

Multiple Inheritance using modules in Ruby

Amruta Gokhale

Computer Science and Engineering
IIT Bombay
{amruta}@cse.iitb.ac.in

November 29, 2004



Introduction

- Ruby is an interpreted scripting language for quick and easy object-oriented programming
- Important Features are :
 - Powerful string operations and regular expressions
 - Variable declarations are unnecessary
 - Everything is an object
 - Classes, Inheritance, Methods, etc.



What is multiple inheritance?

- Some Object Oriented programming languages like C++, allow multiple inheritance
- Multiple Inheritance: A sub-class may be derived from multiple parent classes. Thus a single subclass can have more than one superclass
- A real-world example of multiple inheritance : An alarm clock
Alarm clock belongs to the class of clocks and also the class of things with buzzers



Inheritance in Ruby

- True multiple inheritance purposely not implemented in Ruby
- A good alternative is use of modules which eliminate the need for multiple inheritance, providing a facility called a mixin
- Use of modules
 - Gives the basic functionality of multiple inheritance
 - Allows to represent class relationships with a simple tree structure and so simplifies the language implementation considerably



Modules

- Modules in ruby are similar to classes except:
 - A module can have no instances.
 - A module can have no subclasses.
 - A module is defined by *module ... end*
- On including a module in a class definition, all the module's instance methods are suddenly available as methods in the class as well. Thus modules' methods are effectively appended or “mixed in” to the class
- Mixed-in modules effectively behave as superclasses.



include statement

- Use *include* `<Module Name>` to include a module in a class
- A Ruby *include* does not simply copy the module's instance methods into the class
- Instead, it makes a reference from the class to the included module
- If multiple classes include that module, they will all point to the same thing



Example

- module GetName
 def getName
 end
end
- class Pay_Salary
 include GetName # ...
end
- class Student_Attendance
 include GetName # ...
end
- prof = Pay_Salary.new("Professor1")
 stud = Student_Attendance.new("Student1")
- By including the GetName module, both Pay_Salary and Student_Attendance gain access to the getName instance



Instance Variables in Mixins

- Access to the methods defined in the mixin
- Access to the necessary instance variables as well



Problem with Instance Variables

- It is possible that different mixins may use an instance variable with the same name and create a collision
- For Example –
- Module MajorScales def majorNum
 @numNotes = 7 if @numNotes.nil?
 @numNotes # Return 7
 end
end
- module PentatonicScales def pentaNum
 @numNotes = 5 if @numNotes.nil?
 @numNotes # Return 5?
 end
end



Example continued..

- ```
class ScaleDemo
 include MajorScales
 include PentatonicScales
 def initialize
 puts majorNum # Should be 7
 puts pentaNum # Should be 5
 end
end
```





- The output produced is :
  - 7
  - 7
- The two bits of code that we mix in both use an instance variable named `@numNotes`. Unfortunately, the result is probably not what is expected.

## *Mixin Modules*

- For the most part, mixin modules don't try to carry their own instance data around—they use accessors to retrieve data from the client object.
- If you need to create a mixin that has to have its own state, ensure that the instance variables have unique names to distinguish them from any other mixins in the system (perhaps by using the module's name as part of the variable name)



# Mixin Modules

- Thus Mixin can be thought of as a way of asking for whatever particular properties we want to have.
- For example, if a class has a working *each* method, mixing in the standard library's Enumerable module gives us *sort* and *find* methods for free.



## Enumerable Module

- If desired, your classes can support all the neat-o features of *Enumerable*, thanks to the magic of mixins and module *Enumerable*.
- All you have to do is write an iterator called *each*, which returns the elements of your collection in turn.
- Mix in *Enumerable*, and suddenly your class supports things such as *map*, *include?*, and *find\_all?*.
- If the objects in your collection implement meaningful ordering semantics using the  $\leq$  method, you'll also get *min*, *max*, and *sort*



# References

- <http://www.ruby-doc.org/>
- <http://kylecordes.com/files/IntroToRuby.ppt>
- <http://pragmaticprogrammer.com/>
- <http://www.pragmaticprogrammer.com/talks/perlmongers>

