MI-RUB Sharing Functionality

Pavel Strnad
pavel.strnad@fel.cvut.cz

Dept. of Computer Science, FEE CTU Prague,
Karlovo nám. 13, 121 35
Praha, Czech Republic

MI-RUB, WS 2011/12

Contents

Sharing Functionality

Ruby's Mechanisms

- single class-level inheritance,
- mixins.

What is the difference between single and multiple inheritance?

Sharing Functionality

Ruby's Mechanisms

- single class-level inheritance,
- mixins.

What is the difference between single and multiple inheritance?

Sharing Functionality

Ruby's Mechanisms

- single class-level inheritance,
- mixins.

What is the difference between single and multiple inheritance?

Inheritance

Each Ruby class has exactly one superclass.

```
class Parent
end
class Child < Parent
end
puts "The superclass of Child is #{Child.superclass}"
puts "The superclass of Parent is #{Parent.superclass}"
```

Inheritance

Iheritance

- The child inherits all of the capabilities of its parent class—all the parent's instance methods are available in instances of the child.
- The subclass is a specialization of a superclass.
- If you don't define an explicit superclass when defining a class, Ruby automatically makes the built-in class Object that class's parent.
- When subclassing do not forget call *super()* in *initialize*.

Question

What is a superclass of Object?

Inheritance

Iheritance

- The child inherits all of the capabilities of its parent class—all the parent's instance methods are available in instances of the child.
- The subclass is a specialization of a superclass.
- If you don't define an explicit superclass when defining a class, Ruby automatically makes the built-in class Object that class's parent.
- When subclassing do not forget call *super()* in *initialize*.

Question

What is a superclass of Object?

Inheritance

Iheritance

- The child inherits all of the capabilities of its parent class—all the parent's instance methods are available in instances of the child.
- The subclass is a specialization of a superclass.
- If you don't define an explicit superclass when defining a class, Ruby automatically makes the built-in class Object that class's parent.
- When subclassing do not forget call *super()* in *initialize*.

Question

What is a superclass of Object?

Inheritance

Iheritance

- The child inherits all of the capabilities of its parent class—all the parent's instance methods are available in instances of the child.
- The subclass is a specialization of a superclass.
- If you don't define an explicit superclass when defining a class, Ruby automatically makes the built-in class Object that class's parent.
- When subclassing do not forget call *super()* in *initialize*.

Question

What is a superclass of Object?

Inheritance

Iheritance

- The child inherits all of the capabilities of its parent class—all the parent's instance methods are available in instances of the child.
- The subclass is a specialization of a superclass.
- If you don't define an explicit superclass when defining a class, Ruby automatically makes the built-in class Object that class's parent.
- When subclassing do not forget call *super()* in *initialize*.

Question

What is a superclass of Object?

Method Overriding

```
class Person
  def initialize(name)
    @name = name
  end
  # method overriding
  def to_s
    "Person named #{@name}"
  end
end
pers = Person.new("Michael")
puts pers
```

Question

Where is *to_s* originally defined?

Inheritance

Exercise 1

Ruby comes with a library called GServer that implements basic TCP server functionality. You add your own behavior to it by subclassing the **GServer** class. Let's use that to write some code that waits for a client to connect on a socket and then returns the last few lines of the system log file (or another file, try to look to **/var/log**).

Modules

Modules

Modules are a way of grouping together methods, classes, and constants. Modules give you two major benefits:

- Modules provide a namespace and prevent name clashes.
- Modules support the mixin facility.

Modules cannot be instantiated.

Modules

Modules

Modules are a way of grouping together methods, classes, and constants. Modules give you two major benefits:

- Modules provide a namespace and prevent name clashes.
- Modules support the mixin facility.

Modules cannot be instantiated.

Modules

Modules

Modules are a way of grouping together methods, classes, and constants. Modules give you two major benefits:

- Modules provide a namespace and prevent name clashes.
- Modules support the mixin facility.

Modules cannot be instantiated.

Modules as Namespaces

Modules as Namespaces

- Namespaces are good to organize classes into sandboxes.
- Namespaces can avoid name clashes.
- Namespaces help us organizing reusable pieces of a code.

Modules as Namespaces

Modules as Namespaces

- Namespaces are good to organize classes into sandboxes.
- Namespaces can avoid name clashes.
- Namespaces help us organizing reusable pieces of a code.

Modules as Namespaces

Modules as Namespaces

- Namespaces are good to organize classes into sandboxes.
- Namespaces can avoid name clashes.
- Namespaces help us organizing reusable pieces of a code.

Modules as Namespaces

```
module Trig
  PI = 3.141592654
  def Trig.sin(x)
  # ..
  end
  def Trig.cos(x)
  # ..
  end
end

module Moral
  VERY_BAD = 0
  BAD = 1
  def Moral.sin(badness)
  # ...
  end
end
```

```
require 'trig'
require 'moral'
y = Trig.sin(Trig::PI/4)
wrongdoing = Moral.sin(Moral::
    VERY_BAD)
```

Mixins

Namespaces

Single iheritance is cleaner and easier than multiple, but it has many drawbacks.

- In a real world objects often inherits attributes and behavior from multiple sources.
- Ex. A ball is a spherical thing but also a bouncing thing.
- Ruby offers an interesting and powerful compromise, giving you the simplicity of single inheritance and the power of multiple inheritance.
- Mixin is like a partial class definition.
- Mixins are modules.

Mixins

Namespaces

Single iheritance is cleaner and easier than multiple, but it has many drawbacks.

- In a real world objects often inherits attributes and behavior from multiple sources.
- Ex. A ball is a spherical thing but also a bouncing thing.
- Ruby offers an interesting and powerful compromise, giving you the simplicity of single inheritance and the power of multiple inheritance.
- Mixin is like a partial class definition.
- Mixins are modules.

Mixins

Namespaces

Single iheritance is cleaner and easier than multiple, but it has many drawbacks.

- In a real world objects often inherits attributes and behavior from multiple sources.
- Ex. A ball is a spherical thing but also a bouncing thing.
- Ruby offers an interesting and powerful compromise, giving you the simplicity of single inheritance and the power of multiple inheritance.
- Mixin is like a partial class definition.
- Mixins are modules.

Mixins

Namespaces

Single iheritance is cleaner and easier than multiple, but it has many drawbacks.

- In a real world objects often inherits attributes and behavior from multiple sources.
- Ex. A ball is a spherical thing but also a bouncing thing.
- Ruby offers an interesting and powerful compromise, giving you the simplicity of single inheritance and the power of multiple inheritance.
- Mixin is like a partial class definition.
- Mixins are modules.

Mixins

Namespaces

Single iheritance is cleaner and easier than multiple, but it has many drawbacks.

- In a real world objects often inherits attributes and behavior from multiple sources.
- Ex. A ball is a spherical thing but also a bouncing thing.
- Ruby offers an interesting and powerful compromise, giving you the simplicity of single inheritance and the power of multiple inheritance.
- Mixin is like a partial class definition.
- Mixins are modules.

Mixin Example

```ruby
module Debug
  def who_am_i?
  "#{self.class.name} (\##{self
      .object_id}): #{self.to_s
      }"
end
end
class Phonograph
  include Debug
# ...
end
class EightTrack
  include Debug
# ...
end
```

```ruby
ph = Phonograph.new("West End
    Blues")
et = EightTrack.new("
    Surrealistic Pillow")
ph.who_am_i? # => "Phonograph
    (#330450): West End Blues"
et.who_am_i? # => "EightTrack
    (#330420): Surrealistic
    Pillow"
```

Mixin Example

```ruby
class Person
  include Comparable
  attr_reader :name
  def initialize(name)
    @name = name
  end
  def to_s
    "#{@name}"
  end
  def <=>(other)
    self.name <=> other.name
  end
end
```

```ruby
p1 = Person.new("Matz")
p2 = Person.new("Guido")
p3 = Person.new("Larry")
# Compare a couple of names
if p1 > p2
  puts "#{p1.name}'s name > #{
      p2.name}'s name"
end
# Sort an array of Person
    objects
puts "Sorted list:"
puts [ p1, p2, p3].sort
```

Inheritance

Exercise 2

Examine the method *inject* located in module Enumerable. Try to code a class VowelFinder which includes module Enumerable to be able iterate over vowels in a string. The usage of a VowelFinder is following:

```
vf = VowelFinder.new("the quick brown fox jumped")
vf.inject(:+) # => "euiooue"
```

Code a module Summable with a method *sum* which corresponds to *inject(:+)* and include it to VowelFinder class.

```
class VowelFinder
  include Summable
end
vf = VowelFinder.new("the quick brown fox jumped")
vf.sum # => "euiooue"
```