Pharo 9 by Example

Stéphane Ducasse and Gordana Rakic with Sebastijan Kaplar and Quentin Ducasse

October 18, 2021

Layout and typography based on the sbabook LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

Traits: reusable class fragments

Although Pharo offers single inheritance between classes, it supports a mechanism called Traits for sharing class fragments (behavior and state) across unrelated classes. Traits are collections of methods that can be reused by multiple classes that are not constrained in an inheritance relation. Since Pharo 7.0, traits can also hold state.

Using traits allows one to share code between different classes without duplicating code. This makes it easy for classes to *reuse* useful behavior across classes.

As we will show later, traits propose a way to compose and resolve conflicts in disciplined manner. With traits this is not the latest loaded method that wins as this happens with other languages. In Pharo, the composer (be it a class or a trait) takes always precedence and can decide in its context how to resolve a conflict: Methods can be removed or accessible under a new name at composition time.

1.1 A simple trait

The following code defines a trait. The `uses:` clause in an empty array indicating that this trait is not composed of other traits.

Listing 1-1 A simple trait.

```
Trait named: #TFlyingAbility
   uses: {}
   package: 'Traits-Example'
```

Traits can define methods. The trait `TFlyingAbility` defines a single method `fly`.

```
TFlyingAbility >> fly
  ^ 'I''m flying!'
```

Now we define a class called `Bird` that uses the trait. The class contains then the `fly` method.

```
Object subclass: #Bird
  uses: TFlyingAbility
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Traits-Example'
```

Instances of the class `Bird` can execute the message `fly`.

```
Bird new fly
>>> 'I''m flying!'
```

1.2 Using a required method

The trait methods do not have to define a complete behavior. A trait method can invoke methods that will be available on the class using it.

Here the method `greeting` of the trait `TGreetable` in invoking the method `name` that is not defined in the trait itself. In such a case the class using the trait will have to implement such a *required* method.

```
Trait named: #TGreetable
  uses: {}
  package: 'Traits-Example'
```

```
TGreetable >> greeting
  ^ 'Hello ', self name
```

Notice that `self` in a trait represents the receiver of the message. Nothing changes compared to classes and default methods.

```
Object subclass: #Person
  uses: TGreetable
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Traits-Example'
```

Now in the class `Person` we define the method `name` and the `greeting` method will invoke it.

```
Person >> name
  ^ 'Bob'
```

```
Person new greeting
>>> 'Hello Bob'
```

1.3 Self in a trait is the message receiver

The question of the status of `self` in a trait may be raised. However, there is no difference between `self` used in a method defined in a class or defined in a trait. `self` always represents the receiver. The fact that the method is defined in a class or a trait has no influence on `self`.

We define a little Trait to expose this: the method `whoAmI` just return `self`.

```
Trait named: #TInspector
  uses: {}
  package: 'Traits-Example'
```

```
TInspector >> whoAmI
  ^ self
```

```
Object subclass: #Foo
  uses: TInspector
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Traits-Example'
```

The following snippet shows that `self` is the receiver, even when returned from a trait method.

```
| foo |
foo := Foo new.
foo whoAmI == foo
>>> true
```

1.4 Trait state

Since Pharo 7.0 traits can also define instance variables. Here the trait `TCounting` defines an instance variable named `#count`.

```
Trait named: #TCounting
  instanceVariableNames: 'count'
  package: 'Traits-Example'
```

The trait can initialize its state by redefining the method `initialize` followed by the class named. Here the trait `TCounting` defines the method `initializeTCounting`.

```
TCounting >> initializeTCounting
  count := 0
```

```
TCounting >> increment
  count := count + 1.
  ^ count
```

The class `Counter` uses the trait `TCounting`: its instances will have an instance variable named `count`.

```
Object subclass: #Counter
  uses: TCounting
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Traits-Example'
```

To initialize correctly `Counter` instances, the `initialize` method of the class `Counter` should invoke the previously defined trait method `initializeTCounting`.

```
Counter >> initialize
  self initializeTCounting
```

The following code shows that we created a counter and it has a well initialized instance variable.

```
Counter new increment; increment
>>> 2
```

1.5 A class can use two traits

A class is not limited to use only one trait. It can use several traits. Imagine that we define another trait called `TSpeakingAbility`.

```
Trait named: #TSpeakingAbility
  uses: {}
  package: 'Traits-Example'
```

This trait defines a method named `speak`.

```
TSpeakingAbility >> speak
  ^ 'I''m speaking!'
```

Now we define a second trait `TFlyingAbility`.

```
Trait named: #TFlyingAbility
  instanceVariableNames: ''
  package: 'Traits-Example'
```

This trait defines a method `flying`.

```
TFlyingAbility >> fly
  ^ 'I''m flying'
```

Now the class `Duck` can use both traits `TFlyingAbility` and `TSpeakingAbility` as follows:

```
Object subclass: #Duck
  uses: TFlyingAbility + TSpeakingAbility
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Traits-Example'
```

Instances of the class Duck get all the behavior from the two traits.

```
| d |
d := Duck new.
d speak
>>> 'I''m speaking!'
d fly
>>> 'I''m flying!'
```

1.6 Overriding method takes precedence over trait methods

A method originating from a trait acts as if it would have been defined in the class using that trait. Now the user of a trait (be it a class or another trait) can always redefine the method originating from the trait and the redefinition takes precedence in the user over the method of trait.

Let us illustrate it. In the class Duck we can redefine the method speak to do something else and for example send the message quack.

```
Duck >> quack
  ^ 'QUACK'
```

```
Duck >> speak
  ^ self quack
```

It means that

- the trait method speak is not accessible from the class anymore, and
- the new method is used instead, even by methods using the message speak.

```
Duck new speak
>>> 'QUACK'
```

We define a new method call doubleSpeak as follows:

```
TSpeakingAbility >> doubleSpeak
  ^ 'I double: ', self speak, ' ', self speak
```

The following example shows that the locally redefined version of the method speak of the class Duck takes precedence over the one of the trait TSqueakingAbility.

```
Duck new doubleSpeak
>>> 'I double: QUACK QUACK'
```

1.7 Accessing overridden trait methods

Sometimes you want to override a method from a trait and at the same time still being able to access the overridden method. This is possible by creating an alias to that overridden method in the class trait composition clause using the @ and -> operators as follows:

```
Object subclass: #Duck
  uses: TFlyingAbility + TSpeakingAbility @{#originalSpeak -> #speak}
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Traits-Example'
```

Note that the arrow means that the new method is the same as the old one: it just has a new name. Here we say that `originalSpeak` is the new name of `speak`.

The overridden method can be accessed using its new name as any method, by sending a message using it. Here we define the method `differentSpeak` and it is sending the message `originalSpeak`.

```
Duck >> differentSpeak
  ^ self originalSpeak, ' ', self speak
```

```
Duck new differentSpeak
>>> 'I''m speaking! QUACK'
```

Pay attention that an alias is not a full method rename. Indeed if the overridden method is recursive, it will not call the new name, but the old one. An alias just gives a new name to an existing method, it does not change its definition: nothing in the method body is changed.

1.8 Handling conflict

It may happen that two traits used in the same class define the same method. This situation is a conflict. To solve such a conflict, there are two strategies:

1. Using the exclude operator (-), you can exclude the conflicting method from the trait defining the method. In this case the other method will be the one available in the class,
2. or redefine the conflicting method locally in the class. In such a case, the conflicting methods are overridden and the new redefined behavior is the one that will be available in the class. Note that overriddent methods can be made accessible as previously explained with the @ operator.

Here is an example. Let us define another trait named `THighFlyingAbility`.

```
Trait named: #THighFlyingAbility
  instanceVariableNames: ''
  package: 'Traits-Example'
```

This trait defines a `fly` method.

```
THighFlyingAbility >> fly
  ^ 'I''m flying high'
```

When we define the class `Eagle` that uses the two traits `THighFlyingAbility` and `TFlyingAbility`, we have a conflict when sending the message `fly` because the runtime does not know which method to execute.

```
Object subclass: #Eagle
  uses: THighFlyingAbility + TFlyingAbility
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Traits-Example'
```

```
Eagle new fly
>>> 'A class or trait does not properly resolve a conflict between
    multiple traits it uses.'
```

1.9 Conflict resolution: excluding a method

To solve the conflict, during the composition, we can simply exclude the `fly` method from the trait `TFlyingAbility` as follows:

```
Object subclass: #Eagle
  uses: THighFlyingAbility + (TFlyingAbility - #fly)
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Traits-Example'
```

Now there is only one `fly` method, the one from `THighFlyingAbility`.

```
Eagle new fly
>>> 'I''m flying high'
```

1.10 Conflict resolution: redefining the method

Another way to solve the conflict is to simply redefine the conflicting method in the class using the traits.

```
Object subclass: #Eagle
  uses: THighFlyingAbility + TFlyingAbility
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Traits-Example'
```

```
Eagle >> fly
  ^ 'Flying and flying high'
```

Now there is only one `fly` method: the one defined in the class `Eagle`.

```
Eagle new fly
>>> 'Flying and flying high'
```

You can also access the overridden methods by creating an alias associated with the trait as explained before.

1.11 Conclusion

Traits are groups of methods and state that can be reused in different classes, supporting this way a kind of multiple inheritance in the context of a single inheritance language. A class can be composed of several traits. And traits can define instance variables and methods. When the traits used in a class define method having the same name, this leads to a conflict.

To handle conflict, the class can redefine the class locally: local methods take precedence over trait ones, or exclude a conflicting method. Finally a overridden method (redefined in a class) can be accessed via an alias created at the level of the class composition.