

Ambiguities in Java

JIEN-TSAI CHAN and WUU YANG

National Chiao Tung University, Department of Computer and Information Science,

Hsinchu, Taiwan, 300, R.O.C.

(email: {maxchan, wuuyang}@cis.nctu.edu.tw)

ABSTRACT

Though the Java programming language was designed with extreme care, there are still a few ambiguities left in the language. The ambiguities are those issues that are not defined clearly in the Java language specification. The different results produced by different compilers on several example programs support our observations.

KEY WORDS: Java; language design; language definition; object-oriented programming;

1. INTRODUCTION

Since its first public release in 1994, the Java programming language [1] has rapidly become very popular. Java is a derivative of C++ with some features borrowed from other programming languages, e.g. Objective C, Eiffel, Smalltalk, Cedar/Mesa, and Lisp [2]. Therefore, many programmers find it familiar and easy to learn. Although Java has overcome

or avoided most problems of these languages, new ambiguities appear in Java.

In this paper, we discuss Java from the perspective of a programming language. Several ambiguity problems of Java are reported. Other features of Java, such as security, exceptions, concurrency, and libraries are not considered here. Daconta et al.'s book [3] discusses other features that are not included in this paper. Thimbleby [4] discussed the pros and cons of various features of Java. The issues contained in his paper include the notation traps, the improvement over C and C++, the paradigm of Java, object orientation in Java, and a few other features. The problems we discussed are quite different from his paper. Alexander et al. [5] also discussed a few related problems in Java. The problem of the *protected* modifier was discussed in their paper too. However, we provide a practical example to show the importance of the

problem. There is a similar research that is dedicated to study the problems in Pascal [6].

The ambiguities are caused by the unclear specification of the Java language [1]. Some examples are provided for explanation and are compiled with several available compilers. The compilers we tested include Sun JDK 1.0.2, 1.1.8, 1.2.2, 1.3.0 [7], IBM JDK 1.2.2, 1.3.0 [8], and IBM VisualAge for Java 4.0 [9]. The different results produced by the different compilers demonstrate that even these serious compiler writers do not agree with one another on the ambiguities.

We found two ambiguities in Java. The first ambiguity is caused by mixing the inheritance relationship and the package relationship on default-access members. The Java specification does not define the priority of the two relationships for default-access members. This problem was mentioned in our previous work [10, 11]. This paper gives a detailed discussion. The second ambiguity occurs when the compiler attempts to locate the invoked method by performing a widening conversion on the actual parameters. This ambiguity problem is caused

by the flawed rules in the Java specification for choosing the *most specific* method. This paper is organized as follows. Section 2 introduces the ambiguity problems. Section 3 is our conclusion.

2. THE AMBIGUITY OF THE ACCESSIBILITIES OF MEMBERS

Java defines three access modifiers—*public*, *protected*, and *private*—to control the accessibility of the members of a type (a type could be a class or an interface). These access modifiers are similar to those of the C++ language but there are some differences. In Java, a member declared with the *public* modifier means that any code could access this member. If a member is declared *protected* in the class C, only the code within the same package that C belongs to and the descendant classes of C can access that member. A *private* member of the class C is accessible only within the body of C. A member without any access modifiers has the *default access* (it is also called the *friendly access*). If a member of class C is declared with the default access, the member

```

//----- in file A.java -----
package pkg1;
public class A {
    int x=1;
}
//----- in file B.java -----
package pkg2;
public class B extends pkg1.A {
    int getX() { return x; }    // error
}
//----- in file C.java -----
package pkg1;
public class C extends pkg2.B {
    // correct or error, depend on compiler
    int getX() { return x; }
}

```

Figure 1. Example 1 for the accessibilities of members.

can be accessed only by the code within the same package that the class C belongs to.

Consider the example in Figure 1. Assume the three classes A, B, and C are defined in three different compilation units (that is, source files). Both class A and class C belong to the package `pkg1`. Class B is defined in the package `pkg2`. Furthermore, class C inherits class B and B inherits A.

According to the specification of the Java language [1], a member declared without access modifiers can be accessed within the package in which the type is declared. The field `x` is declared without any access modifier in class A. Because class A and class B are defined in different packages, the field `x` of class A is not

accessible in class B. Therefore, the method `getX` in class B is in error.

The validity of the method `getX` of the class C needs further investigation. According to the inheritance relationship of the classes, the field `x` is inaccessible from class C. However, because class C and class A are in the same package `pkg1`, the field `x` is accessible in class C according to the relationship of the packages. It is quite counter-intuitive because the field `x` of class A is not accessible in class B but it is accessible in a subclass (that is, class C) of B.

We compile the program with several available Java Development Kits (JDKs). The results of compilation are summarized in Table 1. Some compilers (Sun JDK 1.0.2, 1.1.8 and IBM VisualAge for Java 4.0) allow class C to

access the field `x`, but others (JDK 1.2.2 and JDK 1.3.0 of Sun and IBM) do not. Different compilers even issue different error or warning messages. Sun JDK 1.2.2 and IBM JDK 1.2.2 say that the field `x` is not defined in `C`. Sun JDK 1.3.0 and IBM JDK 1.3.0 say that the field `x` is not public. The field `x` cannot be accessed from other packages.

The accessibility of a member `m` in a class `C` is determined by the access modifier of `m`, the package that `C` belongs to, and the inheritance relationship of `C` together. However, the package relationship and inheritance relationship sometime tell us different stories. The ambiguity problem exists because the Java language specification [1] does not define the above situation. The same problem does not happen in C++ because the accessibilities of

members are transmitted only along the inheritance relationship.

Two related problems also originate from the crude mixture of the package relationship and the inheritance relationship. Consider the example in Figure 2. Similar to the previous example, class `A` and class `C` belong to package `pkg1`. Class `B` belongs to package `pkg2`. Class `C` inherits class `B` and class `B` inherits class `A`. In class `A`, method `m1` is declared as an abstract method. According to the Java language specification [1], a class must implement all the inherited abstract methods unless that class is declared as an abstract class. Class `B` cannot see method `m1` because `B` is in a different package. Class `C` can see `m1` according to the package relationship or cannot see `m1` according to the inheritance relationship.

Compiler	Result
Sun JDK 1.0.2	passed
Sun JDK 1.1.8	passed
Sun JDK 1.2.2	error: Undefined variable: <code>x</code>
Sun JDK 1.3.0	error: <code>x</code> is not public in <code>pkg1.A</code> ; cannot be accessed from outside package
IBM JDK 1.2.2	error: Undefined variable: <code>x</code>
IBM JDK 1.3.0	error: <code>x</code> is not public in <code>pkg1.A</code> ; cannot be accessed from outside package
IBM VisualAge for Java 4.0	passed

Table 1. The compilation results of the program in Figure 1.

```

//----- in file A.java -----
package pkg1;
public abstract class A {
    abstract int m1();
}
//----- in file B.java -----
package pkg2;
public class B extends pkg1.A {
    // shall we implement m1 here?
}
//----- in file C.java -----
package pkg1;
public class C extends pkg2.B {
    // shall we implement m1 here?
}

```

Figure 2. Example 2 for the accessibilities of members.

The ambiguity occurs in class C. However, most compilers have troubles for class B. To implement an invisible abstract method in class B is questionable for these compilers. All compilers complain that class B should be declared as abstract when method m1 is not defined in B. When method m1 is defined in class B, the compilation results of the program are shown in the second column of Table 2.

Sun JDK 1.0.2 compiles the program successfully. Sun JDK 1.1.8 gives a warning

message saying that method m1 of class B does not override the corresponding method of class A. IBM and Sun JDK 1.2.2 issue a warning message similar to the one issued by Sun JDK 1.1.8 and an error saying that class B should be declared as abstract because it does not implement the inherited abstract method m1. IBM and Sun JDK 1.3.0 and IBM VisualAge for Java 4.0 give an error with similar complaints as JDK 1.2.2. In our opinion, it is not necessary to implement an invisible abstract method in class B. The compilers should compile class B

Compiler	m1 is defined in class B	m1 is defined in class C
Sun JDK 1.0.2	passed	passed
Sun JDK 1.1.8	1 warning	passed
Sun JDK 1.2.2	1 error, 1 warning	1 error
Sun JDK 1.3.0	1 error	1 error
IBM JDK 1.2.2	1 error, 1 warning	1 error
IBM JDK 1.3.0	1 error	1 error
IBM VisualAge for Java 4.0	1 error	passed

Table 2. The compilation results of the program in Figure 2.

```

//----- in file A.java -----
package pkg1;
public abstract class A {
    static int m2()    { return 1; }
    int m3()          { return 2; }
}
//----- in file B.java -----
package pkg2;
public class B extends pkg1.A {
    int m2()          { return 3; }
    static int m3() {return 4; }
}
//----- in file C.java -----
package pkg1;
public class C extends pkg2.B {
    int m2()          { return 5; }
    static int m3() {return 6; }
}

```

Figure 3. Example 3 for the accessibilities of members.

successfully no matter whether the definition of `m1` exists or not.

To show the ambiguity in class C, class B is redefined as an abstract class with no method definitions. Similarly, all compilers issue error messages if method `m1` is not defined in class C. When method `m1` is defined in class C, the third column of Table 2 shows the compilation result of class C. JDK 1.2.2 and JDK 1.3.0 of IBM and Sun still complain that class C should be declared as an abstract class even though `m1` has already been defined in C. The definition of `m1` in C seems useless. Other compilers compile class C successfully. The specification of the Java language is not defined clearly about this case.

Another problem is concerned with method overriding between static methods and instance methods. Consider the example in Figure 3. Class A and class C belong to package `pkg1`. Class B belongs to package `pkg2`. Class C inherits class B and class B inherits class A. In class A, `m2` is a static method and `m3` is an instance method. In class B and class C, a new instance method `m2` and a new static method `m3` are declared.

According to the specification of the Java language [1], an inherited static method cannot be overridden by an instance method of the same name in a subclass, and vice versa. However, `m2` and `m3` of A are inaccessible in B. In our opinion, the new definitions of `m2` and `m3`

in B do not violate the rule. Java compilers should successfully compile class B.

The compilation results of different compilers are shown in the second column of Table 3. Sun JDK 1.0.2 gives two errors saying that the definitions of m2 and m3 in class B violate the rule. Sun JDK 1.1.8 and JDK 1.2.2 of IBM and Sun give two warning messages saying that m2 and m3 of class B do not override the corresponding methods of class A. JDK 1.3.0 of IBM and Sun and IBM VisualAge for Java 4.0 finish the compilation successfully.

To show the problems in class C clearly, the definitions of m2 and m3 are removed from class B. The compilation results are shown in the third column of Table 3. JDK 1.2.2 and JDK 1.3.0 of IBM and Sun compile class C successfully. Other compilers issue error messages saying that the definitions of m2 and

m3 in class C violate the rule.

Not only different compilers produce different results on the same example, but it is surprising that the same compiler adopts different attitudes toward different examples. For instance, in the first and third examples, Sun JDK 1.3.0 considers that the default access members of class A are inaccessible in class C. However, in the second example, the abstract method of class A is visible in class C. Class C is asked to be abstract. The inconsistent behavior of a compiler really confuses Java programmers. Java programmers will be further confused by the inconsistent behavior among different compilers.

3. THE AMBIGUITY OF METHOD INVOCATIONS

Java allows both method overloading and method overriding. The work of determining the

Compiler	class B	class C
Sun JDK 1.0.2	2 errors	2 errors
Sun JDK 1.1.8	2 warnings	2 errors
Sun JDK 1.2.2	2 warnings	passed
Sun JDK 1.3.0	passed	passed
IBM JDK 1.2.2	2 warnings	passed
IBM JDK 1.3.0	passed	passed
IBM VisualAge for Java 4.0	passed	2 errors

Table 3. The compilation results of the program in Figure 3.

actual method to be invoked is partitioned into two parts. The first part is done by the compiler (at compile time). The second part is done by the run-time system (at run time). When encountering a method call, such as `A.m(...)`, the compiler searches the declared class of variable `A` for a method called `m` that are visible in the current context. If there are one or more such methods, the compiler chooses the *most specific* signature based primarily on the declared types of the parameters in `A.m(...)` (the exact rules are quite complicated, see Section 12.15 of [1]). When the method call `A.m(...)` is about to be executed, the run-time system examines the actual class of the object referenced by the variable `A`. The method whose name is `m` and whose signature is exactly the most specific signature determined by the compiler is invoked.

Several rules are used in the Java language specification to select a method (see Section 12.15 of [1]). However, the rules are ambiguous in a situation (related to the widening conversion). Different compilers produce different results in this situation.

In Java, there are two kinds of widening conversions (i.e. coercion). A widening primitive conversion is a conversion from a primitive type to another primitive type and no information about the overall magnitude of the numeric value is lost. A widening reference conversion is a conversion between non-primitive types and the conversion can be proved correct at compile time (e.g. from a subclass to its superclass). Method invocation allows the use of both widening conversions. The widening conversions are performed automatically and implicitly.

The implicit widening conversions bring about not only conveniences but also potential hazards. The problem comes from the mixture of widening conversion, method overloading, and class inheritance. Consider the example in Figure 4.

Class `D` extends class `C` and `C` extends class `B`. Class `B` extends class `A`. Class `M` extends class `N`. The methods named `m` and `p` use parameters of primitive types and reference types, respectively. The results of compiling the example with different compilers are shown in

```

class A {}
class B extends A {}
class C extends B {}
class D extends C {}
class N {
    public void m(int x) { System.out.println("N.m(int x)"); }
    public void m(short x) { System.out.println("N.m(short x)"); }
    public void p(B x) { System.out.println("N.p(B x)"); }
    public void p(C x) { System.out.println("N.p(C x)"); }
}
public class M extends N {
    public void m(long x) { System.out.println("M.m(long x)"); }
    public void p(A x) { System.out.println("M.p(A x)"); }
    public static void main(String[] args) {
        M x = new M();
        D d = new D();
        byte a = 1;
        x.m(a); // problem here
        x.p(d); // problem here
    }
}

```

Figure 4. Example 4 for the ambiguity of method invocations.

Table 4. Sun JDK 1.0.2, 1.1.8, 1.3.0 and IBM JDK 1.3.0 issue error messages saying that both methods `m` and `p` are referenced ambiguously. These results comply with the Java specification.

IBM and Sun JDK 1.2.2 give the similar error message but the target methods are different. IBM VisualAge for Java 4.0 compiles the program without complaints. These

compilers do not comply with the Java specification.

The ambiguity of method invocation happens when there are both an applicable local instance method and a more appropriate inherited instance method. Furthermore, the methods must be invoked with the widening conversion of parameters. According to the Java specification, the two methods are both the most

Compiler	Ambiguous methods for widening primitive conversion	Ambiguous methods for widening reference conversion
Sun JDK 1.0.2	M.m(long) and N.m(short)	M.p(A) and N.p(C)
Sun JDK 1.1.8	M.m(long) and N.m(short)	M.p(A) and N.p(C)
Sun JDK 1.2.2	M.m(long) and N.m(int)	M.p(A) and N.p(B)
Sun JDK 1.3.0	M.m(long) and N.m(short)	M.p(A) and N.p(C)
IBM JDK 1.2.2	M.m(long) and N.m(int)	M.p(A) and N.p(B)
IBM JDK 1.3.0	M.m(long) and N.m(short)	M.p(A) and N.p(C)
IBM VAJ 4.0	passed	passed

Table 4. The compilation results of the example in Figure 4.

specific methods. Therefore, the compilers have to issue error messages about the ambiguity. However, in the above example, `N.m(short)` and `N.p(C)` are the most specific method intuitively. Furthermore, if either or both of the applicable local instance method and the more appropriate inherited instance method is (are) changed to be a static method, the ambiguity disappears magically. An interesting thing is that both JDKs version 1.2.2 of Sun and IBM even select the wrong one from the inherited methods.

Because the ambiguity happens when the method is invoked with the widening conversion of parameters, it is not easy to be detected by the designer of a library. However, it will surprise the users of the library. The solution is to use the casting conversions for the parameters of the method to specify the desired method.

4. CONCLUSION

Although Java is a general-purpose programming language, one of its major strengths is for writing Internet applications. Therefore, security, robustness, and stability are

very important for Java. A Java program should have consistent behavior, no matter which compiler is used to compile the program and no matter when and where the program runs. Ambiguity problems are the major barriers to achieve the above goals. When designing a programming language, it is very hard to find and remove ambiguity problems completely. Instead of a literal description, a formal framework is a better way to define a programming language. Formal frameworks are helpful to check ambiguities systematically. However, most formal frameworks, like the attribute grammar, are not practical for a real programming language. Designing a formal framework that is simple to use and easy to understand is still a significant challenge.

ACKNOWLEDGEMENT

This work was supported in part by National Science Council, Taiwan, R.O.C. under grant NSC 89-2213-E-009-146 and NSC 90-2213-E-009-142.

REFERENCES

1. Gosling J, Joy B, Steele G, Bracha G. *The Java Language Specification* (2nded). Addison-Wesley: MA, 2000.

2. Gosling J, McGilton H. The Java language environment white paper.
<http://java.sun.com/docs/white/langenv/>
[May 1996].
3. Daconta MC, Monk E, Keller JP, Bohnenberger K. *Java Pitfalls : Time-Saving Solutions and Workarounds to Improve Programs*. John Wiley & Sons: New York, 2000.
4. Thimbleby H. A critique of Java. *Software - Practice and Experience* 1999; **29**(5):457-478.
5. Alexander RT, Bieman JM, Viega J. Coping with Java programming stress. *IEEE Computer* 2000; **33**(4):30-38.
6. Walsh J, Sneeringer WJ, Hoare CAR. Ambiguities and insecurities in Pascal. *Software - Practice and Experience* 1977; **7**(6):685-696.
7. Sun. Java2 Platform Standard Edition program.
<http://java.sun.com/products/archive/>[2001].
8. IBM. Java Developer Kits program.
<http://www.ibm.com/developerworks/java/jdk/> [2001].
9. IBM. VisualAge for Java program.
<http://www.ibm.com/software/ad/vajava/> [2001].
10. Chan J-T, Yang W. An Attribute-grammar Framework for Specifying the Accessibility in Java Programs. Department of Computer and Information Science, National Chiao Tung University, 2001.
11. Yang W. Discovering anomalies in access modifiers in Java with a formal specification. *Journal of Object-Oriented Programming* 2001; **13**(10):12-18.