# Object inheritance with first-class classes: Better JavaScript-style classes in Grace

Timothy Jones and Michael Homer

Once it was clear that the magic of class constructors behaving differently when used in an inherits clause wasn't going to cut it, I spent some time thinking about how we could unify the definition of classes in a way that gave them a relationship on the type level. Originally I defined a class as having a method 'inherit' which takes the value of self and returns a new class with the same constructor, but which just adds its definition to the self object. This led to unwieldy backwards code where the constructor came after the pass of self:

```
def car = object {
    inherits vehicleFactory.newWithWheels(4)
}
```

This code became:

```
def car = object {
    vehicleFactory.inherit(self).newWithWheels(4)
}
```

Furthermore, this was a rewrite that wouldn't work on general expressions (which defeated the whole purpose). In order to write the mechanism in the right way I was struck by the notion that in the case of inheritance we don't even want to create a new object, we just want to pass the parameters of the object constructor. *So why am I invoking a method with the word 'new' in it?* What if the two were separate? What if a class was an object with two methods: 'extend' and 'new'?

## 1 Classes as Objects

My major revelation was that the act of creating the *object* should be separate from producing its *definition* — and that creating the object naturally follows the definition. My issue with the current class definition (other than it didn't produce actual class inheritance) was that there wasn't any relationship between classes: there wasn't the notion of a class at the object structure (and therefore type) level. The definition of "an object with a

method that returns an object" was overly vague, and not every object that fulfils that definition is necessarily a class.

In the case above, we need to produce a definition of a vehicle with four wheels. First we produce this definition, and then we use it to create an object based on that definition:

```
def car = vehicleFactory.withWheels(4).new
```

And when you want to inherit from that definition, you don't reference 'new' at all:

```
def car = object {
    inherits vehicleFactory.withWheels(4)
}
```

The placement of the 'new' might seem a little odd in comparison to how other languages deal with object creation, but I would argue that it's now in the right place. Even cooler, we can save the definition produced from requesting 'withWheels', and create a bunch of vehicles with four wheels without requesting the method again!

```
def carFactory = vehicleFactory.withWheels(4)
def car1 = carFactory.new
def car2 = carFactory.new
```

But let's back up a bit first. What is a class?

## 1.1 Traits as Objects

It turns out that first we want to consider just a definition which is inheritable, rather than one that can necessarily create new definitions. A bit like an abstract class! But really it's a Trait, because it's really just a loose definition of an object that isn't a class (we're going to define class as something that explicitly constructs objects). So what's a trait? Well, it's just an object (as promised), but it implements the following type:

```
type Trait⟨T⟩ = {
    extend⟨U⟩(...) → U & T
}
```

We're going to leave out what the extend method takes for parameters for now, because we don't want to jump in the deep end just yet. But we can see from the type what's going on here. A Trait is a definition of an object of some type T, and can extends a definition of type U to produce an object of their combined type. All this information is available to the static type checker, because a trait is just a regular object and extend is a regular method. The inherits clause in an object definition will accept any object which implements this interface. This can be detected statically for values which have been given a type, and can be detected at runtime if the intended trait is not, in fact, a trait.

## 1.2 Classes as Trait + Factory

A class is a trait and a factory at the same time: an object definition that can be inherited from or have new instances constructed. So we want the type to be the addition of the Trait type and a type containing the method 'new'. It will also need a generic type to represent the underlying object definition, and it passes this along to Trait while also using it to specify the type of objects it constructs.

```
type Class⟨T⟩ = Trait⟨T⟩ & {
    new → T
}
```

As we would expect, a Grace class literal would produce an object of this type. However, any object can implement either Trait or Class and immediately be able to become part of the inheritance mechanism. As was recently proposed on top of the existing factory inheritance, we could define a method that allowed delegation to an object with a method delegate:

```
method delegate⟨T⟩(to : T) → Trait⟨T⟩ { . . . }

inherits delegate(someObject)
```

We provide a full definition of this method later.

We could cause every object to be inheritable by having a default implementation of the 'extend' method on the top object definition, but this would result in a situation where these two objects are both correct, and often appear to do the same thing, but have completely different definitions of 'self'.

```
object {
    inherits vehicleFactory.withWheels(4)
}

object {
    inherits vehicleFactory.withWheels(4).new
}
```

It seems like this is too easy to get wrong by accident, particularly for those new to programming. We merely point it out to show it is possible if desired, and if not, then the second definition will likely result in a static error (unless the developer is being awfully liberal with Dynamic, in which case it will be a runtime error).

Users who know what their doing are able to define their own inheritance mechanisms, and as we will see later users who don't need to know about this system are not exposed to it and are incredibly unlikely to stumble across it. If you wanted to be totally safe, you could just ban definitions of methods with the name 'extend' in their dialect.

## 2 Traits as Object Mutators

So, how does the trait extend the given object with its definition? We propose to solve this issues with open objects, which allow their structure to be mutated by meta-operations. The most important property of these open objects is that this structure mutation **cannot be observed**. We make this point now to avoid the reader balking at the idea of structure mutation — in order to gain access to the object underlying the meta-operations, the object must be 'sealed', which prevents any further modification.

Now we introduce the full type of Trait:

```
type Trait⟨T⟩ = {
    extend⟨U⟩(meta : Meta⟨U⟩) → U & T
}
```

An object of type Meta⟨U⟩ is a metaobject over an open object whose current structure implements the type U. Thus, it's the job of the extend method to add its definition to the open object so that it also implements T, and then seal and return the object.

### 2.1 The Inheritance Chain

Say we define the following classes and object.

```
class animal.makesNoise(noise : String) {
    method makeNoise is public {
        print(noise)
    }
}

class snake.hasName(name′ : String) {
    inherits animal.makesNoise("Hiss")

    def name is readable, public = name′
    print("{name} is making noise:")
    self.makeNoise
}

def emphaticSnake = object {
    inherits snake.hasName("Dave")

    method makeNoise is public, override {
        print("HISS!")
    }
    print("Here comes {self.name}!")
}
```

This is a good example of how the inheritance chain works, because it demonstrates how the object is built up as it proceeds through the chain, and the order of initialisation after the object is sealed. It also demonstrates *why* both upcalls and downcalls work as expected. This is our simple explanation of classes in terms of objects.

With our mutating traits, the object has method definitions added to it as it travels up the inheritance chain. As the object crosses the boundary from mutation to initialisation, the object is sealed. All of the method definitions are available on the way back down through initialisation code.

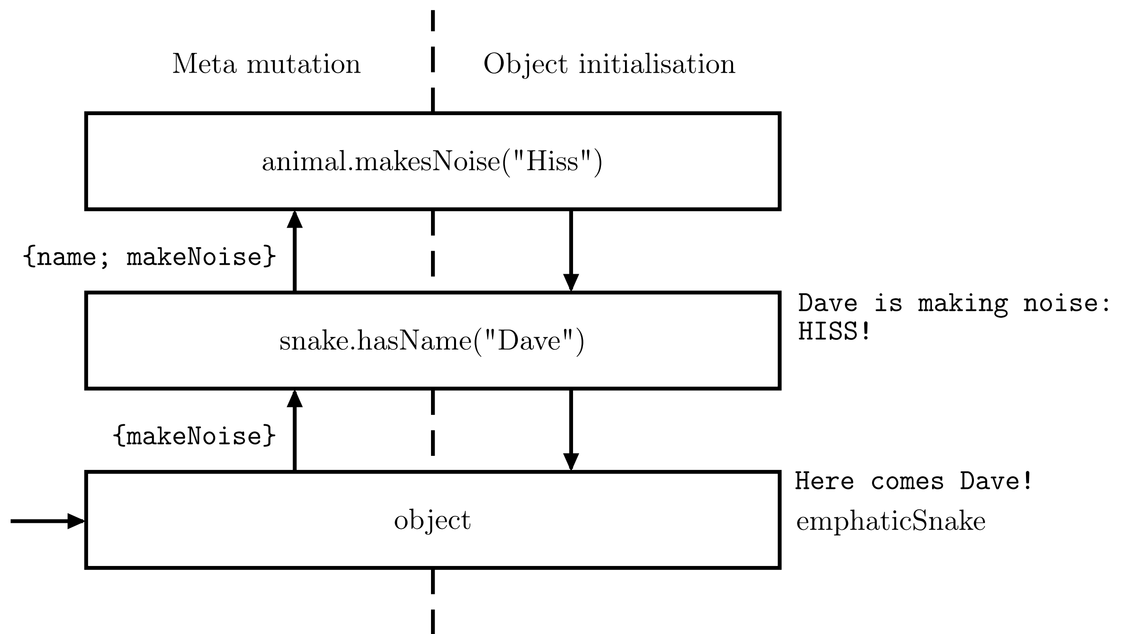


Figure 1: The execution path of inheritance

Note that the class boxes are labelled as the result of the method call on the class declaration: each is a specialised class with the string parameter already set in it.

## 2.2 Translating Objects

The object literal is still the core primitive of the language (read: not defined with a class), but it breaks down into a little jumble of meta-operations if you want to know its exact semantics. There's no need to actually translate it this way, it's just that this is equivalent to the way it works. In order to display this as code we need to represent first-class methods, and our translation uses a pseudo-code anonymous method for this syntax.

In order to demonstrate the translation, take the following object as an example:

```
def myObject = object {
    inherits myClass

    honk

    method honk is confidential {
        print("honk")
    }
}
```

Our translation follows these steps:

1. Request a new open object at the top of the block.
2. Yank the method definitions under this, above the inherits clause.
3. Add the remaining code into a special 'initalize' method.
4. Resolve to the inherits clause as a call to the extend method.

Here is an inscription in a mostly real version of Grace.

```
def meta = internal.newOpenObject
type Self = { honk → Done is confidential }

meta.addMethod("honk", method {
    print("honk")
}) annotations(confidential)

meta.addMethod("initialize", method {
    super.initialize
    self.honk
}) annotations(confidential)

def myObject = myClass.extend(meta.cast⟨Self⟩)
```

We emphasise that this is partly pseudo-code: anonymous methods are not real, and we have invented an interface for meta-operations. The syntax of adding methods and such is not intended to be final. We would like for these operations to be available to sufficiently advanced users through a meta dialect, however.

Every part of the chain adds a new initialize method to the object, and each method begins by invoking the one above it. When the object is sealed, initialize is called automatically. Whether or not it remains available to the object afterwards is merely a matter of design — it exists purely to ensure that constructor code is run in the right context of self, so that constructors may request confidential methods.

Note also the call to the cast method of the metaobject when it is passed up the inheritance chain. This adds the type definitions to the type of the metaobject itself, and cast just ensures that this definition exists. In the case of this translation, the cast can never fail.

A particularly important definition is the one at the top of the inheritance chain, which does not continue to extend the open object, but instead performs the seal and initialisation. We assume here that this is the Object Trait.

```
def top : Trait⟨Object⟩ = object {
    method extend⟨U⟩(meta : Meta⟨U⟩) → U & Object {
        // Top methods added here
        meta.addMethod("asString", method { ...
        meta.addMethod("initalize", method {})
        meta.cast⟨U & Object⟩.initializeObject
    }
}
```

The final line kills the metaobject and seals the real object, then runs the initialisation method on it before returning it.

## 2.3 Translating Classes

Classes translate much the same way as objects. Consider again our object example, this time as an anonymous class:

```
def myClass = class {
    inherits mySuperClass

    honk

    method honk is confidential {
        print("honk")
    }
}
```

This becomes:

```
def myClass = object {
    type BaseType = { honk → Done is confidential }
    type Type is public = BaseType & mySuperClass.Type

    method extend⟨U⟩(meta : Meta⟨U⟩) → U & Type {
        meta.addMethod("honk", method {
            print("honk")
        })

        meta.addMethod("initialize", method {
            super.initialize
            self.honk
        })

        mySuperClass.extend(meta.cast⟨U & BaseType⟩)
    }

    method new → Type {
        object { inherits outer }
    }
}
```

The key difference is that the extend method does not create the object. An application of the new method does this for us. As far as the user is concerned, the object appears to materialise at the top of the inheritance chain fully formed (and sealed), as you would expect from class inheritance.

We use an anonymous class as an example here, but we can equally translate the existing syntax on classes as well.

```
class vehicleFactory.withWheels(wheels : Number) → Vehicle {
    method wheelCount → Number is public { wheels }
}
```

This becomes:

```
def vehicleFactory = object {
    method withWheels(wheels : Number) → Class⟨Vehicle⟩ {
        class {
            method wheelCount → Number is public { wheels }
        }
    }
}
```

Which then translates the inner class as before.

We could even support the method class definition, to easily build factories with multiple constructors.

```
def vehicleFactory = object {
    class withWheels(wheels : Number) → RoadVehicle {
        method wheelCount → Number is public { wheels }
    }

    class withWings(wings : Number) → AirVehicle {
        method wingCount → Number is public { wings }
    }
}
```

The actual syntax is largely irrelevant to the actual problem, though.

### 2.4 Implementing Delegation

We can return to our example of delegation before, and provide an actual definition using the fudged meta-operations from before.

```
method delegate⟨T⟩(to : T) → Trait⟨T⟩ {
    def to′ = mirror(to)
    object {
        method extend⟨U⟩(meta : Meta⟨U⟩) → U & T {
            meta.onNoSuchMethod { name, args →
                to′.requestMethod(name) arguments(args)
            }

            meta.cast⟨U & T⟩.initializeObject
        }
    }
}
```

This shows that sufficiently advanced users are capable of writing their own inheritance mechanisms with this system.

There's also much more room for extensions to this system. We could define Mixins as a Trait that can produce a union of another Trait.

```
type Mixin⟨T⟩ = Trait⟨T⟩ & {
    union⟨U⟩(other : Trait⟨U⟩) → Mixin⟨T & U⟩
}
```

All of these first-class operations can be defined in the type system.

## 3 Conclusion

This implementation gives us all three of our longstanding criteria:

1. Class inheritance works as expected (downcalls)

2. We can inherit from arbitrary objects

3. We have a simple explanation of classes in terms of objects

1) and 3) have always been at odds, and most of the solutions to produce class inheritance have eschewed an object-only stance. The factory inheritance uses awful black magic — the position of a literal in the code! — to get around this. Our proposal avoids this, and defines everything nicely in terms of types.

The main point of contention, then, will be whether this is a 'simple' explanation of classes. The pseudo-code I translated into is not the prettiest, but I would argue that for the most part we do not need to explain the meta-operations (which are likely to be optimized away anyway): the notion of the object travelling up the inheritance chain, adding methods as it goes, and then proceeding back down the chain, executing constructor code, is rather eloquent, and an adequate explanation of what happens until someone is so hardcore that they want to actually write their own inheritance mechanism (which I imagine is quite hardcore).

Whether we want to differentiate the creation of classes from the creation of their instances with the extra call to 'new' may also be contentious. I see it not as a hack to solve this problem but an actual enhancement to the nature of the language, as it not only distinguishes their identities but has the nifty side-effect of allowing class specialisations from pre-calling the class method (as in the car factory example).