Custom type qualifiers via annotations on Java types

Matthew M. Papi and Michael D. Ernst
mernst@cs.washington.edu
March 1, 2018

March 1, 2018

A separate document, "Type annotations (JSR 308)", presents a proposal for extending the syntax of Java (and the class file format) to permit annotations on any use of a type in Java. By contrast, the current Java standard only permits annotations on declarations, which is less general and restricts how annotations can be utilized.

The "Type annotations" document specifies the syntax of Java annotations, but not their semantics. The semantics of each annotation is defined by its author, who also creates plug-ins to check and enforce those semantics. Those plug-ins can operate at compile time and load time — making guarantees about run-time behavior requires load-time checking, but compile-time checking is useful as an early check and is more convenient for programmers.

This document explores implementation strategies for plug-ins that check annotations. These strategies are equally valid for Java's original annotations and for the new extended annotations. Both compile-time and load-time processing are outside the scope of the annotations proposal, but are presented here as an example of how annotations on types might be used. The processors described here do not require any further changes to Java beyond those described in the "Type annotations" document.

1 Compile-Time Processing

A compiler plug-in that enforces annotation semantics (for type qualifiers or any other use) can be written using JSR-269 (the pluggable annotation processing API) and the Tree API, which were defined for just such a purpose. The Tree API allows an application to obtain and traverse (but not modify) an instance of a program's abstract syntax tree. JSR-269 provides a set of classes for working with annotations, as well as an interface to the compiler so that annotation processing code can be run, and errors issued, at compile-time. These interfaces must be slightly extended to accommodate the new locations for annotations.

A plug-in developer can create a class that extends JSR-269's `AbstractProcessor` and provide an implementation of its `process` method. During compilation, the compiler calls `process` when it encounters annotations, passing these annotations as arguments. The `process` method's implementation can use the Tree API to obtain and analyze the program elements surrounding the annotation. If the plug-in discovers a violation of a type qualifier's semantics, the `process` method can use the JSR-269 `Messager` class to report the violation as an error or warning.

A framework for writing compiler plug-ins is outside the scope of this proposal. After obtaining some experience with compiler plug-ins, we or others may propose support for writing them, perhaps in the spirit of [CMM05].

In keeping with Java's separate compilation model, the Tree API gives access to the AST for all `.java` files that are being compiled; a plug-in has access only to signatures for code that is read from `.class` files. A whole-program analysis might need to use a separate infrastructure, and is outside the scope of this proposal.

2 Load-Time Processing

Whenever compile-time checking is used for a given annotation, then load-time checking is required as well. This is exactly how Java already works. The byte-code verifier is the key type-checker for Java, which enforces the guarantees of the type system and prevents certain run-time errors. By contrast, the Java compiler (e.g., `javac`) is a programmer convenience that provides early warnings but cannot make guarantees on its own.

Load-time checking is necessary for two general reasons: unchecked bytecodes and changes in environment. The first reason is that bytecodes can be produced from any source, including being directly constructed, being produced by a buggy compiler, or being produced by a malicious party. To guarantee type safety, the JVM must check the bytecodes. Similarly, the compiler reads and trusts the annotations on signatures of called methods; a type safety guarantee requires the entire program to be checked at load time. The second reason is that the environment — such as called methods — may change between the time that a class is compiled and the time that it is loaded/run, for instance if a library it calls is changed and recompiled. Again, the only way to guarantee type safety is to check the entire program at once, at load time.

A type-checking plug-in that enforces annotation (e.g., type qualifier) semantics at load time can be invoked from the `premain` method, and can use a third-party bytecode library for convenient analysis of the bytecodes of loaded classes. In Java, the `premain` method can be used to perform bytecode analysis (and instrumentation) just before a program's `main` method is invoked.

A plug-in developer can create a class that declares a `premain` method. The virtual machine invokes `premain`, passing an instance of `java.lang.instrument.Instrumentation` as an argument. This instance allows the developer to obtain an array of loaded classes, which the bytecode library can convert to its own representation. This representation can then be used to traverse the bytecodes of each class's methods and perform verification of a type qualifier's semantics — all before the virtual machine invokes the program's `main` method.

As with compile-time processing, a framework for writing load-time plug-ins is outside the scope of this proposal. After obtaining some experience with JVM plug-ins, we or others may propose support for writing them, perhaps in the spirit of [Fon04].

References

[CMM05] Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 85–95, Chicago, IL, USA, June 13–15, 2005.

[Fon04] Philip W. L. Fong. Pluggable verification modules: An extensible protection mechanism for the JVM. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 404–418, Vancouver, BC, Canada, October 26–28, 2004.