model.type.TypeKind`, e.g., `ARRAY` or `BOOLEAN`
- `treeClasses`: an array of class literals for classes implementing `com.sun.source.tree.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`

For example, the `@checkers.nonnull.quals.Nullable` annotation is annotated with `@ImplicitFor(trees={Tree.Kind.NULL_LITERAL})` to denote that the framework should automatically apply `@Nullable` to all instances of `null`.

(*Note:* `@ImplicitFor` handling currently requires explicitly invoking `checkers.types.TypeAnnotator` and `checkers.types.TreeAnnotator` from within the `checkers.types.AnnotatedTypeFactory.annotateImplicit` method. Automatic handling of `@ImplicitFor` will be added to a future release.)

9.2.2 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.

9.3 Visitor: Type Rules

A type system's rules define which operations on values of a particular type are forbidden.

The framework provides a *base visitor class*, `BaseTypeVisitor`, that performs type-checking at each node of a source file's AST. It uses the visitor design pattern to traverse Java syntax trees as provided by Sun's Tree API, and issues a warning whenever the type system induced by the type qualifier is violated.

The 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 Section 3 consists of 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) when an assignment from an expression type to an incompatible type. The assignment may be a simple assignment, or pseudo-assignment like return expressions or argument passing in a method invocation
  In particular, in every assignment and pseudo-assignment, the left-hand side of the assignment is a supertype of (or the same type as) the right-hand side. For example, this assignment is not permitted:
  ```
  @Nullable Object myObject;
  @NonNull Object myNonNullObject;
  ...
  myNonNullObject = myObject;  // invalid assignment
  ```
- invalid generic argument (generic.argument.invalid) when a type is bound to an incompatible generic type variable
- invalid method invocation (method.invocation.invalid) when a method is invoked on an object whose type is incompatible with the method receiver type
- invalid overriding parameter type (override.parameter.invalid) when a parameter in a method declaration is incompatible with that parameter in the overridden method's declaration
- invalid overriding return type (override.return.invalid) when a parameter in a method declaration is incompatible with that parameter in the overridden method's declaration
- invalid overriding receiver type (override.receiver.invalid) when a receiver in a method declaration is incompatible with that receiver in the overridden method's declaration

9.4 TypeChecker: Compiler Interface

The base class for a checker's entry point is `BaseTypeChecker`, which subclasses Sun's `AbstractProcessor`. The Checker class serves two roles: an interface to the compiler and a factory for constructing type-system classes.

The TypeChecker class should be annotated by `@TypeQualifiers`, which lists the annotations that make up the type hierarchy for this checker, 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 TypeChecker class bridges between the compiler and the checker plugin. It invokes the type-rule check visitor on every Java source file being compiler, and provide a simple API, `report(Result, Tree)`, 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 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 name `message.properties`. The property file keys are the strings passed to the Checker.report (like `type.incompatible`) and the values are the strings to be printed (`cannot assign` ...). The properties text file needs to be in the same directory as the `TypeChecker`.

9.5 Testing framework

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

9.6 Debugging options

The Checker Framework provides debugging options that can be helpful when writing checker. These are provided via the standard `javac` "`-A`" switch, which is used to pass options to an annotation processor.

- `-Anomsgtext`: use message keys (such as "`type.invalid`") rather than full message text when reporting errors or warnings
- `-Ashowchecks`: print debugging information for each pseudo-assignment check (as performed by `BaseTypeVisitor`; see Section 9.3 above)
- `-Afilenames`: prints the name of each file before type-checking it

The following example demonstrates how these options are used:

```
$ javac -processor checkers.interned.InternedChecker \
    examples/InternedExampleWithWarnings.java -Ashowchecks -Anomsgtext -Afilenames

[InternedChecker] InternedExampleWithWarnings.java
 success (line  18): STRING_LITERAL "foo"
     actual: DECLARED @checkers.interned.quals.Interned java.lang.String
   expected: DECLARED @checkers.interned.quals.Interned java.lang.String
 success (line  19): NEW_CLASS new String("bar")
     actual: DECLARED java.lang.String
   expected: DECLARED java.lang.String
examples/InternedExampleWithWarnings.java:21: (not.interned)
    if (foo == bar)
            ^
 success (line  22): STRING_LITERAL "foo == bar"
     actual: DECLARED @checkers.interned.quals.Interned java.lang.String
   expected: DECLARED java.lang.String
1 error
```

References

[Dar06] Joe Darcy. JSR 269: Pluggable annotation processing API. http://jcp.org/en/jsr/detail?id=269, May 17, 2006. Public review version.

[Ern07] Michael D. Ernst. Annotations on Java types: JSR 308 working document. http://pag.csail.mit.edu/jsr308/, November 12, 2007.

[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.

[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*, 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*, Paphos, Cyprus, July 9–11, 2008.

[TE05] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, pages 211–230, San Diego, CA, USA, October 18–20, 2005.

[ZPA+07] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam 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*, Dubrovnik, Croatia, September 5–7, 2007.