

JSR 308 Type-checkers and Framework

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

February 11, 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)
- 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

Follow these steps to install the JSR 308 Checkers.

1. Download and install the JSR 308 implementation; follow the instructions at <http://groups.csail.mit.edu/pag/jsr308/>. 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.

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

We recommend that you do this in the parent of the JSR 308 `langtools` directory, so that the `checkers` and `langtools` directories are siblings. If the `checkers` and `langtools` directories are not siblings, then edit the `checkers/build.properties` file so that the `compiler.lib` property specifies the location of the JSR 308 `javac.jar` library. (If the directories are siblings, no edits are necessary.)

3. Build the `checkers.jar` library by running `ant` in the `checkers` directory.

```
cd checkers
ant
```

4. Add the `checkers.jar` file, which is produced in the `checkers` directory by a successful build, 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.)
5. Test that everything works:
 - Run the `NonNull` checker examples (see Section 3.4.1).
 - 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`, then you should also disable Eclipse's on-the-fly syntax checking.

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 [EC06]. 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.qual.*;
```

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

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 classfiles.

If you annotate additional libraries, please share them with us so that we can distribute the annotations with the JSR 308 Checkers; see Section 2.6.

2.4 Suppressing warnings

You may wish to suppress checker warnings because of un-annotated 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, or for other reasons. You can suppress warnings via the `@SuppressWarnings` annotation or the `checkers.skipClasses` Java property.

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.

2.5 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.
- 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.

2.6 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`.

2.7 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. 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. 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/dist/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 3.5 for caveats to the guarantee.

3.1 Annotating your code with @NonNull

In order to perform checking, you must annotate your code with the @NonNull type annotation, which indicates a type that does not include the null value.

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 nobj; // never null
...
nobj.toString() // checker warning: dereference might cause null pointer exception
nobj = obj;      // checker warning: nobj may become null
```

Parameter passing and return values are checked analogously to assignments.

3.2 NonNull default and @Nullable annotation

By default, the NonNull checker regards unannotated types as possibly-null (or “nullable”). This behavior may be overridden for individual classes or methods using the `@checkersquals.Default` annotation.

The `@Default` annotation has a single argument for the fully qualified `String` name of an annotation. If the NonNull checker finds a `@Default("checkersquals.NonNull")` annotation enclosing the scope in which it is currently checking, it treats all types in that scope as though they are annotated with `@NonNull`. The `@Nullable` annotation may be used to locally override the effects of the `@Default` annotation.

This example illustrates the use of the `@Default` and `@Nullable` annotations:

```
@Default("checkersquals.NonNull")
public boolean compile(File file) {
    if (!file.exists()) // no warning: file is @NonNull by default
        return false;

    @Nullable File srcPath = ...;
    // ...
    if (srcPath.exists()) // warning: srcPath might be null
        // ...
}
```

3.3 Implicit non-null types (flow-sensitive inference)

In order to reduce the burden of annotating types in your program with `@NonNull`, the NonNull checker treats some variables and expressions as having non-null types even if you have not annotated them. A variable or expression can be treated as `@NonNull` from the time that it is checked (e.g., via an assertion or `if` statement)

or assigned a non-null value, until it might be re-assigned (e.g., via a method call or via an assignment to a possibly-aliasing variable). The implicitly non-null types permit dereferences, and assignments to explicitly non-null types, without compiler warnings. (This treatment of assertions means that writing an `assert` statement is a way to suppress warnings in the NonNull checker.)

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 be non-null, but the NonNull checker does not do so, please submit a bug report (see Section 2.6) that includes a small piece of Java code that reproduces the problem.

Non-null 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 an un-annotated type can be treated as a `@NonNull` one. An expression of `@NonNull` type is never inferred to be treated as possibly-null; any such use is an error, and the checker issues a warning.

3.4 Examples

3.4.1 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`.

3.4.2 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.

3.5 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 2.5), there are two additional caveats:

- 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.
- The `NonNull` checker does not check whether a variable is initialized. That means that code executing before a variable is initialized — for example, in a constructor — can yield a null pointer exception that the checker does not warn about. (Fähndrich & Leino [FL03] present one possible solution, but we have not yet implemented it in the checker.)

3.6 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 `nonnull` 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 2.5 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 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
```

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

IGJ 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 2.5 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 `@RoMaybe`, 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 `@RoMaybe` annotation simulates mutability overloading. It can be applied to methods and parameters. Read the `@RoMaybe` Javadoc for more details.

5.2 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 2.5 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` is 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 [EC06] 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("Second") 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).

8 How to write a checker plugin

This document describes how to write a type-checking compiler plugin that detects bugs or verifies their absence. A programmer annotates a program using JSR 308 annotations, and the checker plug-in verifies that the code is consistent with the annotations.

8.1 Components of a plugin: checker class and visitor class

A checker plugin is centered around two classes: a *checker* class and a *visitor* class.

The checker class is the primary interface to `javac`'s annotation processing facility. The checker class provides the hooks invoked by the compiler during annotation processing, and methods for reporting errors via the compiler's internal messaging mechanism. The base class for checkers is `checkers.source.SourceChecker`. Typically, the checker class invokes the visitor class on each input source file.

The visitor class is a visitor for Java source syntax trees (as provided by the semi-public Tree API and not the internal `javac` tree representation). The base class for visitors is `checkers.source.SourceVisitor`. Typically, the visitor class performs type-checking as it walks each source file's AST.

8.2 Extending SourceChecker

SourceChecker has “provider” methods that allow the default SourceChecker behavior to be extended — some of these are inherited from Sun’s AbstractProcessor class, and others are defined in SourceChecker itself. These methods should be overridden accordingly. They are:

- `getMessages`: returns a `Properties` instance where the keys are the strings passed to `SourceChecker.report` (like “invalid.assignment”) and the values are the strings to be printed (“cannot assign ...”); this can be used to specialize messages when overriding a checker, and might also be used for internationalization
- `getSourceVisitor`: returns a `SourceVisitor` (in this case, the subclass of `SourceVisitor` specific to this plugin)
- `getFactory`: returns a custom subclass of `AnnotatedTypeFactory`, if this plugin requires it

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

8.3 Extending SourceVisitor

`SourceVisitor` is a wrapper around `TreePathScanner` for performing type-checking using the annotation processing API and the `Annotated*Type` classes.

To extend `SourceVisitor`, 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.

8.4 Using BaseTypeChecker and BaseTypeVisitor

`BaseTypeChecker` and `BaseTypeVisitor` in the `checker.basetype` package implement a generic type-checker for type qualifiers for which the qualified type is the subtype of the unqualified type. Many type qualifiers, including `@NonNull` and `@Interned`, fall into this category.

`BaseTypeChecker` extends `SourceChecker`, and it provides two primary services:

- an overridden `getSourceVisitor` method that returns an instance of `SubtypeVisitor`
- the `isSubtype` method that checks if one type is a subtype of another with respect to the type qualifier-annotations on the type

`BaseTypeVisitor` extends `SourceVisitor`, providing a type-checking visitor implementation that currently checks and reports six errors:

- invalid assignment (`assignment.invalid`) when an assignment from an unqualified type to a qualified supertype is found
- invalid argument (`argument.invalid`) when an argument with the unqualified type is passed to a method for a parameter with the qualified type
- invalid receiver (`receiver.invalid`) when a method whose receiver has the qualified type is called from an object with the unqualified type
- invalid return (`return.invalid`) when the expression in a return statement has the unqualified type but the method declaration has the qualified return 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

Many type-checkers need to override only a few methods in `BaseTypeVisitor`.

8.4.1 Customizing BaseTypeVisitor Error Messages

By overriding the `getMessages` method, a checker built on `BaseTypeChecker` can customize the error messages produced by the `BaseTypeVisitor`. The keys for each message are shown in parentheses in list of errors in the previous section; the overridden `getMessages` method must return a `java.util.Properties` object that maps each key to the desired error message.

8.5 Using AnnotatedTypeMirror

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

8.6 Example: Interned

This is an example of a method from an example subclass of `SourceVisitor` for checking a `@Interned` annotation.

```
// Checks that both the left- and right-hand operands for a
// binary operator "a == b" (or "a != b") are @Interned.
@Override
public Void visitBinary(BinaryTree node, Void p) {

    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();

    if (leftOp.getKind() == Tree.Kind.NULL_LITERAL ||
        rightOp.getKind() == Tree.Kind.NULL_LITERAL)
        return super.visitBinary(node, p);

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

    if (!left.hasAnnotation(INTERNEDED) || !right.hasAnnotation(INTERNEDED))
        checker.report(Result.failure("not.interned"), node);

    return super.visitBinary(node, p);
}
```

8.7 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 `quals/`, `nonnull/`, etc.
- `jdk/annotation_name/`: Annotated “skeleton class” versions of the JDK and other libraries (see Section 7).
- `tests/annotation_name/`: Inputs for the test suite for the checker.
- `tests/src/tests/`: The test suite class for the checker.

When writing your checker, you may wish to use an existing checker as an example.

References

- [EC06] Michael D. Ernst and Danny Coward. JSR 308: Annotations on Java types. <http://pag.csail.mit.edu/jsr308/>, October 17, 2006.
- [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.
- [ZPA⁺07] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kiezun, 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.