Available online at www.sciencedirect.com
SCIENCE DIRECT®
The Journal of Systems and Software 72 (2004) 33–47
The Journal of Systems and Software
www.elsevier.com/locate/jss

Traps in Java

Jien-Tsai Chan *, Wuu Yang, Jing-Wei Huang

Department of Computer and Information Science, National Chiao Tung University, 1001 Ta Hsueh Road, Hsinchu 300, Taiwan, ROC

Received 2 September 2002; received in revised form 15 January 2003; accepted 28 January 2003

Abstract

Though the Java programming language was designed with extreme care, there are still a few ambiguities and irregularities 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 justify our observations. The irregularities are issues that often confuse programmers. It is hard for a confused programmer to write robust programs. Furthermore, a few issues of the Java language are left intentionally open to the compiler writers. Their effects on Java programs are discussed. The problems of ambiguity, irregularity, and dependence on implementations frequently trap an incautious Java programmer. Some suggestions and solutions for the problems are provided in this paper.

Keywords: Java; Language design; Language definition; Object-oriented programming

1. Introduction

Since its first public release in 1994, the Java programming language (Gosling et al., 2000) has gained much popularity. Java is a derivative of C++ with some features borrowed from other programming languages, such as Objective C, Eiffel, Smalltalk, Cedar/Mesa, and Lisp (Gosling and McGilton, 1996). Therefore, many programmers find it familiar and easy to learn. Although Java has overcome or avoided most problems of these languages, new ambiguities and irregularities still appear in Java.

In this paper, we study Java from the perspective of a programming language. Several ambiguity and irregularity problems of Java are reported. Furthermore, the dependences on compiler implementations are also discussed. Other features of Java, such as security, exceptions, concurrency, and libraries are not considered here. Daconta et al.'s book (2000) discusses other features that are not included in this paper. Thimbleby (1999) discussed the pros and cons of various features of Java. The issues contained in his paper include the notational traps, the improvements 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. (2000) 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 significance of the problem. There is a similar research that is dedicated to the study of Pascal (Walsh et al., 1977).

The ambiguities are caused by the unclear specification of the Java language (Gosling et al., 2000). 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 (Sun Microsystems Inc., 2001), IBM JDK 1.2.2, 1.3.0 (IBM, 2001a), and IBM VisualAge for Java 4.0 (IBM, 2001b). The different results produced by the different compilers demonstrate that even these serious compiler writers cannot agree with one another on the ambiguities.

The irregularities are issues that could easily confuse programmers. These irregularities often lead to unexpected results when a program is executed. A seemingly naive program may access types, fields, or methods that are not the

* Corresponding author. Tel.: +886-933303945; fax: +886-35721490.
E-mail addresses: maxchan@cis.nctu.edu.tw (J.-T. Chan), wuuyang@cis.nctu.edu.tw (W. Yang), gis86809@cis.nctu.edu.tw (J.-W. Huang).

0164-1212/$ - see front matter © 2003 Elsevier Inc. All rights reserved.
doi:10.1016/S0164-1212(03)00040-2

ones the programmer intended. Unfortunately, Java compilers usually finish compiling such programs without any warning. Bugs like these are difficult to detect.

Several issues of Java are intentionally left open to compiler implementations. Different implementation methods of a compiler may produce different results. Examples are provided to show how these issues affect a program and in what the situations that the problems may happen.

The remainder of this paper is organized as follows. Section 2 introduces the ambiguity problems. Irregularity problems are discussed in Section 3. The dependences on compiler implementations are shown in Section 4. The last section is our conclusion.

2. Ambiguities

We found two ambiguities in Java. The first ambiguity is the accessibility to the default-access members of a class. It is caused by mixing the inheritance relationship and the package relationship. The Java specification does not define the priority of the two relationships for default-access members. This problem was mentioned in our previous work (Chan and Yang, 2002; Yang, 2001). 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 is caused by the flawed rules in the Java specification for choosing the *most specific* method.

2.1. 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 in Java is either 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 can be accessed only by the code within the same package that the class `C` belongs to.

Consider the example in Fig. 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 (Gosling et al., 2000), a default-access member can be accessed within the package in which the type it belongs to is declared. The field `x` declared in class `A` has the default access. 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`.

```
//------------------------- 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, depending on compiler
    int getX() { return x; }
}
```

Fig. 1. Example 1 for the accessibilities of members.

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

Compiler	Result
Sun JDK 1.0.2	Passed
Sun JDK 1.1.8	Passed
Sun JDK 1.2.2	Error: Undefined variable: x
Sun JDK 1.3.0	Error: x is not public in pkg1.A; cannot be accessed from outside package
IBM JDK 1.2.2	Error: Undefined variable: x
IBM JDK 1.3.0	Error: x is not public in pkg1.A; cannot be accessed from outside package
IBM VisualAge for Java 4.0	Passed

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 the inheritance relationship sometimes tell us different stories. The ambiguity problem exists because the Java language specification (Gosling et al., 2000) 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 Fig. 2. Similar to the previous example, the program has the same inheritance structure and package structure. In class `A`, method `m1` is declared as an abstract method. According to the Java language specification (Gosling et al., 2000), a class must implement all the inherited abstract methods unless that class is declared as an abstract class. The inherited method `m1` is invisible in `B` because `B` is in a different package. The method `m1` is visible in `C` according to the package relationship but it is also invisible in `C` if we follow the inheritance relationship.

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. When method `m1` is not defined in `B`, all compilers complain that `B` should be declared as an abstract class. 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 JDK 1.2.2 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 `B` should be declared as an abstract class because it does not implement the inherited abstract method `m1`. IBM JDK 1.3.0 and Sun JDK 1.3.0 and IBM VisualAge for Java 4.0 give an error with the same complaints as JDK 1.2.2. In our opinion, it is not necessary to implement an invisible inherited abstract method in class `B`. The compilers should compile class `B` successfully no matter whether the definition of `m1` exists or not.

To show the ambiguity in class `C`, class `B` is declared as an abstract class without any 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

```
//------------------------- 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?
}
```

Fig. 2. Example 2 for the accessibilities of members.

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

Compiler	If `m1` is defined in class `B`	If `m1` is defined in class C and `m1` is not defined in class `B`
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

column of Table 2 shows the compilation results 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 if `m1` defined in `C`. To these compilers, the definition of `m1` in `C` seems useless. Other compilers compile class `C` successfully. The specification of the Java language is unclear about this case.

Another problem is concerned with method overriding between static methods and instance methods. Consider the example in Fig. 3. Similar to the previous example, the program has the same inheritance structure and package structure. 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 (Gosling et al., 2000), an inherited static method cannot be overridden by an instance method of the same signature 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 both 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 both IBM and Sun and IBM VisualAge for Java 4.0 finish the compilation successfully.

```
//------------------------- 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; }
}
```

Fig. 3. Example 3 for the accessibilities of members.

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

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

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 both 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 believes that the default-access members of class A are invisible in class C. However, in the second example, the same compiler believes that the default-access abstract method of class A is visible in class C. The inconsistent behavior of a compiler can really confuse a Java programmer. Java programmers will be further confused by the inconsistent behavior among different compilers.

2.2. Method invocations

Java allows both method overloading and method overriding. The process of determining the actual method to be invoked is partitioned into two phases. The first phase is done by the compiler (at compile time). The second phase 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 is visible in the current context. If there are one or more such methods, the compiler chooses the one with the *most specific* signature based on the declared types of the actual parameters in A.m(···). (The exact rules are quite complicated, see Section 12.15 of (Gosling et al., 2000).) 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 specification to select a method (see Section 12.15 of (Gosling et al., 2000)). However, the rules are ambiguous in certain situations related to the widening conversion. Different compilers produce different results in these situations.

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 during the conversion. A *widening reference conversion* is a conversion between non-primitive types (i.e. classes and interfaces) 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 originates from the mixture of widening conversion, method overloading, and class inheritance. Consider the following example.

```
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); // which m?
    x.p(d); // which p?
  }
}
```

Table 4
The compilation results of the program in Section 2.2

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

Class `D` inherits class `C` and `C` inherits class `B`. Class `B` inherits class `A`.Class `M` inherits class `N`. The methods named `m` and `p` use parameters of primitive types and reference types, respectively. The problem lies in which methods should be involved in the last two lines in the `main` method. The results of compiling the example with different compilers are shown in 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 similar error messages but the ambiguous 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 problem of method invocation happens when there are both an applicable local instance method and a more appropriate inherited instance method and then the methods are invoked with widening conversions of parameters. According to the Java specification, both the applicable local instance method and the more appropriate inherited instance method are the most specific methods. Therefore, the compilers have to issue error messages due to 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, i.e. the inherited method.

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

3. Irregularities

In this section, we will discuss the irregularity about import declarations, members in a type, and the *protected* modifier. Import declarations have different effects for top-level types (including interfaces and classes) and member types (i.e. types nested in other types). The rules for declaring and accessing fields and methods are irregular and confusing. These complicated rules for members of a type are discussed in the second subsection. In Java, the *protected* modifier carries a different meaning than its counterpart in C++. The problems with the *protected* modifier are discussed in Section 3.3.

3.1. Import declarations

Import declarations are used to import desired types (i.e. classes and interfaces) into a compilation unit such that these types can be referenced by their simple names. Otherwise, the named types in packages that are not imported must be referenced with the fully qualified names (i.e. the package name plus the simple type name).

There are two kinds of import declarations. A *single-type-import* declaration imports a single type into a compilation unit. A *type-import-on-demand* declaration imports all the public types of a package into a compilation unit. Although the naming structure of a package is hierarchical, such as `A.B.C.D`, there are not any relationships between packages and their subpackages except they have similar names. Types of a package are not visible in its superpackages [1] and subpackages unless they are explicitly imported. Similarly, a package cannot reference the types of its superpackage and subpackages without importing them first. There are no relationships among sibling packages either. An import declaration does not import the subpackages of the imported package.

[1] The package such as `A.B.C.D` is a *subpackage* of the package `A.B.C`. Similarly, `A.B.C` is a *superpackage* of `A.B.C.D`. `A.B.C.D` and `A.B.C.E` are called *sibling packages*.

Import declarations import the types in other packages as well as the nested types in a type. Therefore, Java programmers are sometimes confused when the types in a package and the types nested in a type are imported by type-import-on-demand declarations simultaneously. Consider the following example.

```
//in file A.java
package p.q;
public class A {
  public class B {
    public class C {}
    public C createC() {return new C();}
  }
  public class D {}
  public B createB() {return new B(); }
  public D createD() {return new D(); }
}
// in file X.java
package r;
import p.* ;
import p.q.A.*;
class X {
  q.A a=new A(); // error, subpackage is not imported automatically
  p.q.A a=new p.q.A(); // must use the fully qualified name
  B b=a.createB();
  B.C c=b.createC(); // Yes, it is correct.
  D d=a.createD(); // all inner classes of A can be accessed
}
```

In the first compilation unit, class `A` belongs to package `p.q`.class `B` and class `D` are inner classes of `A`. Class `C` is an inner class of `B`. In the second compilation unit, class `X` belongs to package `r`. The first type-import-on-demand declaration imports all the types in package `p`. But subpackage `q` of package `p` is not imported. Furthermore, the name like `q.A` is neither a valid fully qualified name nor a valid simple name. Therefore, the declaration in the first line of class `X` is not valid. The only way to reference the class `A` is by the fully qualified name `p.q.A`.

The second type-import-on-demand declaration imports all the inner classes (`B` and `D`) of class `A`. Unlike the previous situation, class `C` can be referenced with the name `B.C`. The name `B.C` is neither a fully qualified name nor a simple name. If programmers do not follow the naming convention (i.e. use lowercase letters for package names and capitalize the first letter of each word for type names) of Java (see Section 6.8 of (Gosling et al., 2000)), they can be easily confused by the package names and class names in the above example.

The problem is that the type-import-on-demand declarations do not treat the package structure and the type nesting structure equally.

3.2. Members of a type

Fields and methods of a type are not treated equally in Java. There are two similar but different (hence confusing) sets of rules for fields and methods, respectively. Some rules are only applied to fields. Some rules pertain only to static methods. Some rules pertain only to instance methods. These irregularities often confuse a new, sometimes even experienced, Java programmer. In this section, we will discuss several confusing rules for members of a type.

3.2.1. Overloading

Java allows method overloading. Overloading means that two methods of a type can have the same name but different signatures. The *signature* of a method consists of the name of the method and the number and types of its formal parameters. Note that the return type and the *throws* clause are not parts of the signature of a method in Java. On the other hand, two fields of a class cannot have the same name, even if they have different types. That is, overloading is allowed for methods, but not for fields.

```
public class X {
  int f1;
  float f1; // error, overloading is not allowed for fields
  int m1(int p1) {...}
  int m1(int p1, int p2) {...} // ok, this is overloading
  boolean m1(int p1) {...} // error, return type is not a part of the signature
}
```

For example, in the above program, the second field declaration named `f1` is illegal because fields cannot be overloaded. The first `m1` method and the second `m1` method are overloaded correctly since they have different numbers of parameters. The third `m1` method has the same signature as the first `m1` method even though their return types are different. Because Java does not allow two methods to have the same signature but different return types in the same class, the declaration of the third `m1` method is illegal.

3.2.2. Overriding

A class can redefine the methods inherited from its superclasses. The method in the subclass *overrides* the method with the same signature inherited from the superclass.

In a subclass, it is allowed to define a field with the same name as the one inherited from the superclass. The field in the subclass *hides* the field with the same name inherited from the superclass.

According to the specification of the Java language (Gosling et al., 2000), a method and the corresponding overridden method must be declared with the same return type. However, a field and the corresponding hidden field can be declared with different types. Consider the following example.

```
public class X {
  String f2;
  String m2() {...}
}
public class Y extends X {
  int f2; // ok, this.f2 hides f2, different types are allowed
  int m2() {...} // error, different return type
}
```

The `f2` field in class `Y` hides the `f2` field in class `X` even though the two fields have different types. The `m2` method of `Y` and the `m2` method of `X` have the same signature. However, the declaration of the `m2` method in `Y` is illegal since its return type differs from that of the overridden `m2` in `X`. The return type of a method is ignored when methods are overloaded but is significant when methods are overridden.

Java uses different rules to determine the fields and the methods that are actually referenced or invoked. The mixture of field hiding and method overriding could easily confuse programmers. Consider the following example.

```
class G {
  public String name="G";
  public String getMyName() {
    System.out.println(this.getClass().getName()); // print "H"
    return this.name;
  }
}
public class H extends G {
  public String name="H";
  public static void main(String[] args) {
    H h=new H();
    System.out.println("h.getMyName()="+h.getMyName()); //h.getMyName()=G
  }
}
```

The `main` method invokes the `getMyName` method of class `G`. The method `getMyName` first prints the character "H" (due to the call `this.getClass().getName()`). However, amazingly, the expression `this.name` (the second line of the `getMyName` method) returns "G", not "H". If we add the following method into class `H`,

```
public String getMyName() {
  return this.name;
}
```

the result will become "h.getMyName()=H". The reason is that field references are statically determined at compile time in Java. However, the instance method invocations are determined at run time. The value of the field name depends on the type that the actually invoked method belongs to. When only the method getMyName in class G is available, the method invocation h.getMyName() refers to getMyName of class G. Therefore, the result is "h.getMyName()=G". If each of class G and class H defines a getMyName method, the method invocation h.getMyName() refers to getMyName of class H. The result becomes "h.getMyName()=H".

In Java, only instance methods are bound dynamically (i.e. at run time). On the other hand, fields, static methods, and member types are bound statically (i.e. at compile time). Furthermore, instance methods are still not truly dynamically bound. As we mentioned in Section 2.2, the compiler first determines the signature of the method to be invoked based on the declared types of the parameters. Then the run-time system searches the inheritance hierarchy for a method that has the required signature. Therefore, the actual type of the parameter does not affect the method to be invoked. Consider the following example.

```
class A {}
class B extends A {}
class T {
  public void m1(A a) {System.out.println("T.m1(A)");}
  public void m2(A a) {System.out.println("T.m2(A)");}
}
class V extends T {
  public void m1(A a) {System.out.println("V.m1(A)");}
  public void m2(B b) {System.out.println("V.m2(B)");}
}
public class Test {
  public static void main(String[] args) {
    T t=new V();
    A a=new A();
    A b=new B();
    t.m1(a); // V.m1(A)
    t.m1(b); // V.m1(A)
    t.m2(a); // T.m2(A)
    t.m2(b); // T.m2(A) // Surprise!!
  }
}
```

Method m1 in class V overrides method m1 in class T. Methods m2 in class V and in class T are overloaded. The variable t is declared as type T and it references to an object of type V. Both the method invocations t.m1(a) and t.m1(b) are dynamically bound to method m1 of class V. The two invocations of m2 are bound to method m2 of class T. Although the actual type of b is class B, it does not affect the selected method because the compiler dictates the run-time system to search for a method whose signature is m2(A).

3.2.3. Static and instance members

A static field can hide an inherited instance field (of the same name, of course). Similarly, an instance field can hide an inherited static field. However, this rule does not apply to methods. Consider the following example.

```
public class X {
  int f3;
  static int f4;
  int m3() {...}
  static int m4() {...}
}
```

```
public class Y extends X {
  static int f3; // ok
  int f4; // ok
  static int m3() {...} // error
  int m4() {...} // error
```

The static field `f3` of class `Y` hides the inherited instance field `f3` of class `X`. Similarly, the instance field `f4` of `Y` hides the inherited static field `f4` of `X`. Both field declarations in `Y` are legal. According to the specification of the Java language (Gosling et al., 2000), however, a static method cannot override an inherited instance method. Neither can an instance method override an inherited static method. The method `m3` of `Y` attempts to override the inherited instance method `m3` of `X`. The method `m4` of `Y` is attempts to override the static method `m4` of `X`. Both method declarations—`m3` and `m4`—of `Y` are illegal.

Hidden static fields and hidden static methods of a superclass can be accessed by a qualified name or by an expression that contains the keyword *super* or a cast to the superclass. Hidden instance fields of a superclass can be accessed in the same ways as hidden static fields are accessed except that we cannot use the qualified name. But hidden instance methods of a superclass can only be accessed by the keyword *super*. The following example shows all the possible ways to access the fields and the methods of the superclass. Each line is annotated with a line number. The result of compiling and running the program is shown in the comments of the corresponding statements.

```
01: import java.io.*;
02: class X {
03:   public String f1="X.f1";
04:   public static String f2="X.f2";
05:   public String m1() {return "X.m1()";}
06:   public static String m2() {return "X.m2()";}
07: }
08: public class Y extends X {
09:   public String f1="Y.f1";
10:   public static String f2="Y.f2";
11:   public String m1() {return "Y.m1()";}
12:   public static String m2() {return "Y.m2()";}
13:   public static void prn(String s) {System.out.println(s);}
14: public void test() {
15:     X p=new Y();
16:     Y q=new Y();
17:     // use fully qualified name
18:     // prn("X.f1="+X.f1); //error because f1 is an instance field
19:     prn("X.f2="+X.f2); // X.f2=X.f2
20:     // prn("X.m1="+X.m1()); // error because m1 is an instance method
21:     prn("X.m2()="+X.m2()); // X.m2()=X.m2()
22:     // use the keyword super
23:     prn("super.f1="+super.f1); // super.f1=X.f1
24:     prn("super.f2="+super.f2); // super.f2=X.f2
25:     prn("super.m1()="+super.m1()); // super.m1()=X.m1()
26:     prn("super.m2()="+super.m2()); // super.m2()=X. m2()
27:     // use the a reference
28:     prn("p.f1="+p.f1); // p.f1=X.f1
29:     prn("p.f2="+p.f2); // p.f2=X.f2
30:     prn("p.m1()="+p.m1()); // p.m1()=Y.m1()
31:     prn("p.m2()="+p.m2()); // p.m2()=X.m2()-Surprise!!
32:     // use casting
33:     prn("((X)q).f1="+((X)q).f1); // ((X)q).f1=X.f1
34:     prn("((X)q).f2="+((X)q).f2); // ((X)q).f2=X.f2
35:     prn("((X)q).m1()="+((X)q).m1()); // ((X)q).m1()=Y.m1()
36:     prn("((X)q).m2()="+((X)q).m2()); // ((X)q).m2()=X.m2()
37:   }
```

```
38:   public static void main(String[] args) {
39:     Y y = new Y();
40:     y.test();
41:   }
42: }
```

The field `f1` is an instance field and the field `f2` is a static field in `X`. The method `m1` is an instance method and the method `m2` is a static method in X. The types of the references `p` and `q` are `X` and `Y`, respectively. Both `p` and `q` refer to objects of type `Y`. In the above program, in lines 18–21, fully qualified names are used to access the members of `X`. Because instance fields and instance methods can be accessed only through a reference to an object, the references to `f1` and `m1` (in line 18 and line 20, respectively) will cause errors. The other two statements (in line 19 and line 21) can access the static members `f2` and `m2` correctly.

In lines 23–26, the members of `X` are accessed through the keyword super. The results are straightforward. In lines 28–31, the members of `X` are accessed through the reference `p`. Although the type of `p` is `X`, the method invocation `p.m1` actually invokes the method `Y.m1` because the type of the object that `p` actually references to is `Y`. However, dynamic dispatching affects only instance methods. The static methods that are actually invoked are determined by the compiler based on the declared type of references. Therefore, the method invocation `p.m2` still refers to the method `X.m2` since the type of `p` is `X`.

In lines 33–36, the program tries to access the members of `X` by casting the reference `q` to type `X`. From the results, we conclude that casting is useless when calling an instance method. The actually invoked instance methods are determined by the type of the referenced object, not the type of the reference. Casting the reference does not affect the instance method to be invoked. But it does affect the fields (e.g. `f1` and `f2`) to be used and the static methods (e.g. `m2`) to be invoked.

If the programmers do not understand the above rules clearly, their programs are suspicious.

3.2.4. Access privileges

In Java, the strength (from strong to weak) of access privileges is public, protected, default-access (i.e. without any modifier), and private. Java requires that a method cannot be declared with a weaker access privilege than the corresponding overridden methods, which are inherited from the superclasses. This restriction pertains to both instance methods and static methods. But this restriction is not applicable to fields. Consider the following example.

```
public class X {
  public int f5;
  public int m5() {...}
}
public class Y extends X {
  protected int f5; // ok
  protected int m5() ... // error
}
```

The field `f5` of class `Y` has a weaker access privilege (`protected`) than the inherited field `f5` (`public`) of class `X`. This is allowed. The declaration of the method `m5` in class `Y` is illegal because it tries to override the method `m5` in class `X`, which has a stronger access privilege.

3.3. The protected access modifier

The accessible scope of a protected member is different in Java and in C++. In C++, a protected member in class `C` is accessible in `C` and in all the descendant classes of `C`. In Java, however, a protected member in class `C` is also accessible to the classes that belong to the same package as `C`. This extension will cause some insecurities for Java programs. For example, the well-known `Singleton` design pattern (Gamma et al., 1995) is illustrated in the following code.

```
package p;
public class Singleton {
  private static Singleton anInstance = null;
  protected Singleton() {}
  public static Singleton getInstance() {
```

```
    if(anInstance==null) {
     anInstance=new Singleton();
    }
    return anInstance;
  }
  public int x=1 ; // an extra field for the next example.
}
```

The purpose of the `Singleton` class is to ensure that only one instance of the class exists. The constructor of the `Singleton` class is declared as protected so that only the `Singleton` class and its descendants can call the constructor. The only way to get an instance of the `Singleton` class is through the `getInstance` method. This restriction works in C++ but does not work in Java. In the above example, the `Singleton` class belongs to the package `p`. All other classes in package`p` can call the constructor. Therefore, the assurance that only one instance exists is violated. The following program illustrates an example that creates two instances.

```
package p;
public class Main {
  public static void main(String[] args) {
    Singleton s1=new Singleton();
    Singleton s2=Singleton.getInstance();
    s2.x=2;
    System.out.println("s1.x="+ s1.x+", s2.x="+ s2.x);
  }
}
```

The result of the above program is "s1.x = 1, s2.x = 2". This result proves our observation. Not only the `Singleton` class is affected by the protected access modifier, but also all the programs that intend to use the `protected` access modifier to restrict the accessibility of members within the declared class and its descendants. C++ programmers are likely to ignore the difference of the protected access modifier. One solution to this problem is to put the `Singleton` class and all its descendant classes in a separate package. The constructors are still declared as `protected`. No other classes exist in the same package. This way we can achieve the same effect of the protected access modifier in C++. Therefore, similar applications of the protected access modifier, such as the `Singleton` pattern, will be almost as robust as in C++. Because we cannot prevent a programmer from declaring classes in the same package as the `Singleton` class, this solution is not perfect. The solution is for the designers of a library, not for the users of the library.

Another solution is similar to the previous one but the constructors has the default-access privilege. In this way, the `Singleton` class is restricted to be inherited only by the classes in the same package. However, it still cannot prevent other classes in the same package from invoking the constructors of `Singleton`. Anyhow, both solutions cannot achieve the same effect as the `protected` modifier in C++ unless programmers carefully isolate the `Singleton` class and its descendant classes.

4. Dependence on compiler implementations

Some issues of the specification of the Java language are open to the compiler writers. This means that the behavior of a Java program will be different under different compiler implementations. In this section, two such issues are presented.

4.1. The restriction of one public type per compilation unit

If packages are stored in a file system, some restrictions are applied to the compilation units. A compilation unit is a single file stored in the path indicated by the package name in the file system. The types in a compilation unit can be declared with the `public` modifier or without modifier (default-access). But at most one public type can exist in a compilation unit and the public type must have the same name as the file. Other types in the same compilation unit must be declared without modifiers and can only be referenced within the package the compilation unit belongs to.

The benefit of this restriction is that the compiler can perform type checking efficiently. Only the public types in a package are imported by the `import` declaration. With the above restrictions, when checking types, the compiler will

search for the source file (the name of the referenced type append with ".java") or the bytecode file (the name of the referenced type name append with ".class") in the directory indicated by the package name. If both files are found, the compiler will use the newer one. Then the compiler reconstructs the class structure from the bytecode file or compiles the source file implicitly (but does not produce the bytecode). Problems happen when a default-access type that is defined in the compilation unit with a name other than the name of the source file is referenced. Consider the following example.

```
// in file M.java
package r;
public class M {
  P p=new P();
}
class Q {}
// in file N.java
package r;
public class N {
  Q q=new Q();
}
class P {}
```

The compilation units `M.java` and `N.java` belong to package `r`. The two compilation units reference the default-access types of each other. Class `M` references class `P` in `N.java`. Class `N` references class `Q` in `M.java`. It is impossible to compile the two files one by one. The compiler complains that certain symbols are not resolved. The only way of successful compilation is to compile the two files at the same time.

If packages are stored in a database system, the above restriction does not apply. For example, IBM VisaulAge for Java uses a database system (called the ENVY repository) to store Java programs. There is not such a restriction in VisualAge for Java.

4.2. The scope of unnamed packages

Packages can be named or unnamed (anonymous). If a compilation unit does not specify the package it belongs to, it will be put in an unnamed package. The number of unnamed packages in a program is not defined in the Java specification (Gosling et al., 2000). The specification only *suggests* there is one unnamed package. The way a Java platform supports unnamed package(s) will affect the accessibility between classes that belong to the unnamed package(s). If the Java platform supports only one unnamed package (this is also the approach that all currently available compilers take), the types in different directories but belonging to the unnamed package are considered to be in the same package. On the other hand, if the Java platform supports one unnamed package per directory, the unnamed packages are considered distinct packages. Consider the following example.

```
// in file X.java, stored in the directory p
class X {
  protected int a;
  int b;
}
// in file Y.java, stored in the directory q
class Y {
  protected int c;
  int d;
}
```

The two classes `X` and `Y` belong to unnamed packages and are stored in the directories `p` and `q`, respectively. Furthermore, the two classes are declared as default access, which means that they are only accessible to the types in the same package. If the Java platform supports exactly one unnamed package per program and the environment variable *CLASSPATH* includes the directories `p` and `q`, class `X` and class `Y` are considered to be in the same package. However, if the Java platform supports one unnamed package per directory, `X` and `Y` are considered to be in different packages.

Sometimes, it happens that a type belonging to an unnamed package may be stored in a directory that is used by a named package. The types in the named package are unable to use the type in the unnamed package even though they exist in the same directory. Consider the following example.

```
// in file X.java, stored in the directory p
package p; // belong to the package p
public class X {
  Y a=new Y(); // error
}
// in file Y.java, also stored in the directory p // belong to the unnamed package
public class Y {
}
```

Class X belongs to package p. Class Y belongs to the unnamed package. Both X and Y are stored in directory p. In class X, the reference to Y will cause a compilation error because Y is in a different package than X, even though class Y is public. Furthermore, there is no way to import an unnamed package. Consequently, class X is unable to reference class Y. Actually, no types in the package p can reference the class Y. In this case, the file Y.java or Y.class should be put in a directory that appears in the *CLASSPATH* environment variable.

When the Java platform supports only one unnamed package, the value of the environment variable *CLASSPATH* may change the result of a program. Consider the following example.

```
// in file M.java, stored in the directory p
public class M {
  public static void main(String[] args) {
    System.out.println("N.x="+ N.x);
  }
}
// in file N.java, stored in the directory p
public class N {
  public static int x=1;
}
// in file N.java, stored in the directory q
public class N {
  public static int x=2;
}
// in file N.java, stored in the directory r
public class N {
  public static int x=3;
}
```

In the above example, class N is referenced in the main method of class M. There are the classes named N, one in each directory of p, q, and r, respectively. The run-time system searches each directory listed in the *CLASSPATH* variable in order for a file named N.class and finally the directory where class M resides. The first N.class found will be used. If the value of *CLASSPATH* does not include the directories p, q, and r, the class N in the directory p is used. The result of the program is "$N.x = 1$". However, when the *CLASSPATH* variable is set to "q;r;p", the class N in the directory q is used. The result of the program is "$N.x = 2$". The result of the program will become "$N.x = 3$" if the value of the *CLASSPATH* variable is set to "r;q;p". We can find that the first class found is used. The result of the program depends on the value of the *CLASSPATH* variable. This could become a trap for programmers. This issue does not only affects the unnamed package but also two packages that have the same name and are stored in different directories. However, if the Java platform supports one unnamed package per directory, the value of the *CLASSPATH* variable will not affect the selected classes that belong to the unnamed packages.

Because unnamed packages may affect the accessibility of the default-access types and protected and default-access members (and hence the behavior of the program may be changed when unnamed packages are used), we suggest that the number of unnamed packages should be defined as one per directory. The *CLASSPATH* problem will only affect named packages. Furthermore, the scope rules will become clearer and more stable.

5. 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 identical behavior, no matter which compiler is used to compile the program and no mater when and where the program runs. The ambiguity problems are the major barriers to achieve the above goals. When designing a programming language, it is very hard to discover the hidden ambiguity problems systematically. Instead of a verbal 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.

Programmers should be careful when using the irregular features. The irregularities are not a fault of the language design but a trade-off between simplicity and functionality. However, they may cause some implicit semantic errors that are hard to find.

Acknowledgement

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

References

Alexander, R.T., Bieman, J.M., Viega, J., 2000. Coping with Java programming stress. IEEE Computer 33 (4), 30–38.

Chan, J.-T., Yang, W., 2002. An Attribute-Grammar Framework for Specifying the Accessibility in Java Programs. Computer Languages, Systems & Structures 28 (2), 203–235.

Daconta, M.C., Monk, E., Keller, J.P., Bohnenberger, K., 2000. Java Pitfalls: Time-Saving Solutions and Workarounds to Improve Programs. John Wiley & Sons, New York.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA.

Gosling, J., Joy, B., Steele, G., Bracha, G., 2000. The Java Language Specification, second ed. Addison-Wesley, MA.

Gosling, J., McGilton, H., 1996. The Java Language Environment White Paper.

IBM, 2001a. Java Developer Kits. ver. http://www.ibm.com/developerworks/java/jdk/.

IBM, 2001b. Visualage for Java. ver. 4.0. http://www.ibm.com/software/ad/vajava/.

Sun Microsystems Inc., 2001. Java2 Platform Standard Edition. ver. 1.3.1. http://java.sun.com/products/archive/.

Thimbleby, H., 1999. A critique of Java. Software—Practice and Experience 29 (5), 457–478.

Walsh, J., Sneeringer, W.J., Hoare, C.A.R., 1977. Ambiguities and Insecurities in Pascal. Software—Practice and Experience 7 (6), 685–696.

Yang, W., 2001. Discovering anomalies in access modifiers in Java with a formal specification. Journal of Object-Oriented Programming 13 (10), 12–18.