JSR 308 Type-checkers and Framework

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

April 26, 2008

1 Introduction

This is the documentation for the JSR 308 Type-checkers and Framework, also known as the "JSR 308 Checkers". The JSR 308 Checkers distribution contains:

- Compiler plugins, known as checkers, that find errors or verify their absence:
  1. a checker for null pointer errors (see Section 3)
  2. a checker for equality testing and interning errors (see Section 4)
  3. a checker for mutability errors (incorrect side effects), based on the Javari type system (see Section 5)
  4. another checker for mutability errors (incorrect side effects), based on the IGJ type system (see Section 6)
  5. a customizable checker that can check any annotation you want, so long as the annotation does not require special semantics (see Section 8.1)
- A framework that facilitates the writing of checker plugins.

This document is organized as follows.
Section 1.1 describes how to install the JSR 308 Checkers.
Section 2 describes how to use a checker.
The next sections give specific details for the NonNull (Section 3), Interned (Section 4), Javari (Section 5), and IGJ (Section 6) checkers.
Section 7 describes an approach for annotating external libraries.
Section 8 describes how to write a new checker using the checkers framework.

A technical report [PAJ+07] (`http://people.csail.mit.edu/mernst/pubs/custom-types-tr047.pdf`) describes case studies in which each of the four 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 JSR 308 Checkers, 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 JSR 308 Checkers.

1. Download and install the JSR 308 implementation; follow the instructions at `http://groups.csail.mit.edu/pag/jsr308/dev/README-jsr308.html#installing`. JSR 308 extends the Java language to permit annotations to appear on types.
2. Download the JSR 308 Checkers distribution zipfile from `http://groups.csail.mit.edu/pag/jsr308/releases/jsr308-checkers.zip`, and unzip it to create a `checkers` directory. Example commands:

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

3. Add the `checkers/checkers.jar` file to your classpath. (If you do not do this, you will have to supply the `-cp checkers.jar` option whenever you run `javac` and use a checker plugin.)
4. Test that everything works:
   - Run the NonNull checker examples (see Section 3.5.1).
   - Run `ant all-tests` in the `checkers` directory:
     ```
     ant all-tests
     ```

You can use the checkers framework in an IDE such as Eclipse by setting the external builder to `javac`. (A checkers implementation builds on standard mechanisms such as JSR 269 annotation processing, but also accesses the compiler's AST. In the long run, a checker built using the checkers framework should not be dependent on any compiler specifics.) If you do not place the annotations in comments, as in `/*@NonNull*/ String` (see also 2.1), then you should also disable Eclipse's on-the-fly syntax checking.

1.1.1 Building

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.

First, 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`.)

To build `checkers.jar`, 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.
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 [EC07]. 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) { ... }  // argument
String toString() @ReadOnly { ... }           // receiver
@NonNull List<@Interned String> messages;     // generic argument
@NonNull String[@Interned] messages;          // non-null array of interned Strings
myDate = (@ReadOnly Date) readonlyObject;     // cast
```

For backward compatibility, you may write any annotation inside a `/*...*/` Java comment. The JSR 308 compiler will recognize such an annotation, but your code will still compile with pre-JSR-308 compilers. This is useful when your code needs to be compilable by people who are not using the JSR 308 compiler.

When writing source code with annotations, typically you will add

```
import checkers.quals.*;
```

at the top of the source file, so that you can write annotations such as `@NonNull` instead of `@checkers.quals.NonNull`.

The JSR 308 `javac` compiler also recognizes imports for the annotations as a system property `jsr308_imports`. The system variable can be set as command line argument `-Djsr308_imports="checkers.quals.*"` or as an environment variable (i.e., through commands `set` or `export`). This feature enables you to compile your code with a pre-JSR 308 compiler without requiring the annotation definitions on your classpath, and allows for distribution of the code without the annotations packages.

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 NonNull 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 7.1.2.

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 NonNull 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 JSR 308 Checkers distribution contains such annotations for popular libraries, such as the JDK. Section 7.1.2 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 7.
- You can annotate the compiled `.jar` or `.class` files by using the annotation file utilities (`http://groups.csail.mit.edu/pag/jsr308/annotation-file-utilities/`) to express the annotations textually and then 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 JSR 308 Checkers; see Section 2.7.

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 NonNull checker (Section 3) uses `assert` statements that contain null checks, along with the flow-sensitive type inference (Section 2.5), 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, some checkers treat certain variables and expressions as being annotated, even if you have not annotated them. For instance, the NonNull checker (Section 3) can automatically determine that certain variables are nonnull, without you having to annotate them. The Interned checker (Section 4) also has this functionality, and new checkers that you write can also take advantage of it.

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 NonNull 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 2.7) 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 non-null 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 checkers 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 How to report bugs

If you have any problems with any checker, or with the checkers framework, please let us know at `jsr308-bugs@lists.csail.mit.edu`
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 JSR 308 Checkers you are using. You can determine the JSR 308 version by running `javac -version`.

2.8 Credits and changelog

The JSR 308 Checkers distribution was developed in the MIT Program Analysis Group. The JSR 308 checkers 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 custom 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/dev/changelog-checkers.txt`.

3 NonNull checker

If the NonNull 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 NonNull 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.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.quals.NonNull` 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.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.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 NonNull checker itself is annotated with `@NonNull`.

In addition, you can run the NonNull checker on the annotation scene library, another library that has been fully annotated with `@NonNull`. To run the NonNull 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 NonNull 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 NonNull 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 JSR 308 Checkers `@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.)

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`.

5 Javari checker

Javari 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 NonNull 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
```

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 [EC07] 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 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.

7.1 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 JSR 308 Checkers distribution.)

First, you must install the skeleton class generator (Section 7.2).

7.1.1 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 7.2 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();
   ```

7.1.2 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.

7.2 Installing the skeleton class generator

Source code for the skeleton class generator tool is included in the checkers 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 checkers 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).

7.3 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.

8 How to create a new checker plugin

This section describes how to create a checker — a type-checking compiler plugin that detects bugs or verifies their absence. After a programmer annotates a program using JSR 308 annotations, 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.

Before you create the checker, you should decide its semantics (its rules for checking programmer-written annotations), and you must define the annotations.

There are two ways to create a checker. One way is to use the custom checker (Section 8.1, possibly adding some meta-annotations to your annotation declarations. This approach does not require you to write any Java code, and is sufficient for many simple type qualifiers. The other way is to write Java code to augment any meta-annotations you have written. This approach permits you to create more powerful checkers, and is discussed starting in Section 8.2.

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 Checkers Framework, or to create your checker by modifying another one.

8.1 The Custom checker

The checkers distribution includes the Custom checker, which performs type-checking with no special semantics beyond standard subtyping rules and operates over annotations specified by a user on the command line.

The Custom checker is ideal for type systems that do not require special checks (e.g., warning about dereferences of possibly-null values). For such type systems, the type system creator is encouraged to use the Custom checker and does not need to write any code beyond declarations for the annotations used by the type system.

The Custom checker can accomodate for type systems that require implicit annotations (e.g. literals are implicitly considered `@NonNull`). The type system creator could such default implicit annotations by using `@ImplicitFor`, as described in Section 8.5.1.

The Custom checker is also useful to type system creators that wish to experiment with a checker before writing code; the Custom checker demonstrates the functionality that a checker inherits from the checkers framework.

8.1.1 Using the Custom checker

The Custom 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` meta-annotation used by other checkers (see section 8.5.1).

Note that the annotation provided via the command-line 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.

8.1.2 Custom 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 Custom checker for the `Encrypted` type system, first define an annotation for the `Encrypted` qualifier:

```
package myquals;

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

Then, write add `@Encrypted` annotations to 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);
}
```

Finally, invoke the compiler with the Custom checker, specifying the `@Encrypted` annotation using the `-Aquals` option:

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

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

8.2 Classes in a checker plugin

A checker consists of three classes: a visitor, a type factory, and a compiler interface. The Checkers Framework provides abstract base classes (default implementations), and a specific checker overrides as little or as much of the default implementations as necessary (see Sections 8.3, 8.4, and 8.5).

The *visitor* class performs type-checking at each node of the source file's AST. The abstract *base visitor* issues a warning whenever the type system induced by the type qualifier is violated. For example, it is illegal to assign a supertype to a subtype in Java, so this assignment is not permitted (assuming the obvious variable declarations):

```
  myNonNullObject = myObject;  // invalid assignment
```

In addition to assignments, the base visitor checks method arguments, receivers, return values, overriding, and other Java constructs. The base visitor also provides hooks that are called by the annotation processing facility [Dar06], and it reports errors via the Java compiler's messaging mechanism [vdA06].

The *type factory* class, given an AST node, returns the annotated type of that expression. The abstract *base type factory* class provided by the Checkers Framework 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.

The *compiler interface* class performs all subtyping tests, including accounting for arrays, generics, wildcards, etc. A programmer supplies the compiler interface class name as a javac `-typeprocessor` argument, so the compiler interface usually has a name like `NonNullChecker` or `InternedChecker`.

8.3 Extending the visitor class `BaseTypeVisitor`

The visitor class uses the visitor design pattern to traverse Java source syntax trees (as provided by the semi-public Tree API and not the internal javac tree representation). The base class `checkers.basetype.BaseTypeVisitor` type-checks each AST node as it is visited.

By default, `BaseTypeVisitor` performs assignment and psudo-assignment checks, in a similar fashion to Java subtype rules, but with taking the type qualifiers into account. `BaseTypeVisitor` issues seven errors:

- invalid assignment (type.incompatible) when an assignment from an expression type to an incompatible type. The assignment may be a simple assignment, or psudo-assignment like return expressions or argument passing in a method invocation
- 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 overriden method's declaration

The visitor overrides one method in the base visitor for each special rule in the type qualifier system. Many type-checkers need to 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
```

The class `BaseTypeVisitor` is a wrapper around `TreePathScanner` for performing type-checking using the annotation processing API and the `Annotated*Type` classes. To extend `BaseTypeVisitor`, override the appropriate `visit*` method from `TreeScanner` (these methods have specific tree nodes for parameters, i.e., `visitAssignment` has an argument of type `AssignmentTree`). The protected member `AnnotatedTypeFactory` factory can be used to create `AnnotatedTypeMirrors` for querying the annotations on/in a tree node.

For example, to test for dereferecing of possibly nullable values, one would need to override `visitMemberSelect()`, as the following:

```
@Override
public Void visitMemberSelect(MemberSelectTree node, Void p) {
    AnnotatedTypeMirror type = factory.getAnnotatedType(node.getExpression());
    if (!type.hasAnnotation(NONNULL))
        checker.report(Result.failure("deref.invalid"), node);
    return super.visitMemberSelect(node, p);
}
```

8.4 Extending the type factory `AnnotatedTypeFactory`

The "Annotated Types" framework in checkers.types can be used to obtain annotated on tree nodes. The `AnnotatedTypeFactory` class has `getAnnotatedType` methods that take either a tree node or an element and return an `AnnotatedTypeMirror`.

Type system checker writer need to subclass `AnnotatedTypeFactory` to account for implicit type qualifiers. For example, the Interned checker (Section 4) has a type factory that treats every `String` literal, such as `"JSR 308"`, as having type `@Interned String`, and treats every primitive type value as `@Interned` (because Java guarantees that property). To express this property, the `Interned` type factory can specify it by overriding `annotateImplicit(Tree,AnnotatedTypeMirror)` in the following manner:

```
@Override
protected void annotateImplicit(Tree tree, AnnotatedTypeMirror type) {
  if (tree.getKind() == Tree.Kind.STRING_LITERAL)
    type.addAnnotation(INTERNED);
  if (type.getKind().isPrimitive())
    type.addAnnotation(INTERNED);
}
```

However, type system writers may use `@ImplicitFor` meta-annotation to declare implicit annotation, as described in Section 8.5.1.

8.5 Extending the compiler interface `BaseTypeChecker`

The base class for checkers is `checkers.basetype.BaseTypeChecker`, which subclasses of Sun's `AbstractProcessor`. The abstract *base compiler interface* invokes the visitor class on each input source file.

The compiler interface defines the type hierarchy by providing a method for determining the relationship between two types with respect to the checker type system. Section 8.5.1 describes a mechanism for easily defining this relationship.

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 need to be in the same directory as the `Checker`.

It is a convention that the compiler interface (checker), the visitor, and the annotated type factory are named as `FOOChecker`, `FOOVisitor`, `FOOAnnotatedTypeFactory`, for a type system FOO. `BaseTypeVisitor` uses the convention to reflectively construct the components. Otherwise, the checker writers need to specify the component classes for construction.

Additionally, as recommended by the annotation processing API, checker classes may be annotated with the SupportedAnnotationTypes and SupportedSourceVersion annotations.

8.5.1 Meta-annotations for qualifier hierarchy

Checkers that extend the `BaseTypeChecker` class may specify the relationships between qualifiers in a type system by adding meta-annotations to the declarations of the annotations for these qualifiers. This allows for a declarative specification of the qualifier hierarchy, and it obviates overriding methods related to subtyping in the checker implementation, since the base implementation of `BaseTypeChecker` can deduce type relationships from the qualifier hierarchy.

The five meta-annotations, defined in the `checkers.metaquals` package, are used as follows:

- `@TypeQualifier`: denotes an annotation that is used as a type qualifier. The framework ignores any annotations whose declarations do not bear this annotation (for instance, `@Default`, `@Deprecated`, or `@SuppressWarnings`). For the NonNull checker, the annotations `@NonNull` and `@Nullable` are both annotated with `@TypeQualifier`.

- `@SubtypeOf`: denotes that a qualifier is the subtype of another qualifier or qualifiers, specified as an array of class literals. For instance, for the NonNull checker, the annotation `@NonNull` is annotated with `@SubtypeOf({Nullable.class})`, since for some type T, `@NonNull` T is a subtype of `@Nullable` T.
- `@QualifierRoot`: denotes that a qualifier is the supertype of all other qualifiers in the type hierarchy. For instance, for the NonNull checker, the annotation `@Nullable` is annotated with `@QualifierRoot`, and for the Javari checker, the annotation `@ReadOnly` is annotated with `@QualifierRoot`.
- `@TypeQualifiers`: unlike the others, this annotation is written on the `Checker` class that extends `BaseTypeChecker`; it denotes the set of type qualifiers — annotations with the `@TypeQualifier` meta-annotation — that make up the type hierarchy for this checker, provided as an array of class literals. For instance, the `NonNullChecker` class is annotated with `@TypeQualifiers({NonNull.class, Raw.class, Nullable.class})`; the `InternedChecker` class is annotated with `@TypeQualifiers({Interned.class})`.
- `@ImplicitFor`: written on a qualifier, this annotation 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 members of the `com.sun.source.tree.Tree.Kind` enum, e.g., `NEW_ARRAY` or `METHOD_INVOCATION`
  - `types`: an array of members of the `javax.lang.model.type.TypeKind` enum, 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.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.)

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.

8.6 Example checker: `Interned`

Here is the actual source code for the `Interned` checker of Section 4.

Listing 1: Interned.java

```
package checkers.quals;

import static java.lang.annotation.ElementType.*;

import java.lang.annotation.*;

import checkers.interned.InternedChecker;
import checkers.metaquals.*;
import checkers.types.AnnotatedTypeMirror.AnnotatedPrimitiveType;

import com.sun.source.tree.LiteralTree;

/**
 * Indicates that a variable has been interned, i.e., that the variable refers
 * to the canonical representation of an object.
 *
 * <p>
 *
 * This annotation is associated with the {@link InternedChecker}.
 *
 * @see InternedChecker
```

```
 * @manual #interned Interned Checker
 */
@Documented
@TypeQualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({FIELD, LOCAL_VARIABLE, METHOD, PARAMETER, TYPE})
@ImplicitFor(
    treeClasses={LiteralTree.class},
    typeClasses={AnnotatedPrimitiveType.class})
public @interface Interned {

}
```

Listing 2: InternedChecker.java

```
package checkers.interned;

import checkers.basetype.*;
import checkers.metaquals.TypeQualifiers;
import checkers.quals.Interned;
import checkers.source.*;

import javax.annotation.processing.*;
import javax.lang.model.*;

/**
 * A typechecker plug-in for the {@link checkers.quals.Interned} qualifier that
 * finds (and verifies the absence of) equality-testing and interning errors.
 *
 * <p>
 *
 * The {@link checkers.quals.Interned} annotation indicates that a variable
 * refers to the canonical instance of an object, meaning that it is safe to
 * compare that object using the "==" operator. This plugin suggests using "=="
 * instead of ".equals" where possible, and warns whenever "==" is used in cases
 * where one or both operands are not {@link checkers.quals.Interned}.
 *
 * @manual #interned Interned checker
 */
@SupportedAnnotationTypes({"*"})
@SupportedSourceVersion(SourceVersion.RELEASE_7)
@SupportedLintOptions({"dotequals", "flow"})
@SuppressWarningsKey("interned")
@TypeQualifiers({ Interned.class })
public final class InternedChecker extends BaseTypeChecker { }
```

Listing 3: InternedVisitor.java

```
package checkers.interned;

import java.util.*;

import checkers.source.*;
import checkers.basetype.*;
import checkers.types.*;
import checkers.util.*;
import com.sun.source.tree.*;
import com.sun.source.util.*;

import javax.lang.model.element.*;
import javax.lang.model.type.*;
import static javax.lang.model.util.ElementFilter.*;

/**
```

```
 * A type-checking visitor for the {@link checkers.quals.Interned} type
 * qualifier that uses the {@link BaseTypeVisitor} implementation. This visitor
 * reports errors or warnings for violations for the following cases:
 *
 * <ul>
 *  <li>if both sides of a "==" or "!=" comparison are not Interned (error
 *  "not.interned")</li>
 *  <li>if the receiver and argument for a call to an equals method are both
 *  Interned (optional warning "unnecessary.equals")</li>
 * </ul>
 *
 *
 * @see BaseTypeVisitor
 */
public class InternedVisitor extends BaseTypeVisitor<Void, Void> {

    /** The interned annotation. */
    private final AnnotationMirror INTERNED;
    private final AnnotatedTypeMirror INTERNED_OBJECT;

    /**
     * Creates a new visitor for type-checking {@link checkers.quals.Interned}.
     *
     * @param checker the checker to use
     * @param root the root of the input program's AST to check
     */
    public InternedVisitor(InternedChecker checker, CompilationUnitTree root) {
        super(checker, root);
        this.INTERNED = annoFactory.fromName("checkers.quals.Interned");
        INTERNED_OBJECT = AnnotatedTypeMirror.createType(types.getDeclaredType(
                    elements.getTypeElement("java.lang.Object")),
                    checker.getProcessingEnvironment(), factory);
        INTERNED_OBJECT.addAnnotation(INTERNED);
    }

    @Override
    public Void visitBinary(BinaryTree node, Void p) {

        // No checking unless the operator is "==" or "!=".
        if (!(node.getKind() == Tree.Kind.EQUAL_TO ||
                node.getKind() == Tree.Kind.NOT_EQUAL_TO))
            return super.visitBinary(node, p);

        Tree leftOp = node.getLeftOperand(), rightOp = node.getRightOperand();

        // Check passes if one arg is null.
        if (leftOp.getKind() == Tree.Kind.NULL_LITERAL ||
                rightOp.getKind() == Tree.Kind.NULL_LITERAL)
            return super.visitBinary(node, p);

        // Heuristically check that the comparison is the member of a class of
        // comparisons that should be skipped.
        if (suppressByHeuristic(node))
            return super.visitBinary(node, p);

        AnnotatedTypeMirror left = factory.getAnnotatedType(leftOp);
        AnnotatedTypeMirror right = factory.getAnnotatedType(rightOp);

        // Check passes due to auto-unboxing.
        if (left.getKind().isPrimitive() || right.getKind().isPrimitive())
            return super.visitBinary(node, p);

        if (!checker.isSubtype(INTERNED_OBJECT, left))
            checker.report(Result.failure("not.interned", left), leftOp);
        if (!checker.isSubtype(INTERNED_OBJECT, right))
```

```
            checker.report(Result.failure("not.interned", right), rightOp);

        return super.visitBinary(node, p);
    }

    @Override
    public Void visitMethodInvocation(MethodInvocationTree node, Void p) {
        if (isInvocationOfEquals(node)) {
            AnnotatedTypeMirror recv = factory.getReceiver(node);
            AnnotatedTypeMirror comp = factory.getAnnotatedType(node.getArguments().get(0));

            if (this.checker.getLintOption("dotequals", true)
                    && checker.isSubtype(INTERNED_OBJECT, recv)
                    && checker.isSubtype(INTERNED_OBJECT, comp))
                checker.report(Result.warning("unnecessary.equals"), node);
        }

        return super.visitMethodInvocation(node, p);
    }

    /**
     * Tests whether a method invocation is an invocation of
     * {@link Object#equals}.
     *
     * @param node a method invocation node
     * @return true iff {@code node} is a invocation of {@code equals()}
     */
    private boolean isInvocationOfEquals(MethodInvocationTree node) {
        ExecutableElement method = TreeUtils.elementFromUse(node);
        return (method.getParameters().size() == 1
                && method.getReturnType().getKind() == TypeKind.BOOLEAN
                && method.getSimpleName().contentEquals("equals"));
    }

    /**
     * Heuristically determines whether checking for a particular comparison
     * should be suppressed. Specifically, this method tests the following:
     *
     * <ul>
     * <li>the comparison is a == comparison, and</li>
     *
     * <li>it is the test of an if statement that's the first statement in the method,
     * and</li>
     *
     * <li>one of the following is true:
     *
     * <ul>
     * <li>the method overrides {@link Comparator#compare}, the "then" branch
     * of the if statement returns zero, and the comparison tests equality of
     * the method's two parameters</li>
     *
     * <li>the method overrides {@link Object#equals(Object)} and the
     * comparison tests "this" against the method's parameter</li>
     * </ul>
     *
     * </li>
     * </ul>
     *
     * @param node the comparison to check
     * @return true if one of the supported heuristics is matched, false
     *         otherwise
     */
    private boolean suppressByHeuristic(final BinaryTree node) {

        // Only valid if called on an == comparison.
```

```
if (node.getKind() != Tree.Kind.EQUAL_TO)
    return false;

Tree left = node.getLeftOperand();
Tree right = node.getRightOperand();

// Only valid if we're comparing identifiers.
if (!(left.getKind() == Tree.Kind.IDENTIFIER
        && right.getKind() == Tree.Kind.IDENTIFIER))
    return false;

// If we're not directly in an if statement in a method (ignoring
// parens and blocks), terminate.
if (!Heuristics.matchParents(getCurrentPath(), Tree.Kind.IF, Tree.Kind.METHOD))
    return false;

// Determine whether or not the "then" statement of the if has a single
// "return 0" statement (for the Comparator.compare heuristic).
final boolean returnsZero =
    Heuristics.applyAt(getCurrentPath(), Tree.Kind.IF, new Heuristics.Matcher() {

        @Override
        public Boolean visitIf(IfTree tree, Void p) {
            return visit(tree.getThenStatement(), p);
        }

        @Override
        public Boolean visitBlock(BlockTree tree, Void p) {
            if (tree.getStatements().size() > 0)
                return visit(tree.getStatements().get(0), p);
            return false;
        }

        @Override
        public Boolean visitReturn(ReturnTree tree, Void p) {
            ExpressionTree expr = tree.getExpression();
            return (expr.getKind() == Tree.Kind.INT_LITERAL &&
                    ((LiteralTree)expr).getValue().equals(0));
        }
    });


ExecutableElement enclosing =
    TreeUtils.elementFromDeclaration(visitorState.getMethodTree());
assert enclosing != null;

Element lhs = TreeUtils.elementFromUse((IdentifierTree)left);
Element rhs = TreeUtils.elementFromUse((IdentifierTree)right);

if (returnsZero && overrides(enclosing, "java.util.Comparator", "compare")) {
    assert enclosing.getParameters().size() == 2;
    Element p1 = enclosing.getParameters().get(0);
    Element p2 = enclosing.getParameters().get(1);
    return (p1.equals(lhs) && p2.equals(rhs))
        || (p2.equals(lhs) && p1.equals(rhs));

} else if (overrides(enclosing, "java.lang.Object", "equals")) {
    assert enclosing.getParameters().size() == 1;
    Element param = enclosing.getParameters().get(0);
    Element thisElt = getThis(this.getCurrentPath());
    assert thisElt != null;
    return (thisElt.equals(lhs) && param.equals(rhs))
        || (param.equals(lhs) && thisElt.equals(rhs));
}
```

```
        return false;
    }

    /**
     * Determines the element corresponding to "this" inside a scope.
     *
     * @param path the path to a tree inside the desired scope
     * @return the element corresponding to "this" in the scope of the tree
     *         given by {@code path}
     */
    private final Element getThis(TreePath path) {
        for (Element e : trees.getScope(path).getLocalElements())
            if (e.getSimpleName().contentEquals("this"))
                return e;
        return null;
    }

    /**
     * Determines whether or not the given element overrides the named method in
     * the named class.
     *
     * @param e an element for a method
     * @param clazz the name of a class
     * @param method the name of a method
     * @return true if the method given by {@code e} overrides the named method
     *         in the named class; false otherwise
     */
    private final boolean overrides(
            ExecutableElement e, String clazz, String method) {

        // Get the element named by "clazz".
        TypeElement comp = elements.getTypeElement(clazz);
        if (comp == null) return false;

        // Check all of the methods in the class for name matches and overriding.
        for (ExecutableElement elt : methodsIn(comp.getEnclosedElements()))
            if (elt.getSimpleName().contentEquals(method)
                    && elements.overrides(e, elt, comp))
                return true;

        return false;
    }
}
```

Listing 4: InternedAnnotatedTypeFactory.java

```
package checkers.interned;

import javax.lang.model.element.*;

import checkers.basetype.BaseTypeChecker;
import checkers.flow.Flow;
import checkers.quals.Interned;
import checkers.types.*;
import static checkers.types.AnnotatedTypeMirror.*;

import checkers.util.*;

import com.sun.source.tree.*;

/**
 * An {@link AnnotatedTypeFactory} that accounts for the properties of the
 * Interned type system. This type factory will add the {@link Interned}
 * annotation to a type if the input:
 *
```

```
 * <ul>
 * <li>is a String literal
 * <li>is a class literal
 * <li>is an enum constant
 * <li>has a primitive type
 * <li>has the type java.lang.Class
 * <li>is a call to the method {@link String#intern()}
 * </ul>
 */
public class InternedAnnotatedTypeFactory extends AnnotatedTypeFactory {

    /** Adds annotations from tree context before type resolution. */
    private final TreeAnnotator treeAnnotator;

    /** Adds annotations from the resulting type after type resolution. */
    private final TypeAnnotator typeAnnotator;

    /** The {@link Interned} annotation. */
    final AnnotationMirror INTERNED;

    /** Flow-sensitive qualifier inference. */
    private final Flow flow;

    /**
     * Creates a new {@link InternedAnnotatedTypeFactory} that operates on a
     * particular AST.
     *
     * @param checker the checker to use
     * @param root the AST on which this type factory operates
     */
    public InternedAnnotatedTypeFactory(InternedChecker checker,
        CompilationUnitTree root) {
        super(checker, root);
        this.INTERNED = annotations.fromName("checkers.quals.Interned");
        this.treeAnnotator = new TreeAnnotator(checker);
        this.typeAnnotator = new InternedTypeAnnotator(checker);

        this.flow = new Flow(checker, root, INTERNED, this);
        if (checker.getLintOption("flow", true)) flow.scan(root, null);
    }

    @Override
    protected void annotateImplicit(Element elt, AnnotatedTypeMirror type) {
        typeAnnotator.visit(type);
    }

    @Override
    protected void annotateImplicit(Tree tree, AnnotatedTypeMirror type) {
        treeAnnotator.visit(tree, type);
        if (flow.test(tree) == Boolean.TRUE)
            type.addAnnotation(INTERNED);
        typeAnnotator.visit(type);
    }

    /**
     * A class for adding annotations to a type after initial type resolution.
     */
    private class InternedTypeAnnotator extends TypeAnnotator {

        /** Creates an {@link InternedTypeAnnotator} for the given checker. */
        InternedTypeAnnotator(BaseTypeChecker checker) {
            super(checker);
        }

        @Override
```

```
        public Void visitDeclared(AnnotatedDeclaredType t, Void p) {

            // Enum types and constants: add an @Interned annotation.
            Element elt = types.asElement(t.getUnderlyingType());
            if (elt != null && elt.getKind() == ElementKind.ENUM)
                t.addAnnotation(INTERNED);

            // Annotate class types.
            TypeElement classElt = elements.getTypeElement("java.lang.Class");
            assert classElt != null;
            if (types.isSameType(types.erasure(t.getUnderlyingType()),
                    types.erasure(classElt.asType())))
                t.addAnnotation(INTERNED);

            return super.visitDeclared(t, p);
        }

        @Override
        public Void visitExecutable(AnnotatedExecutableType t, Void p) {

            // Annotate the java.lang.String.intern() method.
            if (t.getElement() != null) {
                ExecutableElement method = t.getElement();
                if (method.getSimpleName().contentEquals("intern")) {
                    Name clsName = ElementUtils.getQualifiedClassName(method);
                    if (clsName != null && clsName.contentEquals("java.lang.String"))
                        t.getReturnType().addAnnotation(INTERNED);
                }
            }

            return super.visitExecutable(t, p);
        }
    }

    @Override
    public AnnotatedPrimitiveType getUnboxedType(AnnotatedDeclaredType type) {
        AnnotatedPrimitiveType primitive = super.getUnboxedType(type);
        primitive.addAnnotation(INTERNED);
        return primitive;
    }
}
```

8.7 Testing framework

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

8.8 Debugging options

The checkers 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 8.3 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.quals.Interned java.lang.String
   expected: DECLARED @checkers.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.quals.Interned java.lang.String
   expected: DECLARED java.lang.String
1 error
```

8.9 Putting your checker in the repository

This section is relevant only if you wish to add your checker to the source code repository for the checkers framework — for example, to include your checker in the JSR 308 Checkers distribution.

The JSR 308 checkers appear in directory `annotations/checkers/` of the `annotations` repository. It contains the following relevant subdirectories:

- `manual/`: Documentation for your checker goes here.
- `src/checkers/quals/`: Definition of the annotation itself — that is, the `@interface` declaration.
- `src/checkers/`*`annotation_name`*`/`: Code for the checker, in a directory that is a sibling of of `quals/`, `nonnull/`, etc.
- `jdk/`*`annotation_name`*`/`: Annotated "skeleton class" versions of the JDK and other libraries (see Section 7).
- `tests/`*`annotation_name`*`/`: Inputs and outputs for the test suite for the checker. A single top-level test suite class goes in `tests/src/tests/`.

References

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

[EC07] Michael D. Ernst and Danny Coward. JSR 308: Annotations on Java types. `http://pag.csail.mit.edu/jsr308/`, November 9, 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.

[PAJ+07] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Pluggable type-checking for custom type qualifiers in Java. Technical Report MIT-CSAIL-TR-2007-047, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, September 17, 2007.

[vdA06] Peter von der Ahe. JSR 199: Java compiler API. `http://jcp.org/en/jsr/detail?id=199`, December 11, 2006.

[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.