

# Call by Declaration

Erik Ernst<sup>1</sup>

Dept. of Computer Science, University of Aarhus, Denmark  
eerst@daimi.au.dk

**Abstract.** With traditional inheritance mechanisms, a subclass is able to use inherited features from its superclasses simply by using the names that these features are declared to have. Mixin based inheritance is more flexible, because a mixin can be applied to many different superclasses, whereas a traditional subclass is statically coupled with one fixed set of superclasses. However, mixins still depend on the exact choice of names in the superclasses to which they are applied. This paper presents a new information transfer mechanism, *call by declaration*, that arises naturally in the process of liberating mixins from the precise choice of names of declarations in their superclasses. As an extra benefit, this information transfer mechanism provides an explicit specification of the inherited features, easing separate type checking and code generation for mixins. We use the name ‘call by declaration’ for this mechanism because it is closely related to such mechanisms as call by value and call by reference, as known from procedure or method invocation.

## 1 Introduction

This paper investigates the mechanism that makes it possible for a subclass (or mixin) to inherit features from its superclasses. It considers traditional inheritance, in Sec. 2 and—via mixin classes in Sec. 3—arrives at genuine mixins in Sec. 4, and extends them to mixins with explicit superclass requirements in Sec. 4.1. In all these cases the approach is based on shared name spaces, i.e., the subclass or mixin contains a usage of a name that is (expected to be) declared in a superclass. In order to make this connection more flexible we take a small excursion, in Sec. 4.2, into the domain of procedure calls and method invocations. Using the inspiration from here, we continue to introduce a new information transfer mechanism, ‘call by declaration’, in Sec. 4.3, and show how it relaxes the coupling between the name spaces. Section 4.4 discusses the implementation, and Sec. 5 motivates the choice of terminology. Finally, Sec. 6 concludes.

## 2 Traditional Inheritance

Taking Java [1] as a typical example of a traditional inheritance mechanism, consider the following classes:<sup>1</sup>

```
class Point {
  int x,y;
  void print() {
    System.out.println("(" + x + "," + y + ")");
  }
}
class ColorPoint extends Point {
  String color;
  void print() {
    System.out.println("(" + x + "," + y + "," + color + ")");
  }
}
```

Ex.  
1

We are allowed to use the names `x` and `y` in the class `ColorPoint` because that class inherits declarations of the names `x` and `y` from the superclass `Point`. Moreover, the effect of declaring the method `print` in `ColorPoint` is to redefine `print`, i.e., to make the new declaration specify the implementation of `print` in an instance of `ColorPoint`, no matter whether the object is statically known as a `Point` or as a `ColorPoint`. Hence, the traditional inheritance mechanism lets a subclass use a name space provided by superclasses in several ways: Instance variables may be evaluated and/or assigned, and methods may be called and/or overridden. Other

<sup>1</sup> For conciseness, we have left out access control keywords such as `public`—all features in this paper may be considered public.

languages may support additional operations, e.g., in C++ it is possible to take the address of an instance variable. In general, anything that we can do to a locally declared entity can also be done to an inherited entity, and an inherited method also allows overriding.

With traditional inheritance, the superclasses are statically known. This makes it easy to check the usage of inherited features, whether it is for evaluation, assignment, invocation, or overriding. In a language like Smalltalk [9], it is possible to check that the current class does indeed inherit an instance variable with a given name, and to compute such things as the offset of that instance variable, to be used during code generation. In a language like Java, it is possible to look up the type in the declaration of an inherited name, and to check that the usage of the name is appropriate for its type. These kinds of checks are not so easy to perform with mixin based inheritance, as we shall see.

### 3 Mixin Classes

The notion of a *mixin class* originated in Flavors [5] and was soon after used in other LISP variants, e.g., CLOS [2]. It denotes an ordinary class that is by convention used in a special manner, namely as one of several superclasses used together; it should not be used alone. The idea is that the mixin class adds certain facilities to some of its fellow superclasses, and it may use facilities of those fellow superclasses to do so. Hence, a mixin class may use features not available in the class itself (neither locally declared nor inherited), because these features are expected to be provided by fellow superclasses. It is possible to write a mixin class in Flavors and in CLOS because the LISP family of languages is not statically type checked, but it is also possible to use a mixin class in a context that causes run-time errors, because the mixin class refers to a feature that is expected but not actually provided by its fellow superclasses. In order to be able to use the mixin style of programming in statically type languages, the mixin concept had to be further developed.

### 4 Mixins

Starting with [4] a concept of *mixins* distinct from classes was formed, and it has been further developed and refined in [8, 3, 10, 7] and elsewhere. A mixin is a building block for classes. A mixin  $M$  can be applied to a class  $C$ , thereby producing a subclass  $C'$  of  $C$ . With a suitable interpretation of classes and  $\oplus$ , this could be formalized as  $C' = C \oplus M$ . In [8], mixins are formalized as functions from classes to classes, but there is no conflict because the function would simply be  $\lambda C. C \oplus M$ .

A mixin such as  $M$  can be reused with several classes. E.g.,  $M$  may be applied to  $D$ , producing a subclass  $D'$ . Using traditional inheritance, we would need two identical copies of the text corresponding to  $M$ , in order to create  $C'$  from  $C$  as well as  $D'$  from  $D$ .

The most primitive kind of mixin is the kind that is supported in C++[12] by means of templates:

```
template < class Super >
class ColorMixin : public Super {
    String color;
    void print() {
        cout << "(" << x << ", " << y
            << ", " << color << ")" << endl;
    }
}
```

Ex.  
2

Because the template class ColorMixin inherits from the template argument Super, it can be used as a mixin. I.e., it can be applied to a superclass and thereby produce a new subclass:

```
class Point {
    int x,y;
    virtual void print {
        cout << "(" << x << ", " << y << ")" << endl;
    }
};

class ColorPoint : public ColorMixin< Point > {};
```

Ex.  
3

With respect to the dependencies on the chosen names in the superclass, this does not differ significantly from other mixin mechanisms. However, `ColorMixin` cannot be type checked separately, and code cannot be generated for it, because its declaration uses names such as `x` and `print` whose meaning is only known when the template is instantiated and the argument `Super` is replaced by a concrete class. We could say that mixin application is unencapsulated in the sense that every detail of the implementation of `ColorMixin` must be known in order to determine whether or not it can be applied as a mixin to a given potential superclass.

#### 4.1 Mixins with Explicit Requirements

A much better approach is taken in `MIXEDJAVA` [8], where the requirements by the mixin on the superclass is specified explicitly, by means of a so-called inheritance interface. We adjust the syntax here for notation homogeneity, and it then appears as follows:

```
class Point {
    int x,y;
    void print() {
        System.out.println(",x,",",y,");
    }
}
mixin ColorMixin requires { int x,y; void print(); } {
    String color;
    void print() {
        System.out.println(",x,",",y,",",color,");
    }
}
class ColorPoint = Point  $\oplus$  ColorMixin
```

Ex.  
4

In fact, inheritance interfaces in `MIXEDJAVA` do not support the specification of instance variables—such as `x` and `y`—but the main point here is that the mixin’s requirements are specified explicitly. The details of that idea may then be designed in many different ways. If we choose to restrict inheritance interfaces to be ordinary interfaces in Java (as in `MIXEDJAVA`) then mixin based inheritance will afford us access to inherited methods, but not to inherited instance variables. This is probably too restrictive.

If a language with mixins relies on some kind of inheritance interfaces to declare the exact requirements of mixins on their superclasses, then it is certainly possible to check the correctness of a mixin separately, and it is probably possible to generate code for it separately. We could not expect, though, to know such things as the exact offset of `x` and `y` in the resulting object layout, so the generated code may be less efficient than with traditional inheritance. However, the mixin is still rigidly dependent on the choice of names in superclasses. If an otherwise perfectly suitable superclass uses the name `write` where `ColorMixin` in Ex. 4 expects `print`, it is impossible to compose the two as intended.

#### 4.2 Considering Ordinary Invocation

There are in fact some very well-known mechanisms capable of providing bindings in a name space, independently of the choice of names at the site where the binding takes place. This happens at every procedure invocation, method call, etc. At the call site, the caller specifies the binding of each name in a name space, namely the formal argument list, to a value (with call by value) or to a possibly mutable entity (with call by reference). It makes no difference whether the invocation syntax uses a comma separated list of anonymous expressions or it uses keyword arguments—the caller is free to use an expression for an argument if it has the right properties (e.g., the right type), no matter what names the expression contains. For example (in C++):

```
int callee(int x, int &y) {
    return x+y;
}

void caller() {
    int a=2,b=3;
    a = callee(a,b);
    b = callee(5,a);
}
```

Ex.  
5

In the two invocations of `callee`, `a` is used to set up both `x` and `y` in the name space used for an execution of the body of `callee`. Since we are using call by value for `x`, it is even possible to use an rvalue (here 5).

This is of course trivial, except that there is no similar facility connecting classes with mixins. If we can allow a mixin application operation to explicitly set up the connection between the superclass name space and the mixin name space then it will be possible to compose classes from mixins in more flexible ways, thereby increasing the reuse potential of mixins, allowing conceptually sound combinations that would otherwise have been prevented by accidental name differences.

### 4.3 Call by Declaration

If we try to use conventional parameter passing mechanisms to establish the required name space in a mixin then we quickly discover that it does not suffice:

```
mixin ColorMixin(int &a, int &b, ?) {
    String color;
    void write() {
        System.out.println(",a,",",b,",",color,"");
    }
}
```

Ex.  
6

Assume for a moment that we can specify call by reference semantics by adding ‘&’ to argument names, as we have done in Ex. 6. This solves the problem for instance variables, but it is not obvious how to handle the method `write`. The problem is that we do not just want to be able to *invoke* that method, we want to be able to *redefine* it. The fact that there is a problem has been indicated by the ‘?’ in the list of formal arguments. To clarify the semantics of the above notation before proceeding with the solution to the method redefinition problem, we will ignore `write` for now and return to it later.

The mixin `ColorMixin` in Ex. 6 can be applied to a superclass as follows:

```
class Point {
    int x,y;
    void print() {
        System.out.println(",x,",",y,"");
    }
}
class ColorPoint = Point (x,y,?) ColorMixin
```

Ex.  
7

Again, we have indicated that there is a problem with `write` by putting a ‘?’ in the list of actual arguments. As opposed to method invocations and procedure calls, this kind of “invocation” (which is really a mixin application) receives its arguments *between* the denotation of the superclass (`Point`) and the denotation of the mixin (`ColorMixin`). This makes the actual argument list appear similar to a specialized operator with two operands, and that is indeed a useful point of view. Note that the names in the actual argument list are looked up in context of the name space defined inside the left hand operand `Point`; with ordinary invocation the names used in actual argument expressions are looked up in the scope that surrounds the invocation as a whole. Furthermore, the names are used to initialize a name space in the right hand operand `ColorMixin`, in contrast with ordinary invocation where the holder and user of the constructed name space (the procedure or method being called) is to the left of the argument list.

The semantics of this application is that `ColorPoint` is a class having the features of `Point` and the features of `ColorMixin`; every usage of `a` in the body of `ColorMixin` is a usage of a reference that is bound to `x`, and similarly for `b` and `y`. This is enough (apart from the unsolved problem with `write`) to make an instance of `ColorPoint` in Ex. 7 work like an instance of `ColorPoint` in Ex. 4 where a traditional mixin is used. One difference is that it will also allow access to `x` under the name `a` and similarly for `y` and `b`. It may be appropriate to make the names `a` and `b` invisible everywhere outside `ColorMixin`, but there may also be cases where the names declared in mixins are very useful in their own right. We do not think that a single, enforced policy in this area will be the right choice.

To handle the method `write`, we need a new transfer mechanism, namely *call by declaration*. A formal argument to be received by call by declaration must be specified by writing a declaration. The declared name is then available in the mixin body as if it had been declared locally. The formal argument is bound to an actual argument—an identifier—at mixin application time, and the effect of the binding is that all usages of the formal

argument in the mixin work as if they had been usages of the actual argument. The effect is similar to renaming in Eiffel [11], except that it is applied at mixin application time, independently of the superclass as well as the mixin. Using call by declaration, we can adjust the mixin and application as follows:

```
mixin ColorMixin(int a; int b; void write();) {
    String color;
    void write() {
        System.out.println(",a,",",b,",",color,");
    }
}
class ColorPoint = Point (x,y,print) ColorMixin
```

Ex.  
8

The contents of `ColorMixin` is the same as in Ex. 6, only the formal argument list has changed. The assumption is now that the formal argument list for a mixin will specify all arguments to have call by declaration semantics, which is why there is no separate indication of argument passing mode. The syntax used to declare formal arguments is similar to the declaration syntax of the underlying language, in particular the interface or specification oriented part thereof. In Ex. 8 we declare two instance variables `a` and `b`, and a method `write`.

Call by declaration semantics is the same as call by reference semantics for instance variables; i.e., a mixin argument that declares an instance variable is just an alias for an instance variable in the superclass. For a method, a call by reference argument provides access to call that method under the name given in the mixin; e.g., in `ColorPoint`, code inside `ColorMixin` may call `print` under the name `write`. Moreover, `ColorMixin` may redefine `print` to `write`, such that invocations of `print` on an instance of `ColorPoint` statically known as a `Point` or a `ColorPoint` will invoke `write`. Finally, we may or may not be able to invoke `write` on an instance of `ColorPoint` using the name `write`—the considerations are similar to the ones presented above in connection with instance variables.

It may be valuable to support multiple modes for mixin arguments. Call by declaration would definitely be needed for a method that the mixin wants to redefine, and call by declaration would allow a mixin to use an instance variable in all the ways that a locally declared instance variable could be used. But call by value would be a useful inheritance mode for a method that the mixin only needs to call, not redefine. If a class declares a certain method to be `final` then it would be compatible with a mixin that received this method by value, but not by declaration. There is a connection to the requirement of using explicit `redefines` clauses in Eiffel, in that the subclass declares how it will use inherited entities. Thus, programmers could use multiple information transfer mechanisms to fine-tune the encapsulation between the mixin and its superclasses, and to allow mixins to declare how *limited* their use of certain features will be.

#### 4.4 Implementation

We have not yet implemented mixins with call by declaration, but it is still possible to say something about the difficulty in doing so. In this section we have to assume a main-stream language implemented with main-stream techniques. If the language is very different, as is the case with for example `gbeta` [6], the implementation difficulties will also be entirely different.

With instance variables, it seems to be easy to support call by declaration. We just need to reserve space in the mixin for an offset to the slot where the instance variable is stored, and this offset should be initialized at mixin application time, according to the corresponding actual argument. For each mixin application we should also check that the declaration whose name is given as the actual argument does in fact declare an instance variable with an appropriate type. The simplest approach would be to require the exact same type, and in the general case it is not sound to allow a proper supertype nor a proper subtype of the formal argument declaration to be the actual argument declaration.

For a method  $m$  specified as a call by declaration formal argument, we need to be able to call it and possibly to redefine it. To call  $m$ , we must store a little bit of information in the mixin; in many languages (assuming a “typical” implementation) this could be in the form of an index into the virtual table of the class. Invocation of  $m$  would then cost one more memory access than a (non-final) ordinary method invocation. Invocation of  $m'$  would cost exactly the same as invocation of an ordinary method. This virtual table index would be initialized at mixin application time.

The simplest possible requirements on the actual argument  $m'$  corresponding to  $m$  is again that they have exactly the same type, i.e., same number and types of arguments, and same result type. Since we can both redefine and call a method, it would not be sound to allow a proper subtype or supertype as the actual return type of the method, nor as the actual type of an argument.

To impose a redefinition, the virtual table index for the actual argument  $m'$  in the superclass is needed. The goal is to make the declaration of  $m$  in the mixin redefine the declaration of  $m'$  in the superclass. The value in the virtual table for the class at this index is the address of  $m'$  in the superclass, and it should be the address of  $m$  when the mixin is present, so we need to put the address of  $m$  in that position in the virtual table. Note that since we are creating a new class we must also create a new virtual table with the abovementioned modification, and if classes can be created dynamically then we may need to take special precautions to avoid creating many identical copies of a virtual table and other class representation objects.

These considerations are based on a very simple and typical language implementation, but we hope that the discussion has at least given the impression that call by declaration may not be so hard to implement. Note, however, that the implementation considerations do *not* assume that mixin application must take place statically. If mixins can be applied dynamically, via denotations of classes and/or mixins which are variable at run-time, but typed, then the above implementation strategies would still work.

## 5 On the Terminology

‘Call by declaration’ is presented here as a renaming mechanism that is used in connection with mixin application. It would then seem natural to use a phrase like ‘mixin application renaming’ for it, and that phrase or a similar one might indeed be a good choice. However, we have chosen to use the phrase ‘call by declaration’ in order to emphasize the idea of improving inheritance by means of information transfer mechanisms developed in connection with behavior (procedures, methods, functions, etc.). I.e., we want to emphasize the source of the inspiration for the mechanism.

In return, the behavior information transfer mechanisms have now been enriched with a new variant, namely call by declaration, and that mechanism might be used to enhance the expressive power of methods and their brethren. In particular, it would be possible to redefine local methods in a caller method in a language (such as BETA or gbeta) that supports nesting of methods within methods. This would give entirely new possibilities, and we plan to explore this idea in the future.

## 6 Conclusion

We have presented a development from traditional inheritance via mixin classes and mixins into parameterized mixins. The parameter passing mechanism used with parameterized mixins is *call by declaration*, and this mechanism enables the same kinds of collaboration between a mixin (or subclass) and a superclass as is known from traditional inheritance. It is well-known that mixins provide more flexible conceptual modeling, and additional reuse opportunities, compared to traditional subclassing mechanisms. But traditional mixins are inflexible with respect to the choice of names in the superclasses to which they are applied. If a class has the right structure but just one “wrong” name then the mixin cannot be applied. Parameterization liberates mixins from this rigid dependency on names. The parameterized mixin uses names declared as formal arguments, and these names are bound to actual declarations as part of the mixin application operation. For instance variables, the call by declaration mechanism is not very hard to understand or implement—it works similarly to call by reference in connection with traditional procedure calls. But for methods, call by declaration has novel properties, in that it allows the mixin (i.e., the “callee”) to change the meaning of a declaration (i.e., the actual argument) in the superclass (the “caller”). Nevertheless, it seems to be relatively straightforward to introduce call by declaration also for methods into an implementation of a language that already supports mixins.

## References

1. Ken Arnold and James Gosling. *The Java™ Programming Language*. The Java™ Series. Addison-Wesley, Reading, MA, USA, 1998.
2. B. Bobrow, D. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. *Common Lisp Object System Specification*. Document 88-002R. X3J13, June 1988.
3. Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In Rachid Guerraoui, editor, *ECOOP '99 — Object-Oriented Programming 13th European Conference, Lisbon Portugal*, volume 1628 of *Lecture Notes in Computer Science*, pages 43–66. Springer-Verlag, New York, NY, June 1999.
4. Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices*, volume 25, 10, pages 303–311, October 1990.
5. Howard I. Cannon. *Flavors: A non-hierarchical approach to object-oriented programming*. Symbolics Inc., 1982.

6. Erik Ernst. *gbeta – A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, DEVISE, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 1999.
7. Erik Ernst. Propagating class and method combination. In Rachid Guerraoui, editor, *Proceedings ECOOP'99*, LNCS 1628, pages 67–91, Lisboa, Portugal, June 1999. Springer-Verlag.
8. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, San Diego, California, 19–21 January 1998.
9. Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley, Reading, MA, USA, 1989.
10. Carine Lucas and Patrick Steyaert. Modular Inheritance of Objects Through Mixin-Methods. In Peter Schulthess, editor, *Advances in Modular Languages*, pages 273–282. Universitätsverlag Ulm GmbH, 1994. Proceedings of the Joint Modular Languages Conference, University of Ulm, Germany, 28-30 September 1994.
11. Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, N.Y., second edition, 1997.
12. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.