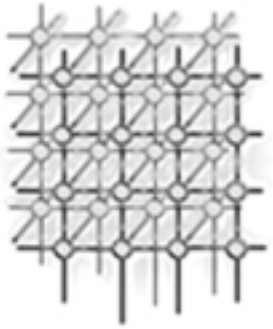
CONCURRENCY—PRACTICE AND EXPERIENCE
Concurrency: Pract. Exper. 2000; **00**:1–9 *Prepared using* cpeauth.cls *[Version: 2000/05/12 v2.0]*

Java Access Protection through Typing



Eva Rose[1] and Kristoffer Høgsbro Rose[2]

[1] *GIE Dyade, INRIA-Rocquencourt, Domaine de Voluceau, Rocquencourt B.P.105, 78153 Le Chesnay (France)*

[2] *IBM T. J. Watson Research Center, 30 Saw Mill River Road, Hawthorne, NY 10532 (USA)*

SUMMARY

We propose an integration of field access rights into the Java type system such that those access permission checks which are now performed dynamically (at run-time), can instead be done statically, *i.e.*, checked by the Java compiler and rechecked (at link-time) by the bytecode verifier.

We explain how this can be extended to remove all dynamic checks of field read access rights, completely eliminating the overhead of get methods for reading the value of a field. Improvements include using fast static lookup instead of dynamic dispatch for field access (without requiring a sophisticated inlining analysis), the space required by get methods is avoided, and denial-of-service attacks on field access is prevented.

We sketch a formalization of adding field access to the bytecode verifier which will make it possible to prove that the change is safe and backwards compatible.

Key words: Java, Java Virtual Machine, Java bytecode verification, read-only field access.

1. Introduction

Object-oriented programming languages in general, and Java in particular, do not distinguish between read- and write-access to fields. Instead the recommended way of only permitting read access to a field is to make the field private and write a get method that accesses the field and returns the stored value. A field with both a get and set method is conventionally called an "attribute" or "property" and it is generally considered good practice to use such instead of fields:

- A property hides the actual data representation of the field so the implementation can be changed without consequence for the property.
- In Java, fields cannot be overridden by a subclass.

*Correspondence to: Kristoffer H. Rose, IBM T. J. Watson Research Center, 30 Saw Mill River Road, Hawthorne, NY 10532 (USA); E-mail: krisrose@watson.ibm.com.

Received 9 October 2000
Revised 16 March 2001
Accepted 28 March 2001

- A get method may trigger an exception or other action that a simple field access cannot.
- In Java, fields cannot be declared `synchronized` thus thread-safe access is only possible by using a property.

On the other hand, if performance issues are critical then using fields may be preferable because the Java semantics of field access states that the actual field location accessed in an object can be determined statically (at compile-time), whereas the actual get method to invoke is determined dynamically (at run-time) [2, §15.11.1]. This has the following consequences:

- Using a get method is significantly slower (at run-time) than using a direct field access. (The traditional remedy for this is to declare get methods `final` which permits the compiler to "inline" its body, *i.e.*, insert the field access instruction directly at the invocation place. In Java this is frequently not feasible because Java employs dynamic class loading so the class containing the get method may not have been loaded yet.)
- It is possible to access the field belonging to a particular (super)class of an object by simply casting the object of the field access to the appropriate class. One cannot obtain a similar effect with a get method. (One may see this as a feature rather than an inconvenience.)
- "Denial-of-service" attacks (or accidents) are possible in that a get method can be overriden by a subclass. (This can also be avoided by declaring the method `final`.)
- Finally, get methods may add a significant space overhead to class files since they must be declared and their code given. For example, get methods account for about one fourth of the total number of methods in the standard Java "`java.*`" package source classes. [†]

Furthermore, the Java virtual machine, JVM [3], specifies that field access control is performed through (dynamic) load and run time checks. This seems a shame since everything else about fields is static.

Here is a traditional example with a get method: an object that simply contains an integer value that should be publicly readable.

Example 1.1 (a traditional get method).

```
public class ReadOnly1 {
  private int it;
  public int getIt() {return it;}
}
```

Access to the `it` field is then done with code such as the following within some other class:

```
  ReadOnly1 cc;
  ...cc.getIt()...
```

with the problems discussed above.

In this paper we propose a simple modification in two steps that eliminates the problem:

[†]This measure was obtained for Sun's JDK 1.1 [6] with the unix command "`find jdk1.1 -name '*.java' -exec grep ' +public .*(' '{}' ';' | wc -l`" to get the total number of public methods (4317), and "`find jdk1.1 -name '*.java' -exec egrep ' +public .* get.*(' '{}' ';' | wc -l`" to get the number of get methods (999).

Table I. Java Access Modifiers.

Accessibility from	Modifier: `private`	"package"	`protected`	`public`
same class	✓	✓	✓	✓
other class, same package	×	✓	✓	✓
subclass outside package	×	×	✓	✓
other class outside package	×	×	×	✓

1. add a special get-specific access modifier that permits making the reading of a field "more public" than the modification of it, and
2. integrate field access checks into the type system.

In effect we propose replacing the class declaration above with the following:

Example 1.2 (a modified get method).

```
public class ReadOnly2 {
  private read public int it;
}
```

which explicitly permits everyone to read the field value with the usual field access syntax:

```
  ReadOnly2 cc;
  ...cc.it...
```

but does not permit assigning to the field outside of the `ReadOnly2` class.

Overview. In Section 2 we formalize field access rights of Java and explain how these could be integrated into the type system implemented by the Java Virtual Machine "bytecode verifier". Section 3 then explains how field access types can be exploited to achieve overhead-free read access to fields. Finally, we conclude in section 4 with some remarks on future work.

2. Field Accessibility Types

In this section we summarize the Java access modifiers [2, §6.6] and formalize accessibility for the field access instructions of the Java Virtual Machine.

The access granted by each modifier is shown in Table I. The first column indicates the four possible field access situations: Either the field is referenced from the *same class* where it is defined, from another class within the *same package* where it is defined, from a *subclass* of the class where it is

defined but outside its package, or, finally, from a class outside its package which is *not a subclass* of the class in which the field is defined. The columns are the Java access modifiers: `private`, `protected`, `public`, and the default case where no keyword is given, denoted "package" protection. For each combination the table indicates with "✓" that a field declared with the given access modifier is accessible from the given context, and with "×" that is it not.

The access rules of Table I are progressive in the sense that each row in the table extends the access permissions of the row above it. We formalize this as an ordering "$<$" on the set of access modifiers, with the meaning that an access modifier is less permissive than another if it permits less access to a field in the four access situations.

Definition 2.1 (accessibility ordering). An *access modifier*, denoted a, is a member of the following totally ordered set:

$$\texttt{private} < \text{“package”} < \texttt{protected} < \texttt{public}$$

In practice accessibility is checked at run-time by the JVM, specifically field accessibility is checked before the field value is accessed when the field reference is "resolved" [3, §5.4.4]. A field reference occurs in two instructions, putfield or getfield, depending on whether it was compiled from a the Java field reference that was a *LeftHandSide* of an assignment [2, §15.26] or not. We shall use getfield in our examples below but everything is equivalent for putfield.

Consider the Java code snippet

$$\texttt{class}\ c_1\ \{\ldots\ ((c_2)e).f\ \ldots\}$$

(where we have explicitly indicated the static type c_2 of the expression e). The field f must be declared by code such as

$$\texttt{class}\ c_3\ \{\ldots\ a\ t\ f;\ \ldots\}$$

with a the access modifier defining from where access to f is permitted, t the field type, and c_3 a superclass of c_2.

The c_1 class compiles to a JVM sequence of bytecode instructions that we may represent symbolically as

$$\text{class}\ c_1\ \{\ldots\ \text{getfield}(c_3, t, f)\ \ldots\}$$

(concretely the "getfield instruction is a single bytecode followed by an index into the JVM class file "constant pool" pointing to `CONSTANT_Fieldref_info` structure which again points to similar structures for a, t, and c_3 [3, §4.4.2]).

Checking that the getfield access is permitted requires the resolver to look up the real field declaration of f in the c_3 class merely because this is the only way to obtain a to use the rules of Table I to determine if f is accessible from c_1.

Our idea for the JVM is the following: if access information – a in our example above – is integrated into the getfield instruction, then

1. the bytecode verifier can check for access violations assuming that the provided access information (in the type) is correct, and
2. resolution can be reduced to merely checking that the type assumptions made at verification time were correct.

$$\frac{c_1 = c_3}{c_1 \vdash c_2 : c_3.a} \quad \text{(SameClass)}$$

$$\frac{\text{same-package}(c_1, c_3) \quad \text{“package”} \leq a}{c_1 \vdash c_2 : c_3.a} \quad \text{(SamePackage)}$$

$$\frac{c_2 \texttt{ extends}^* c_1 \quad c_1 \texttt{ extends}^* c_3 \quad \texttt{protected} \leq a}{c_1 \vdash c_2 : c_3.a} \quad \text{(Subclass)}$$

$$\frac{a = \texttt{public}}{c_1 \vdash c_2 : c_3.a} \quad \text{(Other)}$$

Figure 1. Access Modifier Type Rules.

Using the access modifier order of Definition 2.1 we can express the access rules used by JVM [3, §4.8.2, §5.4.4] as logical rules.

Definition 2.2 (field accessibility). Figure 1 gives the rules for the judgment

$$c_1 \vdash c_2 : c_3.a$$

which expresses that within the class c_1 an object of type c_2 has access to a field declared in c_3 with access modifier a. The rules use the following auxiliary tests: (1) $c' \texttt{ extends}^* c$ is true when c' is the same as or a subclass of c, and (2) same-package(c, c') is true when the two classes are in the same package (this amounts to checking that the fully qualified class names are identical up to the last “`.`”).

Rule (SameClass) states that a field is accessible from the class it is defined in. Rule (SamePackage) states that a field with a non-`private` access modifier can be accessed from the same package.

The (Subclass) rule is the only slightly complex rule. It states that a field with `protected` (or `public`) access can be accessed from a class which satisfies two conditions:

- it is a subclass of the class the field is declared in, and
- it is a superclass of the actual class of the object containing the field.

Finally, the (Other) rule states that only `public` fields are accessible from everywhere.

Our idea can now be formalized.

Theorem 2.3 (Static Field Accessibility Check). *The bytecode verifier can prove field accessibility provided the* getfield *instruction is augmented with the access modifier.*

Proof. If the access modifier is included in the getfield instruction then field accessibility can be approximated at verification time: in the judgment

$$c_1 \vdash c_2 : c_3.a$$

we have c_1, c_3, and a, in the bytecode thus available at verification. These suffice to decide the rules (SameClass), (SamePackage), and (Other) since only rule (Subclass) refers to the run-time type c_2 of

the object containing the field. However, *type safety* [1, 4] guarantees that c_2 is approximated by the verifier with a c_2' type that satisfies c_2 `extends`* c_2'. From this and c_2' `extends`* c_1, which we can check, follows that c_2 `extends`* c_1 thus we are able to check at verification time that the field access will always be safe. □

All that is needed to make this work is two things:

- Encode the access modifier *a* into the JVM *FieldDescriptor* encoding [3, §4.3.2]. Such an encoding is not difficult but beyond the scope of this paper.
- Replace the run-time field accessibility done at resolution with a check that the access modifier stored in the field access instruction is the same as (or stricter than) the one actually present in the field declaration.

3. Read-only Field Access

In Example 1.2, we proposed that a shorter syntax be given to specify explicitly permissions which are valid for a field used to read a value as is the purpose of get-methods. In this section we formalize this by proposing an extension to the official Java field declaration syntax [2, §8.3.1]. The modification consists in adding the `read` keyword to introduce the "read-only access modifier".

Definition 3.1 (extended field declaration). Replace the Java definition of *FieldDeclaration* [2, §8.3] with the one shown in Figure 2; additions are indicated in bold.

FieldDeclaration:
  FieldModifiers$_{op}$ ***ReadModifier*$_{op}$** *Type VariableDeclarators* `;`

***ReadModifier**:*
  `read` ***AccessModifier*$_{op}$**

AccessModifier:
  one of `public protected private`

Figure 2. Extended Field Declaration with Read-only Access Modifiers.

The semantics of the new construction is that it separates out the access modifiers which are specific for reading the value of a field from those which are specific for assigning a value to a field.

In example 1.1, the integer field `it` was declared as a `private` field. So the value of that field can only be read indirectly using the `public getIt` method. In example 1.2 we replaced this with the proposed syntax. We specify two access modifiers: the "write access modifier" *preceding* the `read` keyword and the "read access modifier" *following* the `read` keyword. We set the write access modifier to `private` to restrict assignments to `it` to the same class, as before, but we relax the "read access modifier" to `public` to obtain the same availability of the field value of `it` as was previously provided through the `getIt` access method.

The read access modifier may be omitted in which case it defaults to the write access modifier thus ensuring backwards compatibility in that "old style" declarations with no `read` keyword will set both the read and write access modifier to the specified one. Other modifiers such as `static`, `volatile`, *etc*, are not affected, but they may be intermingled with the write access modifiers for backwards compatibility.

The semantics of the read and write access modifiers is specified by enhancing the definition of *accessibility* [2, §6.6] to distinguish between two cases for field accessibility.

Definition 3.2 (read/write field access separation). If a field (access) occurs as a *LeftHandSide* [2, §15.26] then use the write access declaration, otherwise use the read access declaration.

Definition 3.2 captures the way a Java compiler determines whether a field access should be compiled into a getfield or putfield instruction. The formal semantics of read-only access for the JVM follows:

Definition 3.3 (getfield read access check). Consider a getfield in the context

$$\mathsf{class}\ c_1\ \{\ldots\ \mathsf{getfield}(c_2,\ a_w\ \texttt{read}\ a_r\ t\ ,\ f)\ \ldots\}$$

with c_1 a class names, t a type, f a field name, a_w and a_r the write and read access modifiers, respectively, and c_2 a subclass of the class c_3 with the declaration

$$\mathsf{class}\ c_3\ \{\ldots\ a_w\ \texttt{read}\ a_r\ t\ f\texttt{;}\ \ldots\}$$

This is *permitted* if

$$c_1 \vdash c_2 : c_3.a_r$$

(as defined in Definition 2.2).

Again there should be a similar rule for putfield using a_w instead of a_r in the judgment.

Remark 3.4. Finally remark that we can emulate the effect of the declaration

```
class c {
  w read r t f;
}
```

(with c a class name, w and r field modifiers, t a field type, and f a field name) by

```
class c {
  w t f;
  final r t get_c_f() {return f; }
}
```

where we must then replace all read accesses of the form $o.f$ (for some object o with a static type of c) with `o.get_c_f()`. The inconvenience is that the static type is explicitly present in the get method name, however, if this is acceptable then the argumentation above can be applied to including `final` in the static type information instead of the read/write accessibility.

4. Conclusion

We have outlined how access rights to fields, and specifically *read-only* access rights, can be encoded in the Java type system as implemented by a (slightly modified) Java bytecode verifier, thus eliminating all access right checks at run-time.

One may comment that "static is bad because everything should be run-time configurable." This possibility remains (using set and get methods) but we believe it is important to give the programmer of a class the choice of permitting (efficient) build-in static field access even for read-only fields, specifically for the variants of Java targeted at devices with limited resources [7].

Our first priority will be to combine the above with a formal model of a JVM, specifically with *lightweight* bytecode verification [5], as used by Sun's KVM [8]. This will give proofs of the consistency of the approach as well as help getting static access control even in sparse resources.

One important issue that should be investigated is whether read/write access types interferes with other type system features, notably generic types.

A different but very interesting further venue of research is that using "access types" could be used to implement "sticky" access rights such as "private objects" where the *value* cannot be passed out of the current method, for example.

Another question that one could ask is "why not for set methods?" This can be done but is complicated by the fact that set methods usually also check the value to be stored for validity to ensure that the object is (internally) consistent. One could introduce special "validity checks" such that our get example could be extended, for example, with a

```
public class ReadOnly throws IllegalIt {
  protected read public int it {if (it<0) throw IllegalIt;}
}
```

with the semantics that any assignment to `it` would execute the additional "assertion" code. Such an addition may be worth considering, however, in contrast to the read-only case it complicates the Java language considerably.

ACKNOWLEDGEMENTS

The authors are grateful for funding provided for this project by the "Incitative JavaCard" of INRIA and the Plume group of LIP, ENS-Lyon.

REFERENCES

1. S. Drossopoulou and S. Eisenbach. Java is type safe – probably. In *European Conference of Object Oriented Programming*, Lecture Notes in Computer Science. Springer-Verlag, June 1997.
2. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. The Java Series. Addison-Wesley, second edition, 2000.
3. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, second edition, 1999.
4. T. Nipkow and D. von Oheimb. Java_{light} is type-safe – definitely. In Luca Cardelli, editor, *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 161–170, San Diego, California, January 1998. ACM Press.
5. Eva Rose and Kristoffer Høgsbro Rose. Lightweight java bytecode verification. Technical report, INRIA. To appear.

6. Sun. *JDK 1.1 Documentation*, 1997. Available from http://java.sun.com/products/jdk/1.1/docs/.
7. Sun. Java 2 platform, micro edition. `http://java.sun.com/j2me`, 1999.
8. Sun. The K virtual machine (KVM). `http://java.sun.com/products/kvm`, 1999.