Checking Access to Protected Members in the Java Virtual Machine*

Alessandro Coglio

Kestrel Institute
3260 Hillview Avenue, Palo Alto, CA 94304, USA
Ph. +1-650-493-6871 Fax +1-650-424-1807
http://www.kestrel.edu/~coglio
coglio@kestrel.edu

1 Introduction

This paper illustrates how to check access to protected members in the JVM, which is not explained in [7] and, to the author's knowledge, has been completely neglected in the research literature. This aspect of enforcing type safety in the JVM is rather subtle and its correct implementation is not straightforward, as also evidenced by a bug in Sun's Java 2 SDK version 1.4 that causes some illegal programs to be accepted and some legal programs to be rejected (see [3] for concrete examples with source code).

Java compilers can easily check access to protected members and constructors[1] by looking them up in their declaring classes, which must be available in source or bytecode form. Checking access to protected members in the JVM is made difficult by dynamic class loading.

1.1 Notations

As in [6, 7], $\langle \mathsf{C}, \mathcal{L} \rangle$ denotes the class with name C and defining loader $\mathcal{L}$, while $\mathsf{C}^{\mathcal{L}}$ denotes the class with name C and initiating or defining loader $\mathcal{L}$. See [3] for a more detailed explanation.

A member reference $\mathsf{C.n:d}$ consists of a class name C, a member name n, and a member descriptor d. A field descriptor is (a textual representation of) a type; a method descriptor consists of (a textual representation of) zero or more argument types and a result type (possibly void).

The notation $c < c'$ means that c is a subclass of c'; the notation $c \leq c'$ denotes that c is c' or a subclass of c'.

*This paper is a short version of [3].

[1] In Java, constructors are not members. In the JVM, constructors are realized by instance initialization methods, i.e. methods with the special name <init>, which are members.

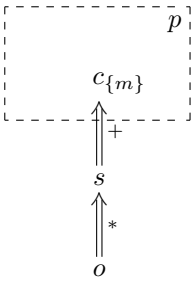

Figure 1: Restriction on protected access

2 Requirements on Protected Members

In the JVM, like in Java, every member has an associated access attribute: public, private, protected, or default. The attribute determines the circumstances in which the member can be accessed.

Let m be a member declared in a class c that belongs to a package p. If m is public, it can be accessed by (code in) any class. If m is private, it can be accessed only by c. If m has default access, it can be accessed only by any class that belongs to p.

If m is protected, things are slightly more complicated. First, m can be accessed by any class belonging to p, as if it had default access. In addition, it can be accessed by any subclass s of c that belongs to a package different from p, with the following restriction: if m is not static, then the class o of the object whose member is being accessed must be s or a subclass of s (if m is static, the restriction does not apply: m can be always accessed by s). The relationship among c, s, o, m, and p is depicted in Figure 1, where the double arrow labeled by $+$ denotes one or more direct superclasses and the double arrow labeled by $*$ denotes zero or more direct superclasses.

These access rules are described in [7, Sect. 5.4.4]. The restriction on protected access (i.e. that $o \leq s$ if $s < c$ and s does not belong to p) is described in the specification of the getfield, putfield, invokevirtual, and invokespecial instructions in [7, Chapt. 6].

These access rules do not include any requirement on the accessibility of c. One might expect, for instance, that if m is public then it can be accessed by a class x not belonging to p only if c is also public. However, c may not be the class directly referenced by x: x may reference a class c' that inherits m from c (i.e. m is found in c, starting the search from c'). It is sufficient that c' can be accessed by x (i.e. c' and x are in the same package or c' is public), regardless of whether c can be accessed by x or not.

The access rules for members in the JVM correspond to the access rules for members and constructors in Java [5, Sect. 6.6].

The restriction on protected access ensures that a protected member m declared in c, is accessible, outside the package p of c, only (on objects whose classes

are) inside the part of the class hierarchy rooted at the accessing (sub)class s [1, Sect. 3.2] [2]. In particular, s cannot access m in a part of the class hierarchy rooted at a (sub)class $s' \neq s$ in a different branch.

The restriction on protected access prevents almost arbitrary access to protected instance members of objects [9]. Suppose that m is a protected instance field declared in c. Without the restriction, any class x could read the content of the field m of any object of class c, using the following trick: define a subclass s of c (the trick works only if c is not final, hence the "almost" adverb above); declare a method in s that takes an object of class c as argument and returns the content of its m field; and have x call this method. The restriction on protected access prevents this situation, because s can access the field only if the class o of the object satisfies $o \leq s$.

Because of dynamic dispatch, the method actually invoked on an object may differ from the method that the reference resolves to. However, access control checks only apply to the resolved method, not the dynamically invoked one. While Java compilers check that an overriding method does not restrict the access attribute, the JVM does not check that: [7] does not explicitly require the check and it is easy to verify that Sun's Java 2 SDK version 1.4 does not perform the check.

3 Checking the Requirements

Most access rules are checked during resolution. For example, when a getfield is executed for the first time, the embedded field reference is resolved and the access attribute of the resulting field is checked against the class where getfield occurs: if access is disallowed, an exception is thrown; otherwise, getfield retrieves the field's content.

The restriction on protected access requires additional checking. Suppose that the getfield occurs in a class s, that the field reference resolves to a protected instance field declared in a superclass c of s, and that c and s belong to different packages. Whether the field access is allowed depends on the class o of the target object, which may be different each time the getfield is executed. The following subsections describe various strategies to check whether $o \leq s$. While [7] states the requirements on protected member access, it does not explain how they can be checked.

3.1 Run Time Checking

The class of an object is always available through a reference to the object, in order to support the semantics of the instanceof and checkcast instructions. In typical implementations of the JVM, the storage for an object includes a reference to the object's class.

The simplest strategy to check the restriction on protected access is to have getfield, putfield, invokevirtual, and invokespecial do it each time they are executed. This run time check is performed only if the referenced member is

protected and declared in a superclass of the current class (i.e. the one where the instruction occurs) that belongs to a different package.

The first edition of [7] describes, in Chapter 9, an optimization for JVM implementations: certain instructions are replaced by quick, internal pseudo-instructions the first time they are executed. For instance, the first time a getfield is executed, it is replaced by the getfield_quick pseudo-instruction: instead of a field reference, getfield_quick embeds implementation-dependent information (typically, an offset) that is determined when the field is resolved and that is used to access the target object's field more quickly than going through the (resolved) field reference.

A similar rewriting approach could be used in a JVM implementation to more efficiently check protected member access at run time. The first time a getfield is executed and the field reference is resolved, there are two cases: if the resulting field is not protected or is not declared in a superclass of the current class that belongs to a different package, getfield is replaced by getfield_quick; otherwise, it is replaced by getfield_quick_prot. This new pseudo-instruction embeds the same implementation-dependent information as getfield_quick; in addition, its execution includes the run time check that $o \leq s$, where o is the class of the target object and s is the current class. An analogous strategy can be introduced for putfield, invokevirtual, and invokespecial.

However, getfield_quick_prot would not be very quick. Even though the run time check is only performed for protected fields declared in superclasses belonging to different packages, the overhead could be significant in certain programs.

In addition, [7, Sect. 5.4.4] states explicitly that the restriction on protected access should be checked as part of bytecode verification. In general, type safety in the JVM could be completely ensured via run time checks, but execution would be painfully slow. The purpose of static checking in the JVM is to increase performance. The following strategies check the restriction on protected access statically.

3.2 Eager Resolution

When analyzing a getfield, putfield, invokevirtual, or invokespecial, the bytecode verifier could resolve the embedded member reference and, if needed, check the restriction on protected access using the statically inferred target type.

Suppose that the class under verification is $s = \langle \mathsf{S}, \mathcal{L} \rangle$, the embedded member reference is $\mathsf{C.n:d}$, and the inferred target type is D. If the member reference resolves to a protected member declared in a class $c' \geq \mathsf{C}^{\mathcal{L}}$ that belongs to a package different from s and such that $c' > s$, the bytecode verifier checks that $\mathsf{D}^{\mathcal{L}} \leq s$. If $\mathsf{D} \neq \mathsf{S}$, this subtype check can be performed by resolving D or, according to the proposal in [8, 4], by generating the subtype constraint $\mathsf{D}^{\mathcal{L}} < \mathsf{S}^{\mathcal{L}}$; if $\mathsf{D} = \mathsf{S}$, no resolution or constraint is necessary because $\mathsf{S}^{\mathcal{L}} = s$. The check that $\mathsf{D}^{\mathcal{L}} \leq s$ is in addition to the check that $\mathsf{D}^{\mathcal{L}} \leq \mathsf{C}^{\mathcal{L}}$, which is always needed, regardless of whether the member is protected and of where it is declared. Since at run time the class o of the target object always satisfies

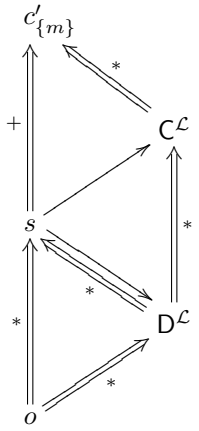

Figure 2: Relationship among all the involved classes

$o \leq \mathsf{D}^{\mathcal{L}}$, it is always the case that $o \leq s$ if $\mathsf{D}^{\mathcal{L}} \leq s$. The relationship among all these classes is depicted in Figure 2, where the single arrows denote resolution.

This static check is less precise than a run time check, because it might always be $o \leq s$ even if $\mathsf{D}^{\mathcal{L}} \leq s$ does not hold. However, less precision is the inevitable price to pay for better performance. In addition, the static check is as precise as the compile time checking in Java: the type statically inferred at compile time corresponds to the type statically inferred by the bytecode verifier. Besides soundness, the only requirement on bytecode verification is that code generated by Java compilers is accepted; if the restriction on protected access is satisfied in some Java code, it is also satisfied in the bytecode generated from that Java code.

The problem with this strategy is that *every* member reference from getfield, putfield, invokevirtual, and invokespecial is eagerly resolved in order to determine whether the member is protected and declared in a superclass of s that belongs to a different package. The additional check that $\mathsf{D}^{\mathcal{L}} \leq s$ is performed if and only if that is the case. Resolution may involve class loading and thus be a costly operation. Furthermore, eager resolution counters lazy loading.

As mentioned in Section 1, Java compilers check access to protected members and constructors by looking them up in their declaring classes. This effectively corresponds to the eager resolution strategy just described for the JVM. At compile time all classes are available and no dynamic loading is involved; thus, the strategy is adequate for compilation.

3.3 Limited Resolution

There are cases in which the restriction on protected access can be checked by the bytecode verifier without eagerly resolving the referenced member. In

addition, in certain cases member resolution is guaranteed to cause no loading. For the remaining cases, the member reference can be resolved as described in Section 3.2.

3.3.1 Current Class as Target Type

Resolving the class name S from a class $s = \langle \mathsf{S}, \mathcal{L} \rangle$ results in s itself because the loaded class cache associates s to S and $\mathcal{L}$ when s is created, i.e. $\mathsf{S}^{\mathcal{L}} = s$.

If the bytecode verifier infers the class name S as the target of a getfield, putfield, invokevirtual, or invokespecial that occurs in s, there is no need to resolve the embedded member reference because $s \leq s$, regardless of whether the member is protected and of where it is declared. In other words, resolution can be soundly avoided if the inferred target type coincides with the name of the class under verification.

3.3.2 Necessary Condition for Subtype Check

Consider a getfield, putfield, invokevirtual, or invokespecial occurring in a class s, with embedded member reference $\mathsf{C.n:d}$. Since the member resulting from resolution must have name n and descriptor d, a necessary condition for m to be protected and declared in a superclass of s that belongs to a different package is that *some* superclass of s, belonging to a different package, declares a protected field with name n and descriptor d.

This necessary condition can be checked by inspecting the superclasses of s, without resolving the member reference. Since the classes loaded into the JVM are closed under the superclass relation, inspecting the superclasses of s does not cause any loading. If no superclass of s declares a protected member with name n and descriptor d or every superclass that declares it belongs to the same package as s, it is impossible that the member resulting from resolution will be protected and declared in a superclass of s that belongs to a different package. Thus, the additional subtype check (i.e. that $\mathsf{D}^{\mathcal{L}} \leq s$) needs not be performed.

3.3.3 Resolution not Causing Loading

As mentioned before, resolving the class name S from a class $s = \langle \mathsf{S}, \mathcal{L} \rangle$ results in s itself without any loading taking place. In addition, resolving the direct superclass name R of s results in the direct superclass $\mathsf{R}^{\mathcal{L}}$ without any loading taking place, because when s is loaded $\mathsf{R}^{\mathcal{L}}$ is loaded too.

So, if the class name referenced by a getfield, putfield, invokevirtual, or invokespecial is S or R, the resolution of the member will cause no loading. This is a property that is satisfied because of the way the JVM is designed; no special action by the bytecode verifier is required.

3.4 Conditional Subtype Constraints

Instead of eagerly resolving a member reference and then checking the restriction on protected access if needed, the bytecode verified could generate a *conditional*

subtype constraint. The condition of the constraint expresses that the referenced member is protected and declared in a superclass of the current class that belongs to a different package.

For example, consider a member reference $\mathsf{C.n:d}$ embedded in a getfield, putfield, invokevirtual, or invokespecial that occurs in a class $s = \langle \mathsf{S}, \mathcal{L} \rangle$. Suppose that D is the inferred target type of that instruction, with $\mathsf{D} \neq \mathsf{S}$. The bytecode verifier generates the conditional subtype constraint

$$\textbf{if } \mathit{ProtCond}(\mathsf{C.n:d}^{\mathcal{L}}, \mathsf{S}^{\mathcal{L}}) \textbf{ then } \mathsf{D}^{\mathcal{L}} < \mathsf{S}^{\mathcal{L}}$$

The condition $\mathit{ProtCond}(\mathsf{C.n:d}^{\mathcal{L}}, \mathsf{S}^{\mathcal{L}})$ holds if and only if the member to which $\mathsf{C.n:d}$ resolves via $\mathcal{L}$ (i.e. the member with name n and descriptor d found searching from $\mathsf{C}^{\mathcal{L}}$) is protected and declared in a superclass of s that belongs to a different package.

Conditional subtype constraints are integrated with the unconditional (equality [6] and subtype [8, 4]) constraints in the JVM. The satisfaction of pending constraints is re-checked whenever classes are loaded and members are resolved. It is necessary to consider the transitive closure of all the constraints, conditional and unconditional.

3.5 Experimental Measures

The 3,915 classes (and interfaces) in the java and javax packages of Sun's Java 2 SDK version 1.4 require 68,621 static checks for protected members. That is the total number of triples $(s, \mathsf{C.n:d}, \mathsf{D})$ where s is a class in the java or javax package, $\mathsf{C.n:d}$ is a reference occurring in a getfield, putfield, invokevirtual, or invokespecial occurring in s, and D is one of the types inferred at that instruction by the bytecode verifier[2]. While other counts are possible, e.g. the total number of pairs (i, D) where i is an occurrence of getfield, putfield, invokevirtual, or invokespecial in some class in java or javax and D is one of the types inferred at i, what is important in the data below is the relative percentages, which are probably largely invariant to the exact count used.

Of the 68,621 checks:

- 30,598 (45%) can be checked as explained in Section 3.3.1 because the target class coincides with the current class;
- 31,571 (46%) do not cause any loading because the class referenced in the member reference is the current class or its direct superclass, as described in Section 3.3.3;
- *all* 68,621 can be checked as explained in Section 3.3.2 because the necessary condition for the restriction on protected access does not hold.

[2]While many bytecode verifiers, including the one in Sun's Java 2 SDK version 1.4, infer only one type for every instruction (by merging types from converging paths), the experimental measures reported here were taken with a bytecode verifier that infers multiple types, for increased precision.

Of course, there is overlap among these three sets of checks.

A reasonable implementation of the bytecode verifier could perform those checks as follows:

- 30,598 of the total 68,621 (45%) by recognizing that the target class coincides with the current class;
- 1,085 of the remaining 38,023 (1% of the total, 3% of the remaining) by recognizing that the referenced class is the current one or its direct superclass, so that resolution does not cause any loading;
- *all* the remaining 36,938 (54% of the total) by inspecting the superclasses and discovering that the necessary condition does not hold.

Thus, no eager resolution or conditional subtype constraint is necessary.

4 Conclusion

The best way to correctly and efficiently check protected member access seems to be the following: first, use the strategies described in Section 3.3; then, deal with the remaining checks by eagerly resolving member references or by generating conditional subtype constraints. If the classes in the java and javax packages of SDK are representative of typical Java programs in the usage of protected members, the experimental measures in Section 3.5 suggest that all or most checks can be performed without any eager resolution or conditional subtype constraints. Anyhow, there is a trade-off between eagerly resolving member references and generating conditional subtype constraints: the former is simpler and requires less machinery in the JVM, while the latter supports lazier class loading.

References

[1] Ken Arnold and James Gosling. *The Java™ Programming Language*. Addison-Wesley, second edition, 1998.

[2] Lawrence Brown. Protected access. In Data Structures Course Notes, available at http://www.apl.jhu.edu/Classes/Notes/LMBrown/resource/Protected-Access.pdf.

[3] Alessandro Coglio. Checking access to protected members in the Java virtual machine. Technical report, Kestrel Institute, February 2004.

[4] Alessandro Coglio and Allen Goldberg. Type safety in the JVM: Some problems in Java 2 SDK 1.2 and proposed solutions. *Concurrency and Computation: Practice and Experience*, 13(13):1153–1171, November 2001.

[5] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification*. Addison-Wesley, second edition, 2000.

[6] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java™ virtual machine. In *Proc. 13th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, volume 33, number 10 of *ACM SIGPLAN Notices*, pages 36–44, October 1998.

[7] Tim Lindholm and Frank Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[8] Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A formal specification of Java class loading. In *Proc. 15th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, volume 35, number 10 of *ACM SIGPLAN Notices*, pages 325–336, October 2000.

[9] Frank Yellin. Private communication, August 2002.