Traits as Non-Objects in Grace

Andrew P. Black

March 17, 2013

Abstract

Here is my attempt to explain how traits as templates will work in Grace. I know that James and Michael have already written other versions; I'm deliberately starting afresh, to keep my critical thinking paths open. I suspect that this way, we will collectively come to consider a wider range of issues.

1 Introduction

Rather than having multiple meanings for the same syntax, with the interpretation depending on context, this proposal adds a new construct to Grace, which I'm calling a trait and thinking of *not* as an object, but instead as an object template. The idea is that *instantiating* an object template creates an object.

2 Object Constructors

2.1 Syntax

The existing object constructor syntax continues, with (I think) unchanged semantics, but with a new grammar and interpretation.

```
objectConstructor ::=  object trait
trait             ::=  traitLiteral |
                       namedTraitWithArgs
traitLiteral      ::=  { inheritsClause? (   methodDeclaration |
                                             variableDeclaration |
                                             constantDeclaration |
                                             typeDeclaration |
                                             traitDeclaration |
                                             statement )*
                       }
inheritsClause    ::=  inherits namedTraitWithArgs
```

Thus, the stuff in braces that follows the keyword **object** in the current syntax is re-interpreted as a trait, and inheritance (when used) is always from a trait. In addition, traits can be named and reused:

traitDeclaration	::=	**trait** identifier typeParameterList objectParameterList
		= traitLiteral
namedTrait	::=	identifier —*from a traitDeclaration*
namedTraitWithArgs	::=	namedTrait (< typeParameterList >)$^?$ ($\underline{(}$ parameterList $\underline{)}$)$^?$

The grammar rule for namedTrait is intended to hint at what cannot be captured in a context-free grammar: that a namedTrait is an identifier that has appeared on the right-hand side of a traitDeclaration.

2.2 Semantics

A traitLiteral is a template for an object: when used in an objectConstructor, we say that the trait is instantiated. Instantiating a trait causes an object to be created; that object will have the methods, types and traits defined in the traitLiteral (including any methods defined by **is** public or **is** confidential annotations on variables and constants). We haven't specified the semantics of anything formally, so I'll merely point out what is strange about trait literals.

We can't treat traits as macros, that is, we can't define their meaning by textual substitution. But neither are traits objects: they cannot be manipulated at runtime, and thus they don't have types. The reasons that we can't define them by textual substitution is because they have strange binding rules:

1. The keyword **self** appearing in a traitLiteral means the object that will eventually be created when the traitLiteral is instantiated, either directly (as a named-TraitWithArgs or indirectly, e.g., in an inheritsClause. Specifically, **self** does *not* refer to the enclosing object.

2. References to variables, constants, types and traits defined in the traitLiteral itself refer to the variables, constants, types and traits that will be in the object that is created by instantiating the trait. It may be best to think of these declarations, when appearing in a trait, as templates for actual declarations that will appear in th instantiated object.

3. References to variables, constants, types and traits that are defined in the lexical scope surrounding a trait are bound those lexically-surrounding variables, constants, types and traits. Note that, in the case of a named trait, these entities may *not* be in scope when the trait is instantiated.

Statements included in a trait are executed when that trait is instantiated, with the names used in the statement being interpreted according to the above binding rules. The same is true for expressions appearing in the right-hand side of variable and constant definitions. For these reasons, we may still choose to eliminate the use of **self** in such statements and expressions.

3 Trait Parameters

A namedTrait can have one or more parameters. When a namedTrait with parameters is instantiated, the parameters are replaced by the values of the arguments, which must be provided. Arguments must be objects (not types, or traits).

A namedTrait can also have one or more type parameters. When a namedTrait with type parameters is instantiated, the type parameters are replaced by the type arguments, which must be provided, and which must be types (not objects, or traits).

We should probably allow multi-part names for traits, and allow the parameters and arguments to be distributed through the parts, as with method names. The above syntax doesn't show this, because I was being lazy.

4 Trait Combinators

Inheritance is now a trait combinator: it is an operation on two traits, one being a named trait and the other a trait literal. It does what it has always done, allowing overriding of methods, but binding variables, constants, types and traits lexically, and thus keeping them private to the trait that defines them. It might be more consistent to change the syntax to make inheritance a binary combinator binary, so that either or both of the arguments can be named traits or trait literals. This would mean moving the inherits clause outside the braces, and putting the parent trait second, or changing the keyword. That is,

```
{
      inherits base
      var x
      method foo {...}
}
```

might become

```
{
      var x
      method foo {...}
}
extends base
```

or

```
base extendedBy {
      var x
      method foo {...}
}
```

We can define the trait summation operation +, the method removal operation − and the aliasing operation @{_ −> _}. I suggest that these facilities be available only in the more advanced dialects of Grace.

We can also define a *deep rename* operation that has been missing from most (all?) previous descriptions of traits. The reason that we can provide deep rename is because Grace traits contain *descriptions* of methods, rather than methods. Deep rename differs from alias in the way that recursive method definitions are changed when a composite trait overrides a name used in such a method. An alias ensures that the overridden method is still available, under the alias; however, what were recursive requests of the same method become requests to the overriding method. (So, aliasing subsumes super-requests.) Deep rename ensures that the overridden method is still available — indeed, it is not actually overridden at all, because it has been renamed! So recursive requests remain recursive requests.

5 Questions

Now I'm going to take a look at James' questions, and see how the above definition answers them.

Q1: Why call them "traits". Why not "classes" ? The term "class" is traditionally used for two different things: a constructor of objects, and a template that one can inherit and instantiate. I'm using "class" for the first, and "trait" for the second. Using different terms for different things is good pedagogical practice. Traits are always "abstract", in the sense of "abstract superclass": they are fragments of reusable behavior, and will often be incomplete. A class must be complete: we can statically check, for example, that there is a method corresponding to every self request made in a class. This check can be made whenever the programmer instantiates a trait.

Q2: How do you extend dialects or modules or true? All of those things re objects, and can't be inherited from. Instead, our dialects, modules and definitions of primitive objects will need to take pains to put most methods in traits, specifically to allow reuse. As James notes, "utility of modules and dialects as *objects* is somewhat diminished by this design". For "somewhat", I would say "significantly".

Q3: Are traits objects? No, not as envisaged here. I believe that allowing them to be objects will introduce a great deal of extra complexity. If we did make traits objects, then we could conceivably provide a meta-operation on any object that retrieves the trait that was instantiated to produce it. Note that implementing this operation would involve something akin to Michael's proposal to keep two forms of every class around — one corresponding to the class, and the other corresponding to the class generator. This would do something to restore the utility of modules as objects. SuccessfulMatch could then inherit from true.**trait**, for example.

Q4: What requests do those trait objects understand? What's a trait expression? Whether or not they are objects, traits understand the inheritance, summation, method removal, aliasing and, I propose, the deep method rename operations described above. we should rationalize the inheritance operation, as discussed in Section 4 above. I'm thinking of traits as language primitives, not objects, and these meta operations as being static, not dynamic, but I'm not sure that most novice programmers will care.

James: I'm looking forwards to Andrew giving us some definitions that are legal ASCII and legal Grace.

For the other operations, I see no reason not to use + for trait sum, − for method removal, and @ for alias. Deep method rename could be traitname[new/old].

Q5: What about abstract, and value, and all that stuff from yesterday? For simplicity, I would not bother with abstract methods — I would just omit them. It's the job of the programming environment to tell the programmer what methods need to be added to make a trait complete. Many traits would be abstract, i.e., they would have "missing" methods. As mentioned above

Value objects are important, but, I think, quite independent of the mechanisms described here.

Q6: How do you pass a trait around as a parameter? You can't. Traits are not objects. If you want to do static checking on traits, then they can't be parameters. This is effectively the same restriction that we had before, which said that the expression after **inherits** had to be both fresh and definitively static. It could not be a parameter. In the past, it could not even be an object named in a **def**. Now, of course, it can be a names trait, which is much more reasonable.

Q7: How do you make a class with multiple constructors? James didn't actually ask the question this way, but I think that he meant to, based on his answer.

```
def catFactory = object {
    method cat(name) {
        object likeaCat(name) }
    method mimi { cat("mimi") }
    method fergus { cat("Fergus") }
}
```

As James said, this is the way that we've always done it.

I think that the more interesting question is how do you make a class with a single constructor. We used to have a special syntax for this — the class syntax.

```
class aCat.named(n) {
    method name { n }
    method speak { print "meow" }
}
```

Now we would have to write:

```
def aCat = object {
    method named(n) {
        object {
            method name { n }
            method speak { print "meow" }
        }
    }
}
```

Should we retain — or perhaps, better, retrain — the class syntax? I think that perhaps we should, and that it should do two things: (1) declare a trait, and (2) declare a factory object. In the above example, the class syntax would be equivalent to:

```
trait likeaCat(n) = {
        method name { n }
        method speak { print "meow" }
}
def aCat = object {
        method named(n) {
                object catable(n)
        }
}
```

The point of this is that someone wishing to inherit the methods of a cat can inherit from liekaCat. Generating the name automatically might be difficult — I considered catTrait, catable and catlike.

Q8: How can we implement this?

We have a dictionary of named object templates available at compile time, and we do essentially what Java does, except for delaying completeness-checking until trait-instantiation time.

Q9: You mean traits are really just factory methods?

No, not at all. Traits are static. They aren't method, they are not inside any object, and they can't be requested. They are syntax.

Q10: What about optimisation? Making things static so we can understand programs?

Yes, you've got it! Traits are static. We can optimize them, and we can understand programs.

Q*: Other questions? James seems to go off into meta-space with traits as objects. This might be worth exploring as a research topic. It seems to me that it will give us all of the dynamics that James and Kim didn't like about my earlier proposals. Good research fodder. Bad novice langauge.