Bytecode Analysis for Checking Java Access Modifiers

Andreas Müller
Institute for System Software
Johannes Kepler University Linz
Linz, Austria
mueller@ssw.jku.at

ABSTRACT

When compiling a Java program, too restrictive access modifiers immediately lead to compilation errors. However, developers do not even get informed, if an access modifier could be more restrictive. We developed a tool called AccessModifierAnalyzer (AMA) that reports insufficiently restrictive Java access modifiers based on a static bytecode analysis. AMA analyses fields and methods and reports on visibility modifiers, the static modifier, and the final modifier. It can either be used as a stand-alone application, as part of an Apache Ant script, or as a NetBeans plug-in. In this paper, we motivate the need for a report on insufficiently restrictive access modifiers and give an overview of the architecture and usage of AMA.

1 Introduction

Java modifiers define the way in which types, methods, and fields are accessed in a Java virtual machine (JVM). They are defined in the Java VM specification [1]. Visibility modifiers for class members define whether another class is able to use a field or method. Members declared *public* can be accessed from everywhere, members declared *protected* only from subclasses and classes in the same package, and members declared *private* only from the class they are defined in. If no access modifier is specified, a member is said to have *default* access. It can only be accessed from classes in the same package. The *static* modifier defines that a field or method is per class and not per instance. The *final* modifier defines that a field cannot change its value despite a single initialization in the constructor.

An access modifier is more restrictive if it disallows more operations on the class member. For visibility modifiers, the sorting based on increased restrictiveness is as follows: *public*, *protected*, *default*, *private*. A field with a *final* modifier is restricted, because its value cannot be changed. A method with a *static* modifier is restricted, because it cannot access instance members of its class.

If a class member has an access modifier that is too restrictive, the Java compiler will not produce a class file and output an error message. In contrast, if a class member could be more restrictive, the compiler does not produce any warning. Therefore,

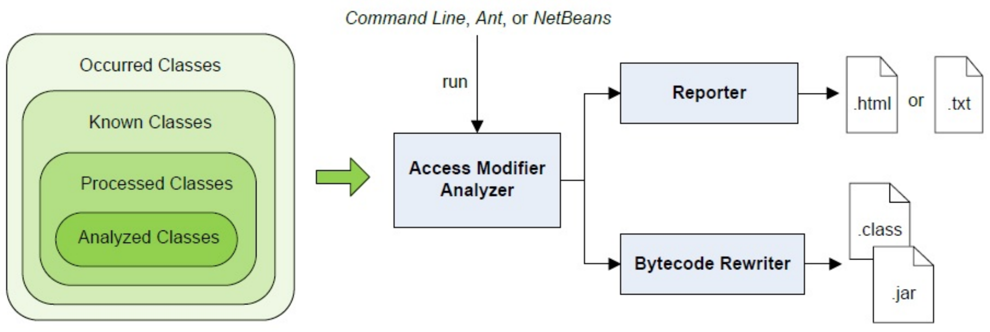

Figure 1: Overview of the AccessModifierAnalyzer (AMA) tool.

during development it is likely that access modifiers are declared less restrictive than possible. Declaring class members with more restrictive access modifiers has several advantages:

- **Code Readability:** If the access modifiers are accurate, they also provide most information to a programmer reading the code. For example, when looking at a field declaration, the developer will immediately see whether this field is modified in methods of the class or only in its constructors.
- **Runtime Performance:** The virtual machine can take advantage of the additional information restrictive modifiers provide. The compiler can make special assumptions about fields that are declared as *final*. Additionally, declaring a method *private* will result in a smaller virtual method table of the containing class. A call to such a method will no longer need a virtual method dispatch, but can be performed as a direct call.
- **Modularity:** Restrictive access modifiers help focusing on modularity. A well-known Java idiom states "create privately, publish later" [2]. The general idea behind this strategy is to increase the visibility of a class member only if it is really necessary and not because it might be necessary later.

In this paper, we present a tool that performs a global bytecode analysis for finding unrestrictive modifiers. We deal with the visibility modifiers for fields and methods, the *final* modifier for fields, and the *static* modifier for methods. The tool generates a report about class members, whose modifiers could be more restrictive. It is also able to perform a binary rewriting of the class files that changes the modifiers automatically.

2 Architecture

Figure 1 shows an overview of our tool. The AccessModifierAnalyzer (AMA) can be started from the command line, via an Apache Ant [3] task, or from within the NetBeans IDE [4]. It uses the two bytecode analysis frameworks BCEL [5] and ASM [6] for analysing and rewriting the bytecode.

During the analysis, it maintains a type universe and distinguishes between occurred classes, known classes, processed classes, and analysed classes. The Java bytecode for processed classes and analysed classes is available to the tool. The instructions of methods of those classes are processed and used to define the sufficient modifiers. The Java bytecode instructions relevant for deciding on the modifiers are the invoke instructions and the instructions for accessing fields (`getfield`, `getstatic`, `putfield`, and `putstatic`). Reports about improvable modifiers are only generated for analysed classes.

When the bytecode for a class is not available, but it is found on the classpath, then it is a known class. If a class is not found on the classpath, but nevertheless occurs in the bytecode of the processed classes, the tool makes conservative assumptions about it. In particular, if it is a super class of an analysed class, we preserve the modifiers of its methods, because we do not know whether a method overrides a method in the occurred class.

The report for all analysed classes is sent to a report generator that is capable of producing HTML or text output. Additionally, the AMA can automatically apply the proposed modifications and create class files with new modifiers for the members of analysed classes. We use this feature for verifying the correctness of the reports generated by AMA, i.e. this can be used to check that no access modifier is made more restrictive than allowed.

3 Usage

The AMA can either be started from the command line, as an Ant task, or from within the NetBeans IDE.

The `source` attribute defines the input directory in which the tool searches for Java class and jar files. Any Java class whose bytecode is found in this directory is processed. The two subelements `analyze` and `ignore` help to define the classes that are actually analysed and therefore included in the generated report. The `analyze` element includes all classes whose full name (including the package name) start with the specified identifier. The `ignore` element excludes classes from the analysis. If an output directory is specified as the `destination` attribute, then the binary rewriter is used to produce modified bytecode and write the new class and jar files. The `report` attribute can be used to produce either HTML or text output.

There is a custom fine-grained way to control the analysis based on annotations. Classes, methods and fields marked with an `@API` annotation are not analysed. The `@MinAccess` annotation specifies the minimum visibility level for a field or method. The AMA will not suggest a stricter visibility modifier in its report. The `@Mutable` annotation is only applicable for fields to prevent the suggestion of a *final* modifier. Similarly, the `@NotStatic` annotation specifies that a method should not be declared *static* even if this is possible.

The report includes detailed suggestions for fields and methods as well as statistics about the whole program and about packages. The tool reports on the following issues:

Visibility Modifiers: Our tool checks visibility modifiers of both fields and methods. The AMA inspects accesses on fields or methods and determines the strictest possible visibility modifier taking possible overrides into account. Furthermore, it is separately reported when a field or method is never accessed.

Final Modifier: The *final* modifier requires the analysis of methods and constructors of classes. If a field is only set once and only in a constructor, the field should be declared as *final*.

A method can possibly be declared as *static* if it does not access the `this` pointer, does not override another method and is not overridden itself. If these conditions hold and the method is not declared as *static*, the AMA reports this method.

Wrong Class: An experimental feature of our tool is to generate hints whether a method should be declared in a different class. For generating this suggestion, we analyse which class members are accessed by the method. If the method accesses significantly more class members of a different class than it was declared in, it is suggested to move it to a different class.

4 Future Work

In future work, we want to improve the tool in case of programs with extensive reflection usage. The bytecode analysis does not find calls to methods using the reflection API. One possible solution would be to intercept reflection calls and make methods accessible for the reflection API before calling them.

Another possible direction for future work is to find a better algorithm for determining whether a method should be declared in another class. Currently this feature is experimental and the generated reports are not accurate enough. We believe that the automatic analysis of this property of an object oriented program could be useful for enhancing its design.

5 Conclusions

We presented a tool that performs a bytecode analysis for finding insufficiently restrictive access modifiers. We believe that having restricted modifiers is beneficial for code readability and program modularity. Additionally, it can improve runtime performance of Java programs.

Acknowledgments

I would like to thank Thomas Wuerthinger for supporting and contributing ideas to this work.

References

[1] Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[2] *Create Privately Publish Later*, 2010. `http://c2.com/ppr/wiki/JavaIdioms/CreatePrivatelyPublishLater.html`.

[3] The Apache Software Foundation. *The Apache Ant Project*, 2010. `http://ant.apache.org`.

[4] Oracle Corporation. *NetBeans IDE*, 2010. `http://www.netbeans.org`.

[5] The Apache Software Foundation. *Byte Code Engineering Library*, 2010. `http://jakarta.apache.org/bcel`.

[6] OW2 Consortium. *ASM*, 2010. `http://asm.ow2.org`.