On the Interaction of Method Lookup and Scope with Inheritance and Nesting

Gilad Bracha

Cadence Design Systems
gilad@cadence.com

Abstract. Languages that support both inheritance and nesting of declarations define method lookup to first climb up the inheritance hierarchy and then recurse up the lexical hierarchy. We discuss weaknesses of this approach, present alternatives, and illustrate a preferred semantics as implemented in Newspeak, a new language in the Smalltalk [GR83] family.

1 Introduction

The combination of inheritance and block structure is potentially very powerful, as shown in the language Beta [MMPN93], and later languages such as gBeta [Ern01] and Scala [Ode07]. The idea has gained currency in the mainstream with the introduction of nested classes in Java^{TM}[1].

In all these languages, situations such as the following (illustrated in Java) can arise:

```
class Sup {  }
class Outer {
  int m(){ return 91}
  class Inner extends Sup {
    int foo(){return m()} // case 1: new Outer.Inner().foo() = 91
  }
 }
```

The expectation is that a call to foo will yield the result 91, because it returns the result of the call to m, which is defined in the enclosing scope.

In general, the semantics of a method invocation that has no explicit target (receiver) are that method lookup begins by searching the inheritance hierarchy of **self** (aka **this**); if no method is found, then the lookup procedure is repeated recursively at the next enclosing lexical level. This notion is described in detail in the Java Language Specification [GJSB05] and formalized in [SD03]. It is called

[1] Java is a trademark of Sun Microsystems.

"comb semantics" in NewtonScript [Smi]. [2] Consider what happens if one now modifies the definition of Sup:

```
class Sup {
 int m(){ return 42}  // case 2: new Outer.Inner().foo() = 42
}
```

The result of calling foo is now 42.

This is undesirable; the behavior of the subclass changes in a way that its designer cannot anticipate. Of course, inheritance in general suffers from modularity problems, but there is no reason to aggravate them further.

The issue has received scant attention in the literature (though see [Fri] section 8.4). If this problem is so acute, why has it not been addressed previously? We believe this is because the combination of block structure and inheritance has not been used aggressively on a sufficient scale.

The only widely used language that supports such nesting is Java. Java nested classes are used in very restricted, stylized ways. They are often used simply for packaging; the nested classes are static and the scope of the enclosing class is inaccessible to them so the issue does not arise. The situation in Python is similar: nested classes have no access to the scope of their enclosing class.

Aggressive use of class nesting offers considerable possibilities. In addition to the classic techniques using nested and especially virtual classes [MMP89] demonstrated by the Beta community, nested classes can enable powerful features such as mixins, class hierarchy inheritance and modules. This paper focuses, however, on one specific technical problem: the appropriate rules for looking up method names in the presence of block structure and inheritance.

The remainder of this paper expands upon this point and investigates alternative semantics. We have implemented one such alternative as part of the development of Newspeak, a new language descended from Smalltalk. Newspeak differs from Smalltalk in a number of ways, in particular in its support for arbitrary class nesting.

[2] NewtonScript has no lexical nesting, but its lookup semantics are nevertheless essentially the same

2 Relevant Highlights of Newspeak

Here is how we could write case 1 above in Newspeak

```
class Sup = ()
class Outer  = (
  m = (^91)
  class Inner = Sup (
    foo = (^m)
  )
)
```

A class declaration begins with the word **class**, followed by the class name, followed by an equal sign. Then the name of the superclass may be given; if it is omitted, it is assumed to be Object. Next comes a class body delimited by parentheses. Member declarations such as methods and nested classes appear in the class body.

Our example includes two methods, m and foo. A method begins with a Smalltalk style message pattern giving its name, followed by an equal sign, with the method body again delimited by parentheses. Inside m, the body is a return statement, identical to the Smalltalk return statement. The method foo is part of the nested class Inner. Its body is a return statement that consists of a send of m.

Each class in Newspeak has a *primary constructor*, which is a class method that can be used to create new instances. By default, the primary constructor is called new. The expression Outer new m will yield a value of 91, as will the expression Outer new Inner new foo. The latter expression shows how to create an instance of a nested class by sending a message to an instance of the enclosing class.

Each instance of an enclosing class has its own, unique, nested classes. To see why this must be the case, consider the code below.

```
class Outer1 s: aClass  = (
| SuperInner = aClass. |
  m = (^91)
  class Inner = SuperInner (
    foo = (^m)
  )
)
```

The primary constructor of Outer1 is defined explicitly by the message pattern s: aClass immediately after the class name. Outer1 is parametric with respect to the formal parameter aClass. When an instance of class Outer1 is created, as

in Outer1 s: Object, the instance's SuperInner slot is bound to the actual parameter Object. Hence, the superclass of Inner in this case is Object. In contrast, in the instance Outer1 s: Number, Inner is a subclass of Number. Obviously, different instances have distinct Inner classes. Each nested class is also bound to the state of the enclosing instance, so each instance must have its own unique nested classes.

The example above shows one way of deriving mixins. This feature is an inevitable consequence of the combination of class nesting and first-class class objects, and is reminiscent of the work on Units [FF98].

3 What are the Desired Semantics?

The "comb rule" has the advantage of regularity and simplicity, which should never be discounted, but it does suffer from a serious problem, namely the risk of unforeseen name capture by superclasses as described above. Can we do better?

It is often possible to define a problem away. In Smalltalk, for example, an explicit receiver is always required. If we introduce a way of designating the desired receiver for a call like the one to m in

```
foo = (^m)
```

the problem literally disappears. In Newspeak, we could rewrite our example as

```
foo = (^outer Outer m)
```

The notation outer N designates the innermost enclosing instance of class N. For a definition of the concept of enclosing instance, see [GJSB05], section 8.1.3.

However, Newspeak does not *require* an explicit receiver for every method invocation. Unlike Smalltalk, Newspeak is an entirely message-based language. All computation is done by (usually synchronous) message passing, as in Self [US87] and Emerald [BHJL86]. In particular, note that any reference to a nested class is in fact a send invoking an implicit accessor method that returns the corresponding class object. Hence all nested classes are also virtual classes; they are dynamically bound, and their definition may be overridden in subclasses.

The message-based approach has the well-known advantage that code is completely independent of object representation. It would be unduly burdensome to have to specify an explicit receiver for every message send in a language where everything is a message send.

Another possibility is to state that

(a) case 2 is ambiguous, and should yield a compile-time error.

Setting aside the question of whether such rules are compatible with the spirit of dynamic languages, the proposal has an obvious flaw: the fact remains that when the change is made to the superclass, Sup, the subclass is invalidated.

The argument can be made that it is better to invalidate the class statically, with a diagnostic clearly identifying the problem, than to allow the program to execute and misbehave in ways whose cause may be difficult to ascertain. However, our experience has been that it is extremely difficult to foresee the many interactions between such static rules.

It is difficult to adapt these rules to changing circumstances. For example, dynamic linking or hotswapping may create circumstances where the running program differs from the compiled one. Rules such as (a) may be costly to implement at run time, and their intended result, terminating a running program, is undesirable. It is much better to have simple rules that allow the program to execute in the face of dynamic change.

Moreover, we argue that the circumstances of case 2 above are not ambiguous at all. Consider the following example, given in Newspeak syntax.

```
class Outer2 s: aClass = (
 | Sup = aClass |
   m = (^91)
  class Inner = Sup s: Sup (
    foo = (^m)
  )
)
```

The example declares a class Outer2, with a nested class Inner, whose superclass, Sup, is an incoming parameter to Outer2's constructor, much as the previous example. The difference is that we now expect Sup itself to have a non-trivial constructor, which gets passed Sup itself as its argument. Here is an interesting instantiation of Outer2:

```
Outer2 s: Outer2
```

We now have an object with a nested class that is a subclass of its enclosing class. Now consider the meaning of the expression (Outer2 s: Outer2) Inner new foo or, better yet, of

```
(Outer2 s: Outer2) Inner new Inner new Inner new foo.
```

Each instance of Inner here is a subclass of Outer2, and hence itself has its own distinct member class Inner, which is also a subclass of Outer2 and so on ad infinitum. This works because nested classes in Newspeak are created lazily

on demand, when a message requesting them is processed by their enclosing instance.

Under standard "comb" semantics, however, the program makes for a nice puzzler in the Java tradition [BG05]. Note that since Inner is a subclass of Sup, which is bound to aClass, which is bound to Outer2, Inner has its own slot named Sup. The call to the superclass constructor, s: Sup, occurs within the scope of Inner, so that incoming parameters may be passed on to the superclass, and possibly processed by invoking methods of the subclass (Inner in our example).

Using the "comb" rule, the inherited slot Sup has priority over the lexically enclosing slot of the same name. But since the superclass constructor has not yet been invoked, the inherited slot is not yet initialized, and is therefore **nil**. So the Inner class inside an instance of Inner has a **nil** superclass, which is unfortunate to say the least.

We do not believe that an abundance of such puzzlers is a language feature to be emulated. Language design should not follow the principle of maximal astonishment.

The behavior of this program in Newspeak is of course different. In Newspeak an identifier refers to the nearest lexically visible declaration, subject to overriding by subclasses. If no lexically visible binding exists, we interpret it as a self send. An alternative we are considering is to treat the latter situation as an error, effectively requiring any reference to an inherited method (or one defined only in a subclass) to use **self** as its explicit receiver.

If one wishes to refer to an inherited method of an enclosing class, the outer N syntax described earlier expresses this intent unambiguously.

While the code in Outer2 is inherently tricky, we believe its behavior under the Newspeak semantics is more understandable and conforms to a strong tendency people have to rely on lexically visible bindings. We argue further that such examples have an unambiguous intended interpretation, and reporting them as ambiguity errors is unjustified. Finally, let's revisit our original problem (case 2)

```
class Sup = (m = (^42))
class Outer  = (
  m = (^91)
  class Inner = Sup (
    foo = (^m)
  )
)
```

It should now be clear that Outer new Inner new foo yields 91, regardless of the definition of m in Sup. The code is immune to capture of lexically scoped names due to changes in inherited libraries. We believe this approach is more robust in the face of program evolution, especially on larger scales.

4 Conclusions and Contributions

We argue that the classic "comb" semantics whereby inherited members may obscure lexically visible ones are counterintuitive. Raising this issue is itself a contribution of this paper. Beyond that, we have demonstrated a solution in the context of Newspeak, a language derived from Smalltalk.

The combination of nested classes and inheritance in a dynamic language is a relatively unexplored topic. While Newspeak and its features are not the focus of this paper, it should be clear that the aforementioned combination has enormous potential: in Newspeak, it provides not only nested classes, but also virtual classes, mixins, class hierarchy inheritance and a module system. We expect to explore this potential further in future work.

Acknowledgements

Peter von der Ahé suggested the Outer2 example. Eliot Miranda proposed the alternative semantics where inherited methods always require an explicit receiver. I am indebted to them both for these suggestions and for useful discussions. I also wish to thank the anonymous referees for their very helpful comments.

References

[BG05] Joshua Bloch and Neal Gafter. *Java Puzzlers: Traps, Pitfalls, and Corner Cases*. Addison-Wesley, 2005.

[BHJL86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, pages 78–86, November 1986. Published as Proceedings OOPSLA '86, ACM SIGPLAN Notices, volume 21, number 11.

[Ern01] Erik Ernst. Family polymorphism. In *European Conference on Object-Oriented Programming*, pages 303–326, 2001.

[FF98] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming*, pages 94–104, 1998.

[Fri] D. P. Friedman. Object-oriented style. Invited talk at the International LISP Conference, October 2003. Available at http://www.cs.indiana.edu/hyplan/dfried/ooo.pdf.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, Reading, Massachusetts, 2005.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its Implementation*. Addison-Wesley, 1983.

[MMP89] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, pages 397–406, October 1989. Published as Proceedings OOPSLA '89, ACM SIGPLAN Notices, volume 24, number 10.

[MMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993.

[Ode07] Martin Odersky. Scala by example, January 2007.

[SD03] Matthew Smith and Sophia Drossopoulou. Inner Classes visit Aliasing. In *ECOOP Workshop on Formal Techniques for Java Programs (FTfJP 2003)*, 2003.

[Smi] Walter R. Smith. *NewtonScript: Prototypes on the Palm*, pages 109 – 139. In Prototype-Based Programming: Concepts, Languages and Applications, Noble, Taivalsaari and Moore, editors, Springer-Verlag 1999.

[US87] David Ungar and Randall Smith. SELF: The power of simplicity. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, October 1987.