Status report on JSR-305: Annotations for Software Defect Detection

David Hovemeyer
Dept. of Physical Sciences
York College of Pennsylvania
dhovemey@ycp.edu

William Pugh
Dept. of Computer Science
Univ. of Maryland
pugh@cs.umd.edu

Abstract

Java Specification Request 305 defines a set of annotations that can understood by multiple static analysis tools. Rather than push the bleeding edge of static analysis, this JSR represents an attempt to satisfy different static analysis tool vendors and address the engineering issues required to make these annotations widely useful.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program analysis; D.2.4 [*Software/Program Verification*]: Reliability

General Terms Reliability, Security

Keywords static analysis, bugs, Java, software quality, specifications

1. Introduction

This poster gives a status report on Java Specification Request 305: Annotations for Software Defect Detection [5]. This JSR defines standard annotations such as @Nonnull that can be used to annotate Java code to better describe the intent of the designers/developers in ways that can be checked by tools. Such annotations have proved to be useful in tools such as FindBugs [2] and IntelliJ [4], but developers are hesitant to invest time in adding annotations which might not be portable across tools.

Rather than push the bleeding edge of static analysis [1, 7], this JSR represents an attempt to make these annotations widely useful for a variety of tools. JSR-305 is designed to require only standard Java SE 5 annotation support; however, in the future, JSR-308 [6] (which proposes language changes to allow annotations on Java types) would allow JSR-305 annotations to be used in a more general way.

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.
ACM 978-1-59593-865-7/07/0010.

At the time of the submission, JSR-305 is still being revised. By the time of the poster presentation, we hope to be in a nearly final draft form.

2. Standard Annotations

JSR-305 will define standard annotations that can be interpreted by multiple tools. Currently, existing tools use a variety of annotations to express whether or not a value can be null. JSR-305 must address mundane issues such as the name of the annotation used to denote something that can't be null: @NonNull, @NotNull or @Nonnull. Another question is whether and how we should provide a distinct treatment for (1) values with unknown nullness, and (2) values for which a null value is definitely possible and thus must not be unconditionally dereferenced. (An example of the second kind of value is the parameter to an `equals(Object)` method, which is required to handle a null argument by returning false.)

In addition to annotations for nullness, annotations for the following concepts are planned.

2.1 Numeric ranges and Unsigned values

At a minimum, we'll want to provide annotations that denote which values must be unsigned (non-negative), since code that cannot correctly handle negative values is relatively common. We could also provide annotations that allow a specification of ranges (e.g., that a value must be in the range $0 \ldots 9$), but there are fewer use cases for such an annotation.

2.2 Patterns and Syntax

An annotation could describe the legal values for a string or character sequence. The annotation might be given as a regular expression, or be described as the syntax in some particular domain specific language (e.g., that the String must be the string representation of a regular expression, a format string, or SQL).

2.3 Methods with important return values

In Java, there are a number of methods where it doesn't make sense to invoke the method without using the return value.

An example is `String.toLowerCase()`. Since Java Strings are immutable, this method returns the lower case version of the String. If the method is invoked and the return value is ignored, it almost always represents an error where the programmer thought the method would modify the String. Such errors are more common than might be expected, so having a standard @CheckReturnValue annotation would thus be useful.

2.4 Closing responsibility

When a method takes an InputStream as an argument (or any other kind of object representing a resource that must be closed), figuring out who is responsible for calling the `close()` method on the object can be tricky. We will provide annotations that allow a specification of whether a method is responsible for closing a value passed as a parameter.

3. Default and Inherited annotations

A number of studies have suggested that a majority of method parameters are designed to nonnull, and passing null argument values to such parameters will cause errors at runtime. However, many parameters can handle null parameters, and a static analysis that assumed all unannotated parameters must not be null would tend to generate a significant number of warnings, many of which do not reflect errors in the code.

This observation suggests that a mechanism for specifying a *default* nullness annotation for parameters is needed. A default annotation (typically @Nonnull) would be applied to a larger scope, such as a package or class, setting the default nullness annotation for all parameters in that scope. Any parameters requiring a different annotation could be annotated explicitly, overriding the default annotation. In general, a mechanism for specifying default annotations can significantly reduce the developer's annotation burden.

The issue of inherited annotations arises when a subclass method overrides a method defined in a superclass or interface. Any annotations pertaining to the superclass or interface method are inherited by the subclass method. Two concerns must be addressed: under what circumstances the subclass method may strengthen or weaken an inherited annotation, and what happens when annotations are inherited from multiple supertypes.

4. Concurrency Annotations

JSR-305 will define some annotations designed to assist in documenting threading/concurrency design intent and finding concurrency errors. Examples would be annotations on which methods should only be invoked from particular threads (e.g., methods that should only be invoked from the Swing event thread), which classes are intended to be work correctly even if methods are simultaneously invoked on an object from separate threads, and which fields should be protected by which locks.

5. User defined type qualifiers and validators

One of the most interesting aspects of JSR-305 is that it defines meta-annotations, which allow developers to define their own annotations to be interpreted by static analysis tools. Generally, these annotations are type qualifiers [3]: they provide a distinct and parallel type system that allows detection of additional errors. In many places throughout existing JDK libraries, int or String parameters are used where enum values would be a more appropriate and type safe choice. An example is the `createStatement` method in java.sql.Connection, which takes 3 int parameters, each of which should be one of a set of predefined static fields (e.g., ResultSet.TYPE_FORWARD_ONLY). However, since these parameters are typed as ints, no error is generated if the parameters are provided in the wrong order. By defining an annotation @ResultSetType, and using this on both the method parameter and the final static fields, errors such as accidently providing a ResultSetConcurrency value where a ResultSetType value is expected can be detected using static analysis.

User defined type qualifiers can also provide validators: these are classes associated with a type qualifier that can check to see if a particular value is consistent with that type qualifier. For example, a type qualifier @CreditCardNumber could be defined that would check to see if a particular value was a valid credit card number. This could be applied by static analysis when constant values are supplied, or invoked at runtime by dynamic tools.

References

[1] P. Chalin. Early detection of jml specification errors using esc/java2. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, pages 25–32, New York, NY, USA, 2006. ACM Press.

[2] FindBugs. `http://findbugs.sourceforge.net`, 2007.

[3] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 192–203, New York, NY, USA, 1999. ACM Press.

[4] IntelliJ. `http://www.jetbrains.com/idea/`, 2007.

[5] JSR 305: Annotations for Software Defect Detection. `http://jcp.org/en/jsr/detail?id=305`, 2007.

[6] JSR 308: Annotations on Java Types. `http://jcp.org/en/jsr/detail?id=308`, 2007.

[7] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.