ÉCOLE
POLYTECHNIQUE
DE LOUVAIN

LSINF2335

Programming Paradigms: Theory, Practice and Applications

Sebastián González

22 March 2016

Lecture 7

Reflection in Ruby

Partly inspired on :

- the pickaxe book “Programming Ruby” by Dave Thomas
- slides by Prof. Tom Mens (UMons)
- slides by Prof. Wolfgang De Meuter (VUB)

6.1 Higher-Order Programs

About blocks, procs and lambdas

Different ways of manipulating “chunks of code” in Ruby:

methods, blocks, arg& notation, procs, lambda

Blocks

Typical usage:

```
3.times { print "Ruby" }
```

But blocks are not first-class

Trying these in Ruby produces a syntax error:

```
{ print "Ruby" }
x = { print "Ruby" }
```

Blocks should always be associated to a method invocation,

and *`yield`* executes that block in the method

But blocks can be turned into first class objects...

The &arg notation


```
class ClosureTester
  def evaluate_block(arg1,arg2)
    yield(arg1,arg2)
  end
  def reify_block(arg1,arg2,&arg3)
    arg3
  end
end

c = ClosureTester.new
puts c.evaluate_block(1,2) { |a1,a2| a1 + a2 }
# prints 3
aClosure = c.reify_block(1,2) { |a1,a2| a1 + a2 }
# returns a “procedure” containing the block
puts aClosure.call(3,4)
# prints 7
aClosure.class
# => Proc
```

Blocks are the last argument of a message. &arg explicitly refers to that argument

The &arg notation


```
class ClosureTester
  def evaluate_block(arg1,arg2)
    yield(arg1,arg2)
  end
  def reify_block(arg1,arg2,&arg3)
    arg3
  end
end

c = ClosureTester.new
puts c.evaluate_block(1,2) { |a1,a2| a1 + a2 }
# prints 3
aClosure = c.reify_block(1,2) { |a1,a2| a1 + a2 }
# returns a “procedure” containing the block
puts aClosure.call(3,4)
# prints 7
aClosure.class
# => Proc
```

Blocks are the last argument of a message. &arg explicitly refers to that argument

Note: yield produces an error when no block is passed with method

Proc and lambda

Procs are a more direct way of turning a block into a first class closure

```
aClosure = Proc.new { |a1,a2| a1+a2 }
puts aClosure.call(5,6)
# prints 11
```

Alternative: with proc or lambda:

```
aClosure = proc { |a1,a2| a1+a2 }
puts aClosure.call(5,6)
# prints 11
aClosure = lambda { |a1,a2| a1+a2 }
puts aClosure.call(5,6)
# prints 11
```

Proc and lambda

Procs are a more direct way of turning a block into a first class closure

```
aClosure = Proc.new { |a1,a2| a1+a2 }
puts aClosure.call(5,6)
# prints 11
```

Alternative: with proc or lambda:

```
aClosure = proc { |a1,a2| a1+a2 }
puts aClosure.call(5,6)
# prints 11
aClosure = lambda { |a1,a2| a1+a2 }
puts aClosure.call(5,6)
# prints 11
```

Note: `proc` and `lambda` are equivalent (but `proc` is mildly deprecated)

6.2 Singleton Methods

Singleton Methods

- Methods specific to a particular instance
- A form of dynamic object extension

```
class Student
  def reflection
    puts "I do not understand reflection"
  end
end

aStudent = Student.new
aStudent.reflection
# prints : I do not
  understand reflection
```

```
uclStudent = Student.new

def uclStudent.reflection
  puts "I understand reflection"
end

uclStudent.reflection
# prints : I understand reflection
```

Singleton Methods

- Methods specific to a particular instance
- A form of dynamic object extension

```
class Student
  def reflection
    puts "I do not understand reflection"
  end
end

aStudent = Student.new
aStudent.reflection
# prints : I do not
  understand reflection
```

no equivalent in Smalltalk (at language level)

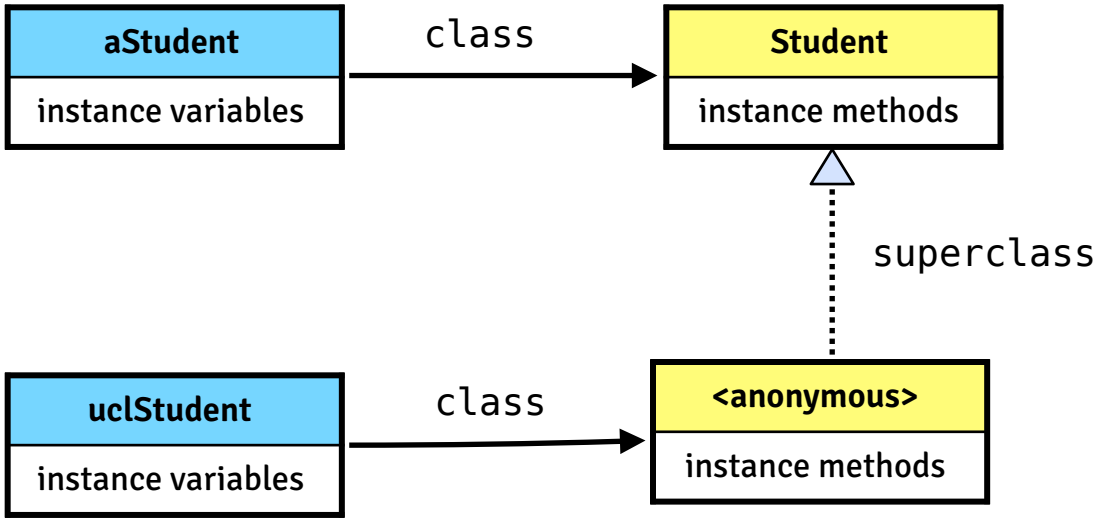
```
uclStudent = Student.new

def uclStudent.reflection
  puts "I understand reflection"
end

uclStudent.reflection
# prints : I understand reflection
```

Singleton Classes

If methods are stored in classes, then where are **singleton methods** stored?

Ruby creates an anonymous ***singleton class*** specific to the instance

This singleton class has the original class of the instance as superclass

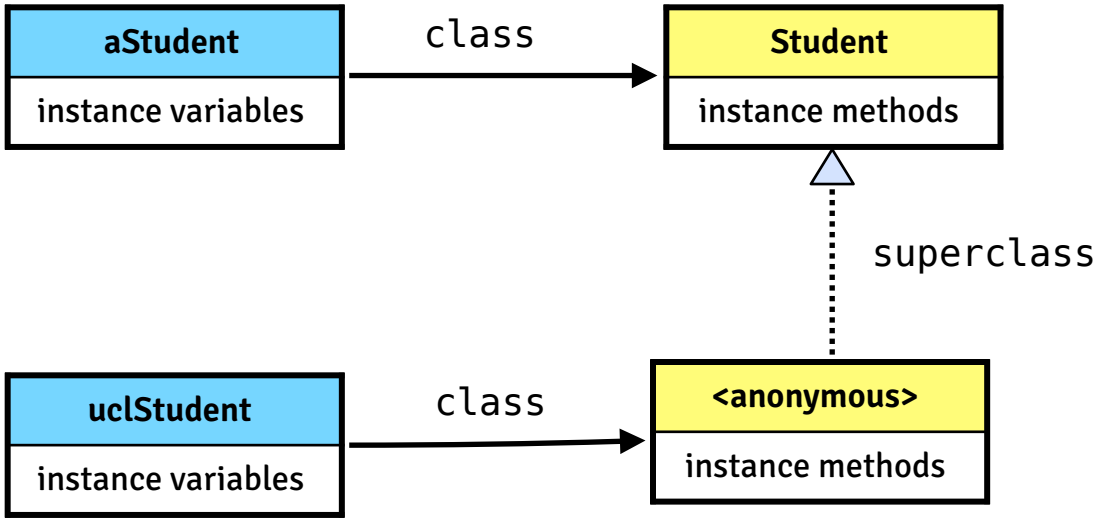


Singleton Classes

If methods are stored in classes, then where are **singleton methods** stored?

Ruby creates an anonymous ***singleton class*** specific to the instance

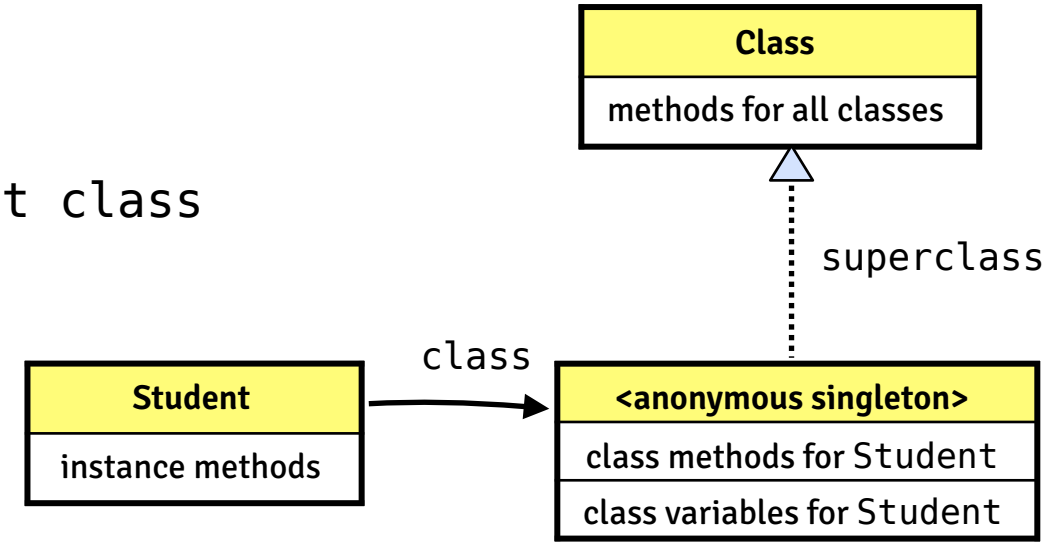
This singleton class has the original class of the instance as superclass



The anonymous singleton class is "hidden": `smartStudent.class` returns `Student`

Class Methods Are Singleton Methods


```
class Student
  def self.some_class_method
    puts "I'm the Student class"
  end
end
aStudent = Student.new
aStudent.some_class_method
# NoMethodError
Student.some_class_method
# prints : I'm the Student class
```



Some Remarks

- Notice the analogy with Smalltalk
  - In Smalltalk, class methods (and variables) are stored in special meta-classes (one meta-class per class)
  - In Ruby, class methods (and variables) are stored in singleton classes, specific for each class
- Anonymous singleton classes are “hidden”
  - `smartStudent.class` returns `Student`
  - but there are ways of accessing the singleton class...

6.3 Mixin Modules

Mixin Modules

What if I want to inherit from multiple classes?

Mixin modules are an interesting alternative to multiple inheritance

- Rather than inheriting from multiple parent classes, you can "mix in" different modules

Mixins modules

- are like a partial class definition
- with late binding
  - use of `self` in a mixin method is late bound

Defining a Mixin Module


```
module Reflection
  def reflection
    puts "I understand reflection"
  end
end
```

a simple mixin module

```
module Debug
    def who_am_i
        print self.class.name + "(\#" +
              self.object_id.to_s + ")" +
              self.to_s
    end
end
```

another mixin module

Late binding of `self` in class where module will be mixed in

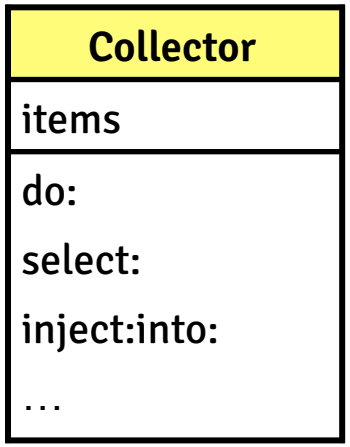
Including a Mixin In a Class


```
class Student
end
class ReflectionStudent < Student
  include Reflection
  include Debug
end

aStudent = ReflectionStudent.new
aStudent.reflection
# prints : “I understand reflection”

aStudent.who_am_i
# prints :
ReflectionStudent(#292320)#<ReflectionStudent:
0x8ebc0>
```

Implementing Iterators With the Enumerable Mixin


```
→ ri Enumerable


The Enumerable mixin provides collection classes with several traversal and
searching methods, and with the ability to sort. The class must provide a
method each, which yields successive members of the collection. If
Enumerable#max, #min, or #sort is used, the objects in the collection must
also implement a meaningful <=> operator, as these methods rely on an ordering
between members of the collection.
------------------------------------------------------------------------------
= Instance methods:

  all?, any?, chunk, collect, collect_concat, count, cycle, detect, drop,
  drop_while, each_cons, each_entry, each_slice, each_with_index,
  each_with_object, entries, find, find_all, find_index, first, flat_map,
  grep, group_by, include?, inject, lazy, map, max, max_by, member?, min,
  min_by, minmax, minmax_by, none?, one?, partition, reduce, reject,
  reverse_each, select, slice_before, sort, sort_by, take, take_while, to_a,
  to_set, zip
```

Class Collector In Smalltalk




```
do: aBlock
    ^items do: aBlock
select: aBlock
    ^items select: aBlock
inject: aValue into: aBlock
    ^items inject: aValue into: aBlock
…
```

How can we make such a Collector class in Ruby?

- which implements typical operators on collections like `collect`, `select`, `sort`, ...

Easy:

- implement the method each to iterate over the items
- including the `Enumerable` mixin does the rest

Using the Enumerable Mixin


```
class Collector



    attr_reader :items
    def initialize(collection)
        @items = collection
    end









end
```

Using the Enumerable Mixin


```
class Collector

    include Enumerable

    attr_reader :items
    def initialize(collection)
        @items = collection
    end

    def each
        @items.each do |item|
            yield item
        end
    end



end
```

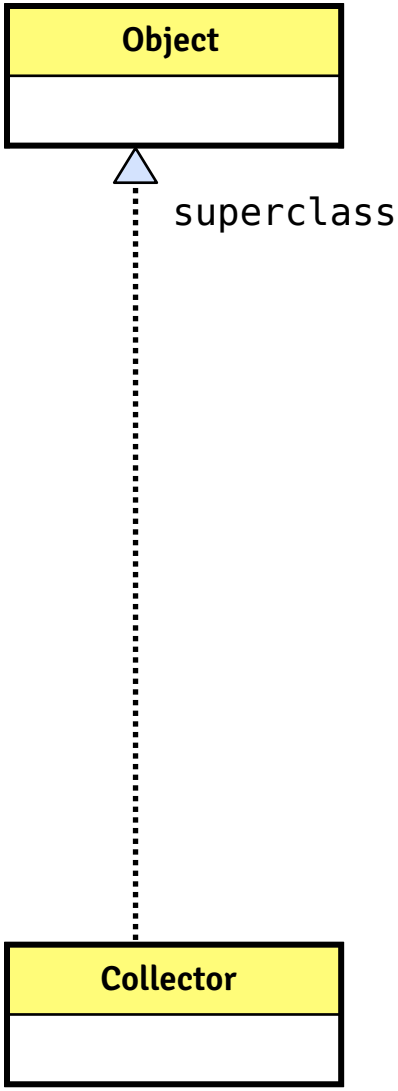
Using the Enumerable Mixin


```
class Collector

    include Enumerable
    include Debug
    attr_reader :items
    def initialize(collection)
        @items = collection
    end

    def each
        @items.each do |item|
            yield item
        end
    end
    def to_s
        @items.to_s + "\n"
    end
end
```

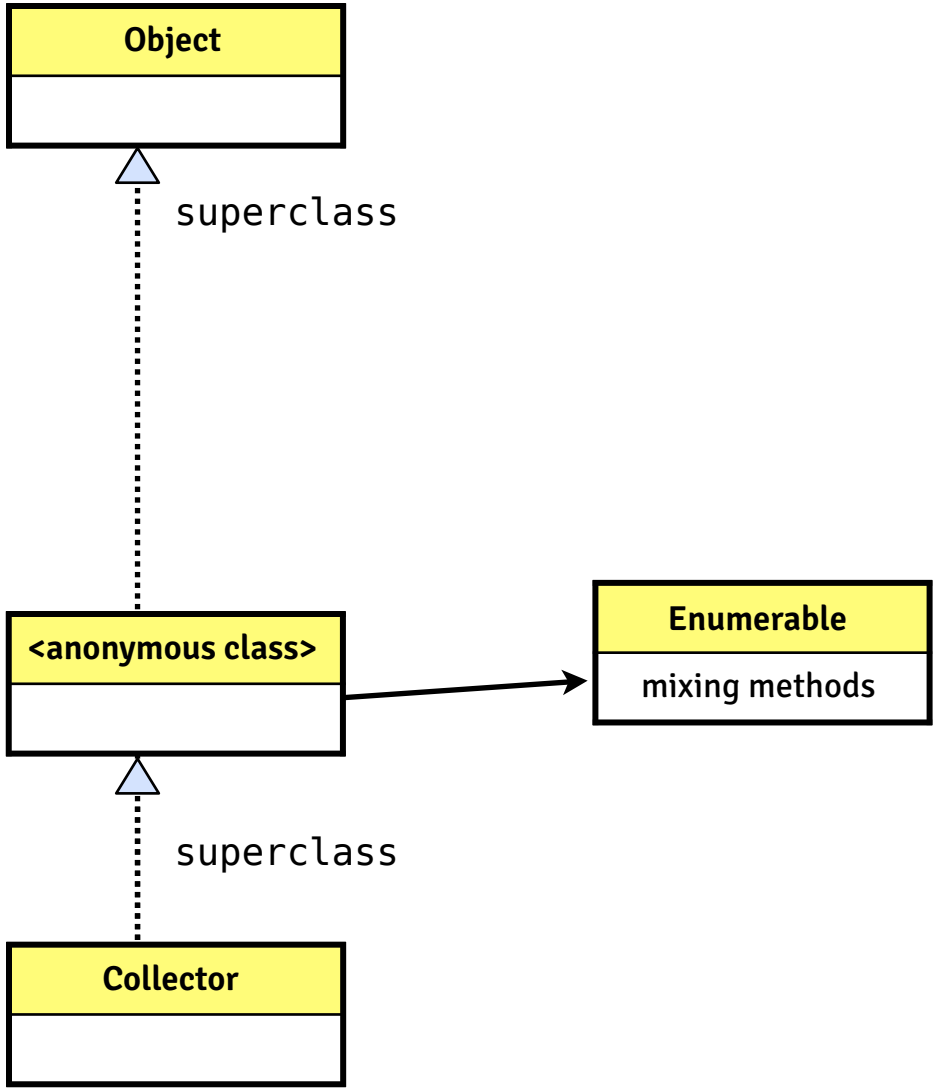
Let's Try It Out…


```
myCollector = Collector.new([1,3,2])
  => #<Collector:0x321dc4 @items=[1, 3, 2]>

myCollector.who_am_i
  Collector (#1642210): 132
  => nil

myCollector.each { |el| print " Value: "+el.to_s }
   Value: 1 Value: 3 Value: 2
  => [1, 3, 2]

newCollector = Collector.new(myCollector.sort)

  => #<Collector:0x31aa38 @items=[1, 2, 3]> newCollector.who_am_i
  Collector (#1627420): 123

myCollector.max
  => 3
```

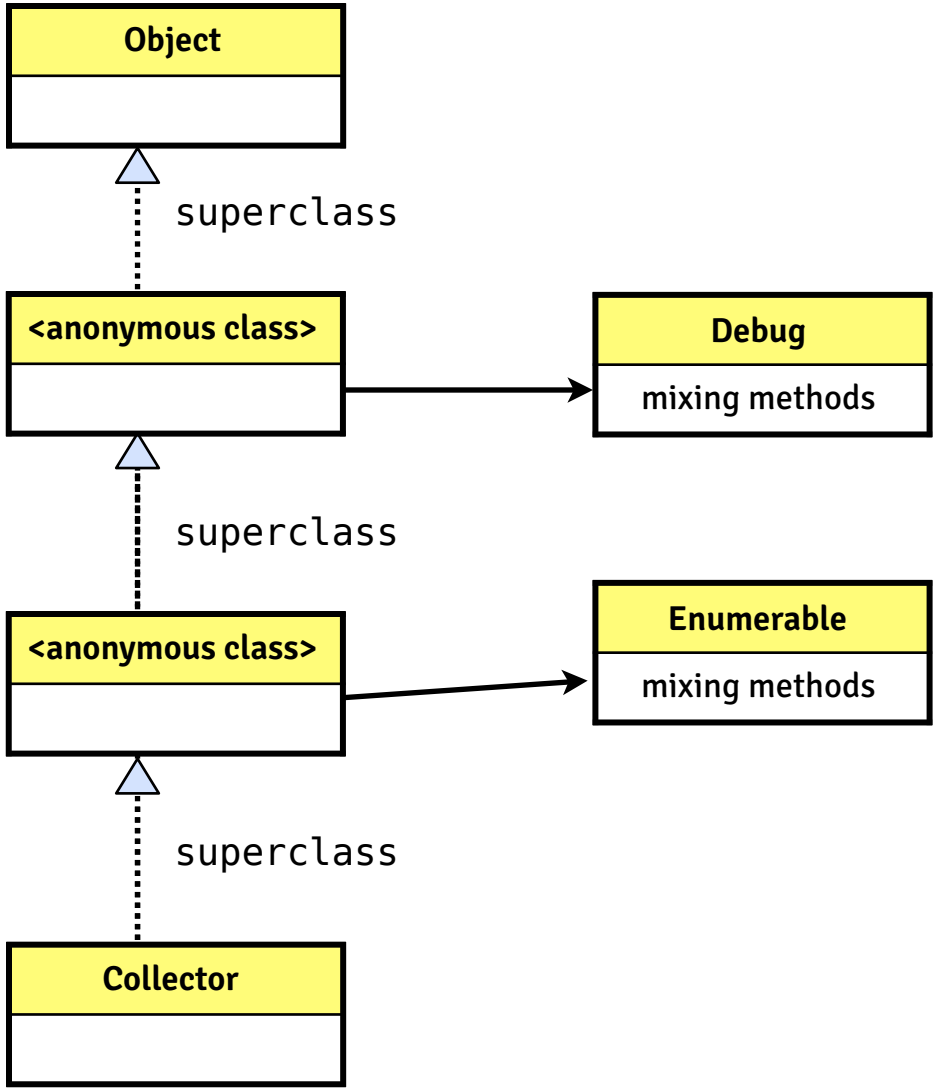
How Do Mixin Modules Work?



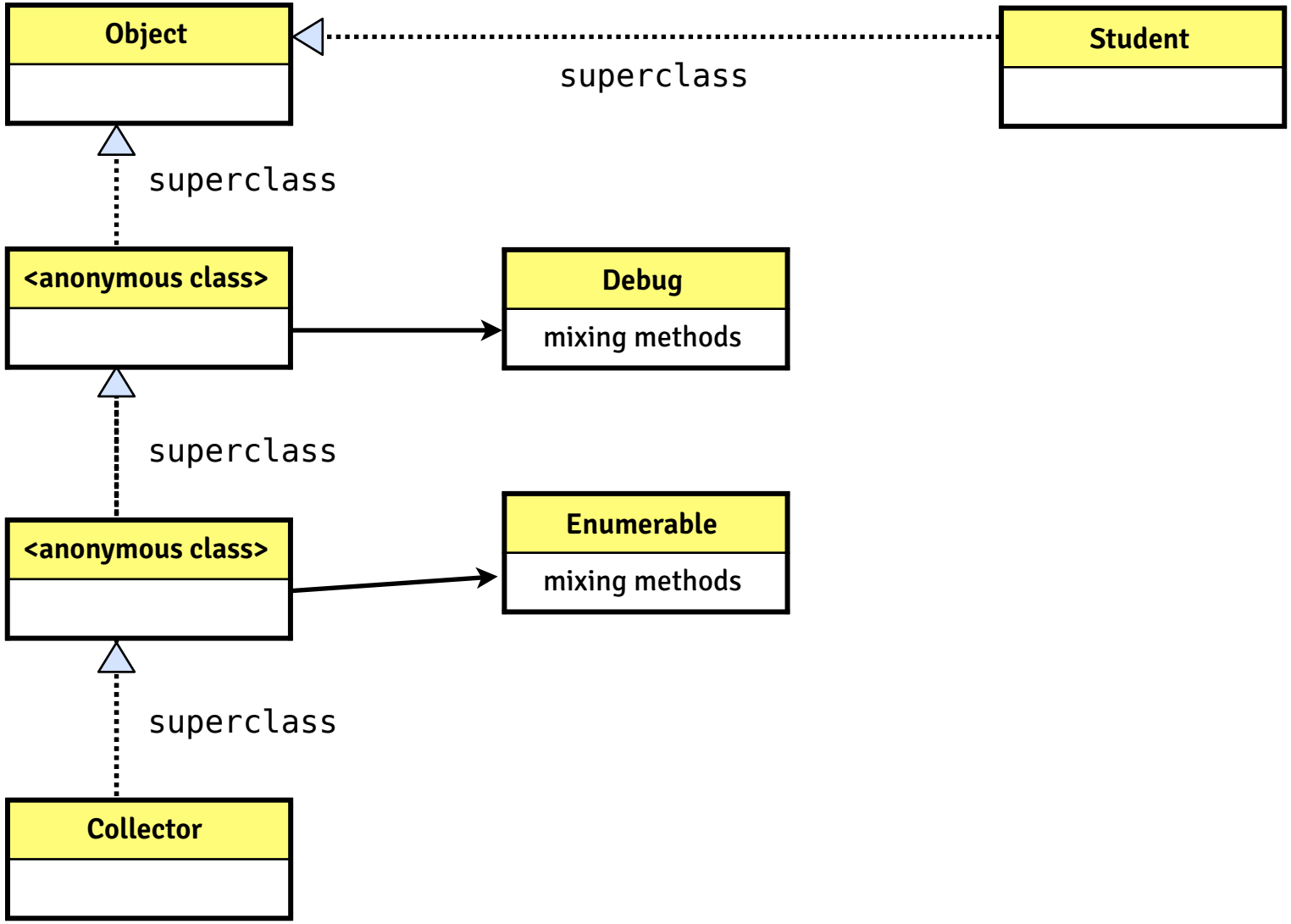
How Do Mixin Modules Work?



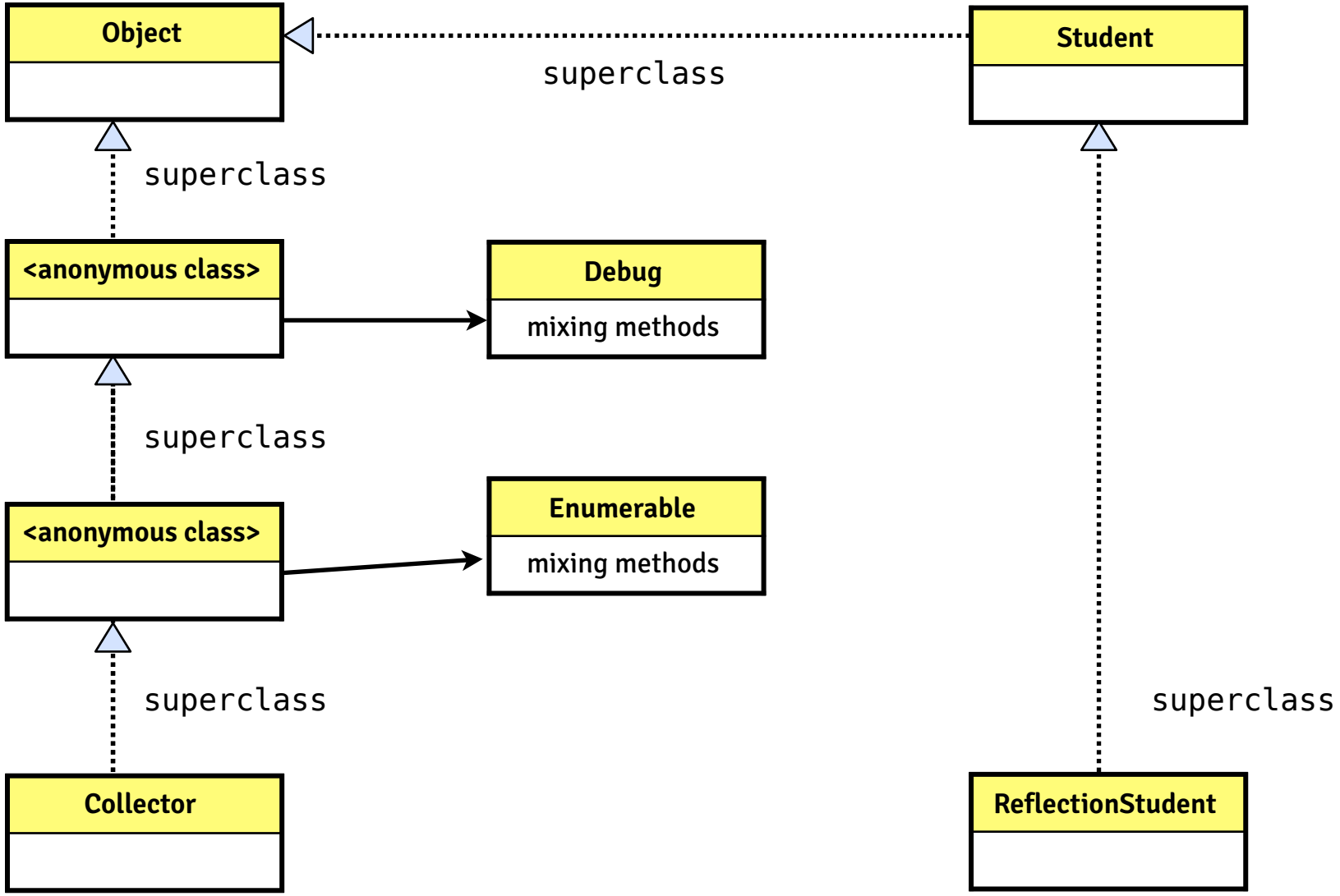
How Do Mixin Modules Work?



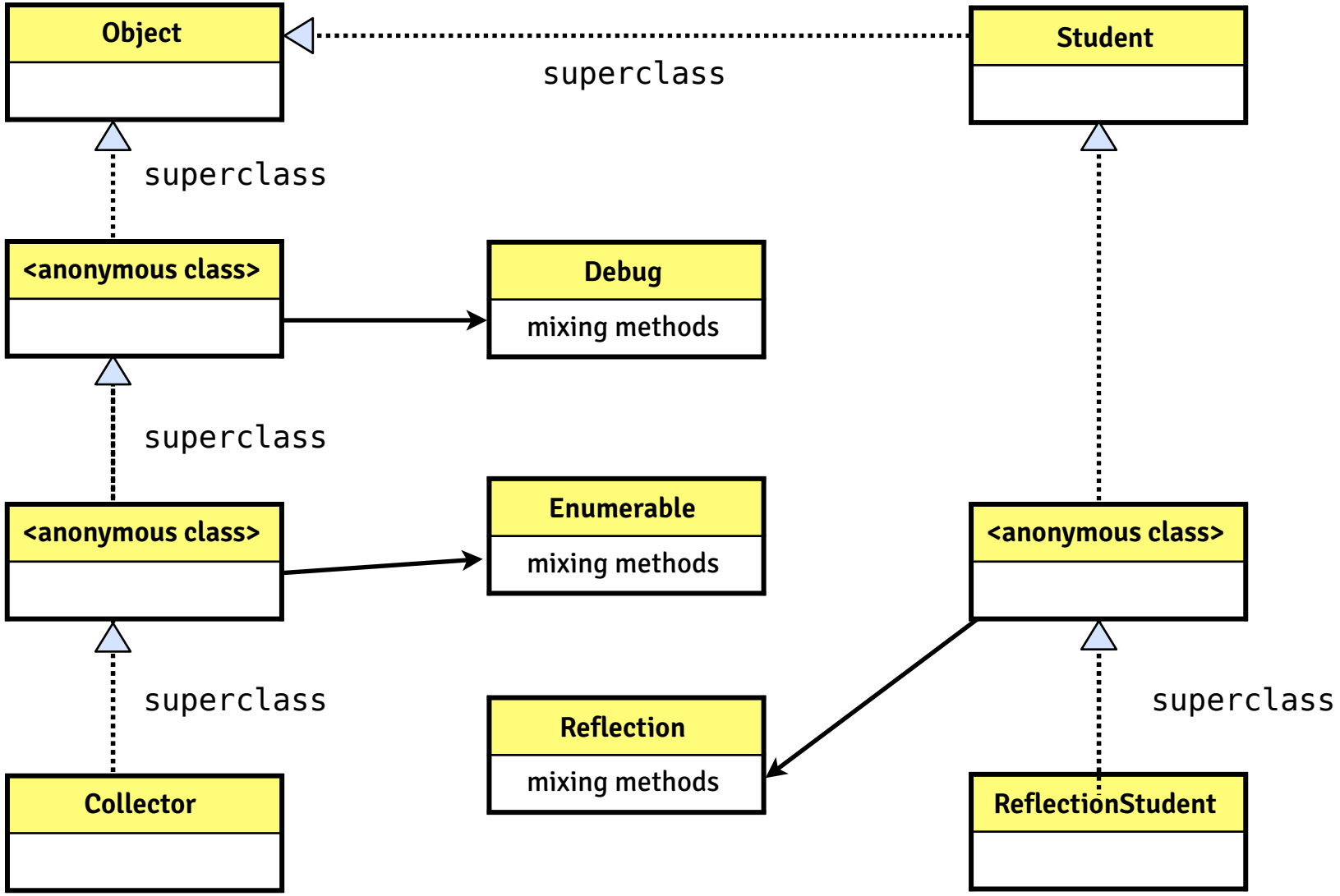
How Do Mixin Modules Work?



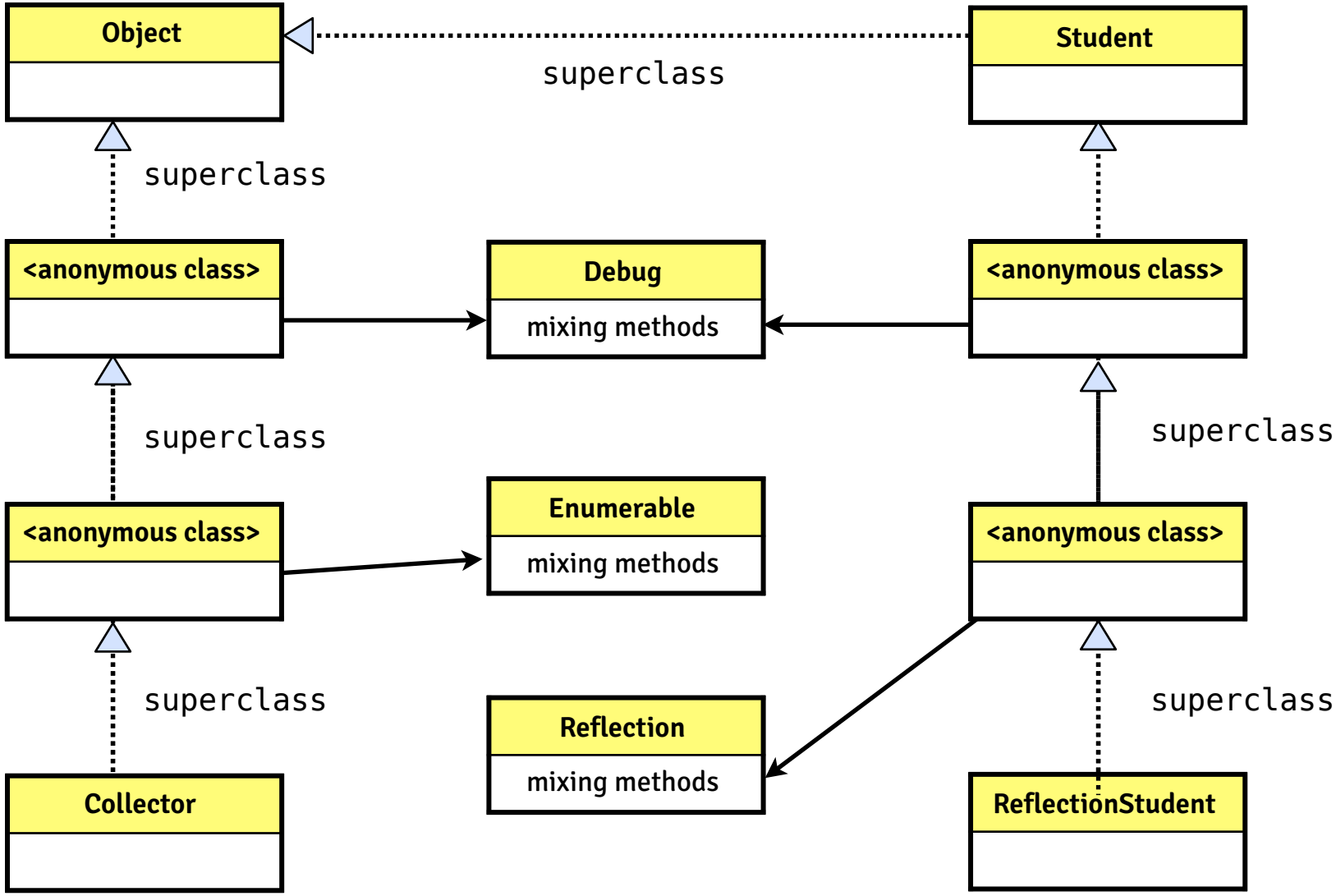
How Do Mixin Modules Work?



How Do Mixin Modules Work?



How Do Mixin Modules Work?



Extending An Object With a Mixin


```
module Reflection
  def reflection
    puts "I understand reflection"
  end
end
```

```
aStudent = Student.new
aStudent.extend Reflection
aStudent.reflection
# prints : “I understand reflection”
otherStudent = Student.new
otherStudent.reflection
# NoMethodError
```

6.4 Reflective Features in Ruby

Before Starting…

- A bit of introspection at the prompt
- Use ri to read ruby documentation:

  `ri Fixnum` for documentation of a class

  `ri Fixnum#meth` for documentation on an instance method

  `ri Fixnum::meth` for documentation on a class method

  `ri -l` for documentation on all classes

Reflective Features in Ruby

1. The Class `Class`
2. Meta-Object Protocol
3. Class-Level Macros
4. Eval
5. Hook Methods

Reflective Features in Ruby

1. The Class `Class`
2. Meta-Object Protocol
3. Class-Level Macros
4. Eval
5. Hook Methods

Going Meta

Everything is an object; you can ask any object for its class:

```
42.class => Fixnum
:foo.class => Symbol
“Ruby”.class => String
true.class     => TrueClass
[1,2,3].class  => Array
{ :a => 1 }.class => Hash
```

Even a class is an object; its class is a special class called `Class`

```
Fixnum.class   => Class
42.class.class   => Class
```

`Class` is an object too; the class of `Class` is `Class` itself

```
Class.class   => Class
```

The Class `Class`

All classes are an instance of the class `Class`

`Class` implements some interesting introspection methods:

- `instance_variables, instance_variable_get, class_variables, …`
- `methods, public_methods, private_instance_methods, …`
- `superclass, class, …`
- and many more.

And some intercession methods too:

- `instance_variable_set`

If you forgot what reflective methods are available, use introspection :-)

```
Class.public_methods.sort
```

Manipulating An Instance Reflectively


```
class Example
end

ex = Example.new

ex.instance_variables                # => []

ex.instance_variable_set(:@x,1)      # => 1

ex.instance_variables                # => ["@x"]

ex.instance_variable_defined?(:@x) # => true

ex.instance_variable_defined?(:@y) # => false

ex.instance_variable_get(:@x)        # => 1
```

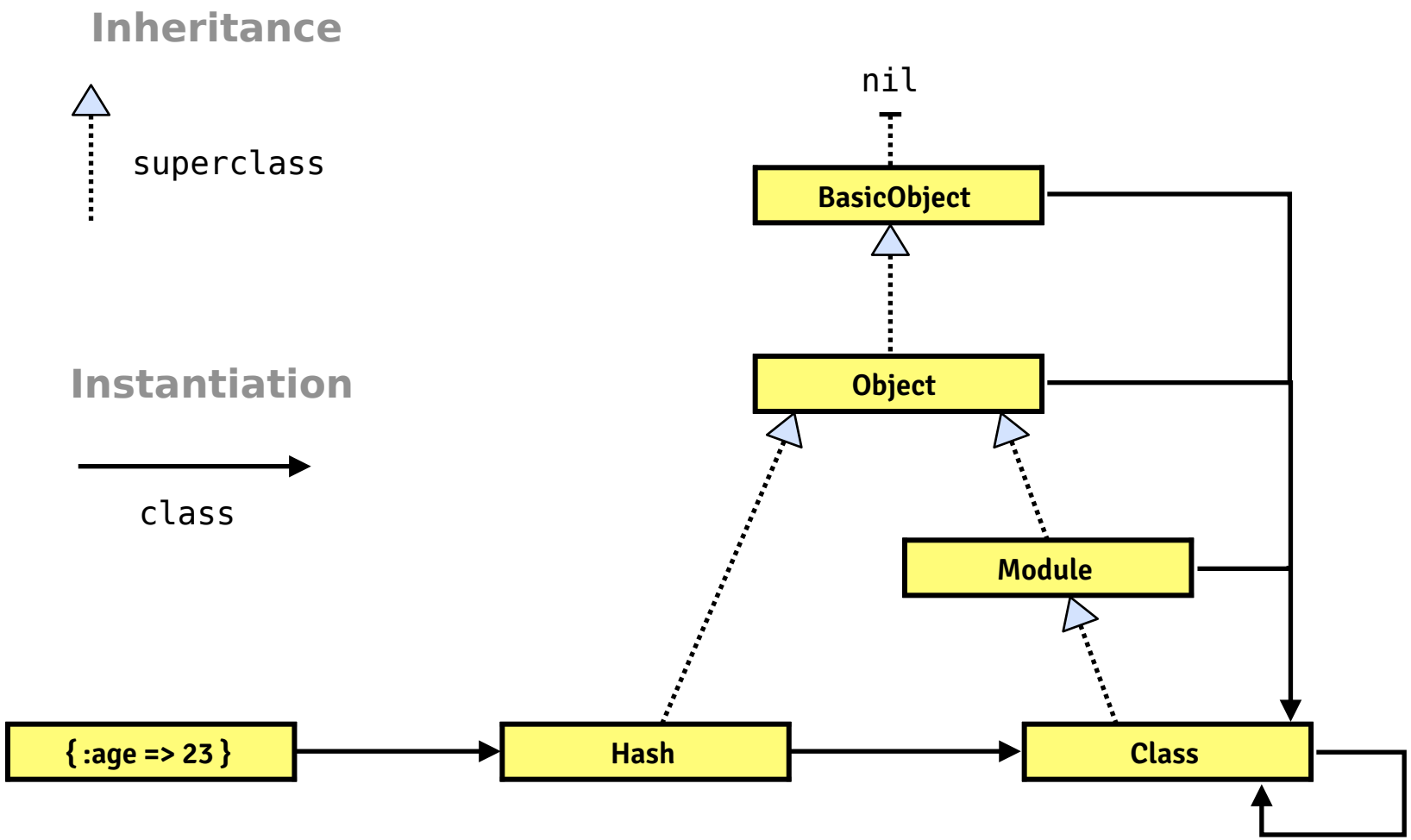
Reflective Features in Ruby

1. The Class `Class`
2. Meta-Object Protocol
3. Class-Level Macros
4. Eval
5. Hook Methods

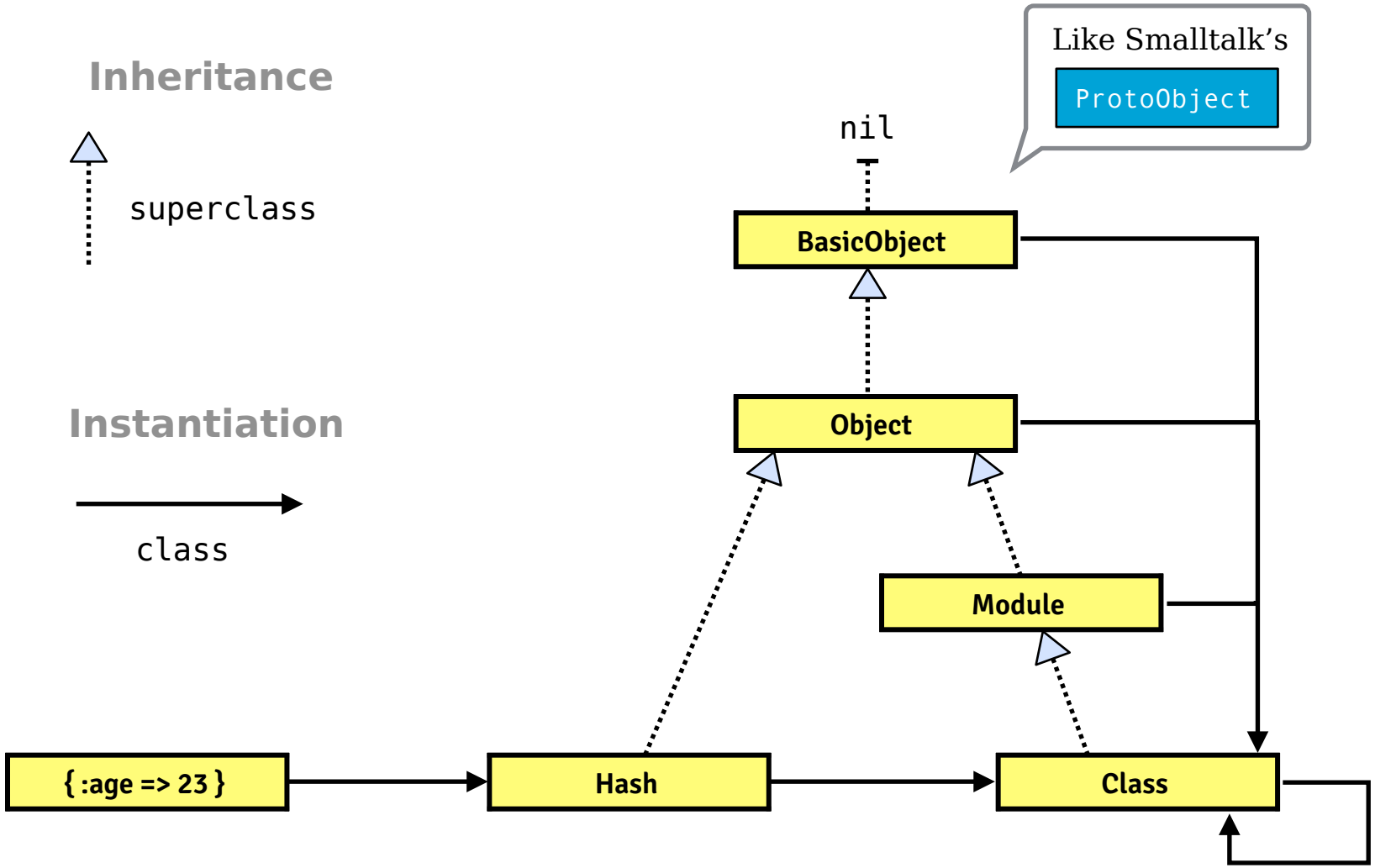
Reflective Features in Ruby

1. The Class `Class`
2. Meta-Object Protocol
3. Class-Level Macros
4. Eval
5. Hook Methods

Ruby's Meta-Object Protocol

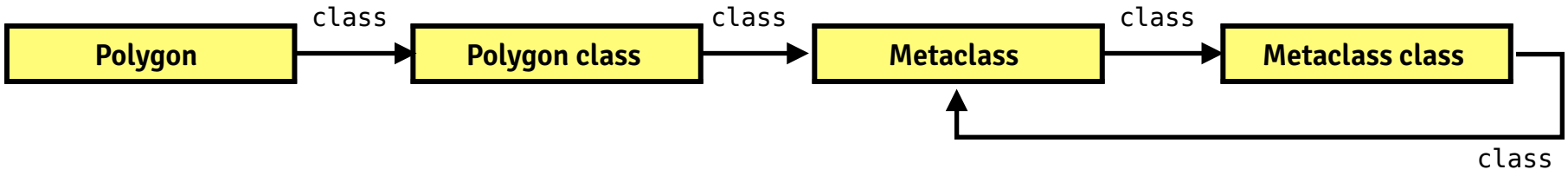


Ruby's Meta-Object Protocol

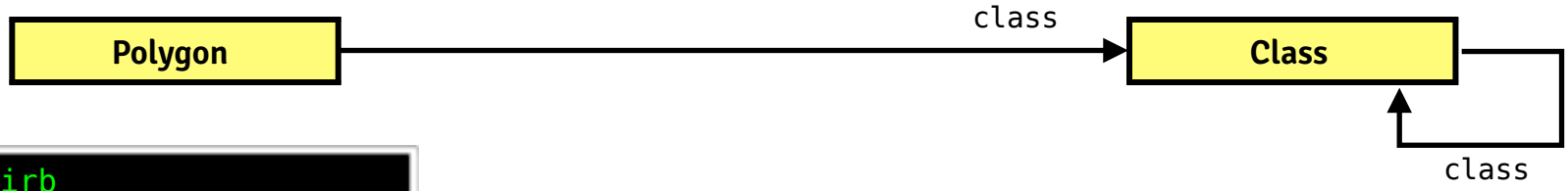


Ruby's Absorbed Metaclasses

Smalltalk ✓ metaclasses are first class



Ruby

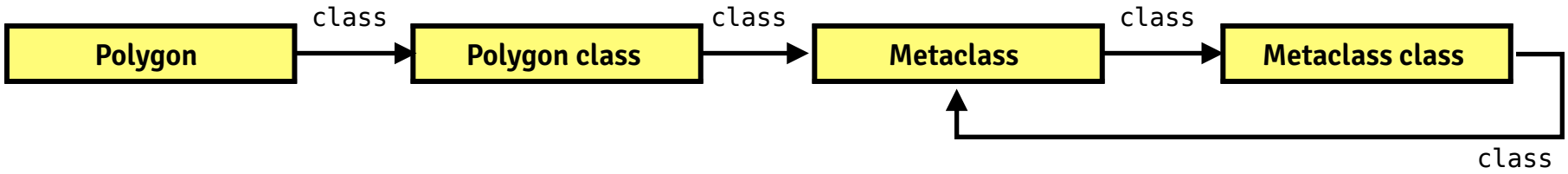



```
$ irb
> class Polygon; end
> Polygon.class
Class
> Polygon.new
=> #<Polygon:0x007fe4…>
```

"an instance of Polygon"

Ruby's Absorbed Metaclasses

Smalltalk ✓ metaclasses are first class



Ruby


```
$ irb
> class Polygon; end
> Polygon.class
Class
> Polygon.new
=> #<Polygon:0x007fe4…>
```

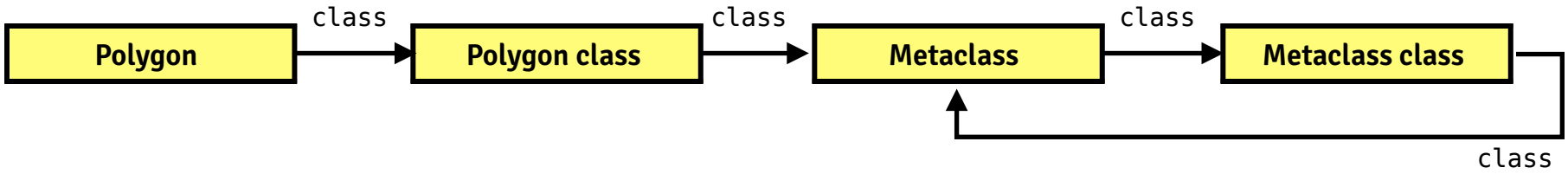
"an instance of Polygon"

```
$ irb
> class Polygon
>   def self.metaclass
>     class << self
>       self
>     end
>   end
> end
=> nil
> Polygon.metaclass
=> #<Class:Polygon>
```

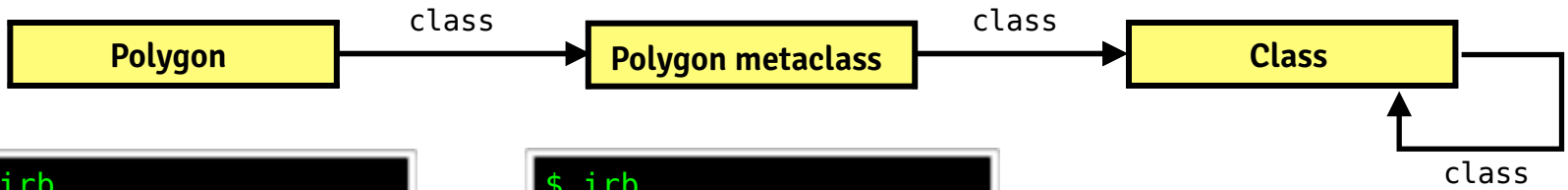
"an instance of Class for the Polygon class"
= metaclass for Polygon

Ruby's Absorbed Metaclasses

Smalltalk ✓ metaclasses are first class



Ruby




```
$ irb
> class Polygon; end
> Polygon.class
Class
> Polygon.new
=> #<Polygon:0x007fe4…>
```

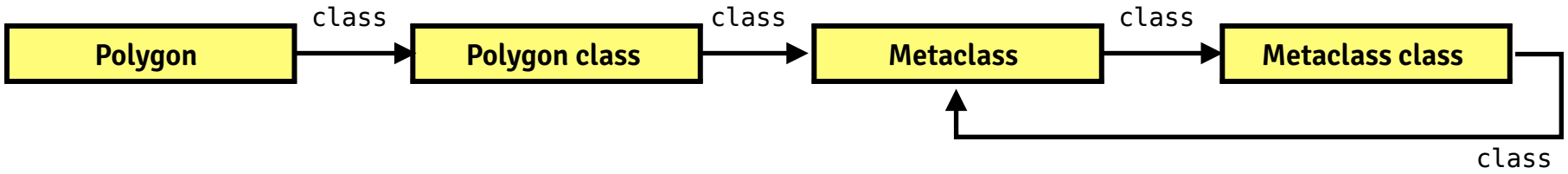
"an instance of Polygon"

```
$ irb
> class Polygon
>   def self.metaclass
>     class << self
>       self
>     end
>   end
> end
=> nil
> Polygon.metaclass
=> #<Class:Polygon>
```

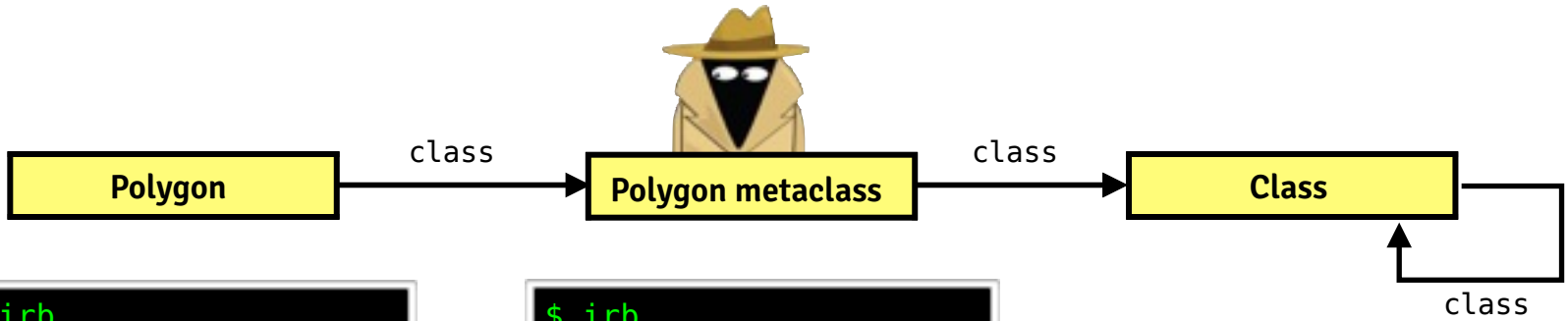
"an instance of Class for the Polygon class"
= metaclass for Polygon

Ruby's Absorbed Metaclasses

Smalltalk ✓ metaclasses are first class



Ruby ~ metaclasses are an absorbed (hidden) concept




```
$ irb
> class Polygon; end
> Polygon.class
Class
> Polygon.new
=> #<Polygon:0x007fe4…>
```

"an instance of Polygon"

```
$ irb
> class Polygon
>   def self.metaclass
>     class << self
>       self
>     end
>   end
> end
=> nil
> Polygon.metaclass
=> #<Class:Polygon>
```

"an instance of Class for the Polygon class"
= metaclass for Polygon

Reflective Features in Ruby

1. The Class `Class`
2. Meta-Object Protocol
3. Class-Level Macros
4. Eval
5. Hook Methods

Reflective Features in Ruby

1. The Class `Class`
2. Meta-Object Protocol
3. Class-Level Macros
4. Eval
5. Hook Methods

Remember These?

```
class ComplexCartesian
    ...
    attr_reader :x, :y
    attr_writer :x, :y
    ...
end
```

or, alternatively:

```
class ComplexCartesian
    ...
    attr_accessor :x, :y
    ...
end
```

These are examples of **class-level macros**

i.e., class-level methods that generate code behind the scenes

Creating a Class-Level Macro

Example: define accessor methods that not only read/write a variable, but also log access to the variable.

Example of intended usage:

```
>> ex = Example.new
=> #<Example:0x352208>
>> ex.value=6
Assigning 6 to value
=> 6
>> ex.value
Reading value
=> 6
```

Let's Start Simply


```
class Logger

  def self.add_logging

    def log(msg)

      STDERR.puts Time.now.strftime("%H:%M:%S: ")
                    + "#{self} (#{msg})"

    end
  end
end

class Example < Logger
  add_logging
end
ex = Example.new
ex.log("hello")
```

Let's Start Simply


```
class Logger

  def self.add_logging

    def log(msg)

      STDERR.puts Time.now.strftime("%H:%M:%S: ")
                    + "#{self} (#{msg})"

    end
  end
end

class Example < Logger
  add_logging
end
ex = Example.new
ex.log("hello")
```

```
>> ex.log("hello")
10:11:51: #<Example:0x35ad04> (hello)
=> nil
```

Let's Start Simply

```
class Logger

  def self.add_logging

    def log(msg)

      STDERR.puts Time.now.strftime("%H:%M:%S: ")
                    + "#{self} (#{msg})"

    end
  end
end
```

This is a **class method**

```
>> ex.log("hello")
10:11:51: #<Example:0x35ad04> (hello)
=> nil
```

```
class Example < Logger
  add_logging
end
ex = Example.new
ex.log("hello")
```

Let's Start Simply


```
class Logger

  def self.add_logging

    def log(msg)

      STDERR.puts Time.now.strftime("%H:%M:%S: ")
                    + "#{self} (#{msg})"

    end
  end
end
```

This is a **class method**

This instance method will get installed when executing the class method

```
>> ex.log("hello")
10:11:51: #<Example:0x35ad04> (hello)
=> nil
```

```
class Example < Logger
  add_logging
end
ex = Example.new
ex.log("hello")
```

Let's Start Simply


```
class Logger

  def self.add_logging

    def log(msg)

      STDERR.puts Time.now.strftime("%H:%M:%S: ")
                    + "#{self} (#{msg})"

    end
  end
end
```

This is a **class method**

This instance method will get installed when executing the class method

```
>> ex.log("hello")
10:11:51: #<Example:0x35ad04> (hello)
=> nil
```

```
class Example < Logger
  add_logging
end
ex = Example.new
ex.log("hello")
```

Install logging method for instances of class `Example`

Let's Start Simply

```
class Logger

  def self.add_logging

    def log(msg)

      STDERR.puts Time.now.strftime("%H:%M:%S: ")
                    + "#{self} (#{msg})"

    end
  end
end
```

This is a **class method**

This instance method will get installed when executing the class method

```
>> ex.log("hello")
10:11:51: #<Example:0x35ad04> (hello)
=> nil
```

```
class Example < Logger
  add_logging
end
ex = Example.new
ex.log("hello")
```

Install logging method for instances of class `Example`

Instances of `Example` now understand the log method

A Logged Getter And Setter


```
class AttrLogger

  def self.add_logged_accessors
    def set(newval)
      puts "Assigning #{newval}"
      @val = newval
    end
    def get
      puts "Reading"
      @val
    end
  end

end

class Example < AttrLogger
  add_logged_accessors
end
```

```
ex = Example.new
=> #<Example:0x3534c8>
>> ex.set 6
Assigning 6
=> 6
>> ex.get
Reading
=> 6
```

A Logged Getter And Setter


```
class AttrLogger

  def self.add_logged_accessors
    def set(newval)
      puts "Assigning #{newval}"
      @val = newval
    end
    def get
      puts "Reading"
      @val
    end
  end

end

class Example < AttrLogger
  add_logged_accessors
end
```

Class method

```
ex = Example.new
=> #<Example:0x3534c8>
>> ex.set 6
Assigning 6
=> 6
>> ex.get
Reading
=> 6
```

A Logged Getter And Setter


```
class AttrLogger

  def self.add_logged_accessors
    def set(newval)
      puts "Assigning #{newval}"
      @val = newval
    end
    def get
      puts "Reading"
      @val
    end
  end

end

class Example < AttrLogger
  add_logged_accessors
end
```

Class method

Instance method to set value

```
ex = Example.new
=> #<Example:0x3534c8>
>> ex.set 6
Assigning 6
=> 6
>> ex.get
Reading
=> 6
```

A Logged Getter And Setter


```
class AttrLogger

  def self.add_logged_accessors
    def set(newval)
      puts "Assigning #{newval}"
      @val = newval
    end
    def get
      puts "Reading"
      @val
    end
  end

end

class Example < AttrLogger
  add_logged_accessors
end
```

Class method

Instance method to set value

Instance method to get value

```
ex = Example.new
=> #<Example:0x3534c8>
>> ex.set 6
Assigning 6
=> 6
>> ex.get
Reading
=> 6
```

A Logged Getter And Setter


```
class AttrLogger

  def self.add_logged_accessors
    def set(newval)
      puts "Assigning #{newval}"
      @val = newval
    end
    def get
      puts "Reading"
      @val
    end
  end

end

class Example < AttrLogger
  add_logged_accessors
end
```

Class method

Instance method to set value

with logging

Instance method to get value

with logging

```
ex = Example.new
=> #<Example:0x3534c8>
>> ex.set 6
Assigning 6
=> 6
>> ex.get
Reading
=> 6
```

Method `define_method`

- But how can I create a method of which also the name is parameterized?

  for example, for a given symbol `:value` I want to generate
  - an instance variable `@value`
  - a logged reader method named `value`
  - a logged writer method named `value=(newvalue)`

- Use **`define_method`**

```
define_method(name) do |param1 ... paramn|
  body
end
```

Becomes the method name

Becomes the method parameters

Becomes the method body

Putting Everything Together


```
class AttrLogger
  def self.attr_logger(name)
    define_method("#{name}=") do |val|
      puts "Assigning #{val.inspect} to #{name}"
      instance_variable_set("@#{name}", val)
    end
    define_method("#{name}") do
      puts "Reading #{name}"
      instance_variable_get("@#{name}")
    end
  end
end

class Example < AttrLogger
  attr_logger :value
end
```

```
ex = Example.new
=> #<Example:0x3534c8>
>> ex.value=6
Assigning 6 to value
=> 6
>> ex.value
Reading value
=> 6
```

Class-Level Macros In RoR

Class-level macros are used heavily in Ruby on Rails

```
class Project < ActiveRecord::Base
    belongs_to                :portfolio
    has_one                   :project_manager
    has_many                  :milestones
    has_and_belongs_to_many :categories
  end
```

Reflective Features in Ruby

1. The Class `Class`
2. Meta-Object Protocol
3. Class-Level Macros
4. Eval
5. Hook Methods

Reflective Features in Ruby

1. The Class `Class`
2. Meta-Object Protocol
3. Class-Level Macros
4. Eval
5. Hook Methods

eval

Methods `instance_eval` and `class_eval`

- defined on `Object`
- allow you to temporarily set `self` to some arbitrary object
- evaluates the block, passed with eval, in that context
- then resets `self` to its original value

`class_eval`

sets `self` as if you were in the body of the class definition of the receiver

`instance_eval`

sets `self` as if you were working inside the singleton class of the receiver

class_eval: Adding An Instance Method


```
class TestClass
end

test = TestClass.new
test.testMethod
# NoMethodError

TestClass.class_eval do
    def testMethod
      puts "I exist"
    end
end
# instance method testMethod has now been added

test.testMethod
# prints: I exist
```

class_eval: Adding An Instance Method


```
class TestClass
end

test = TestClass.new
test.testMethod
# NoMethodError

TestClass.class_eval do
    def testMethod
      puts "I exist"
    end
end
# instance method testMethod has now been added

test.testMethod
# prints: I exist
```

Define a class

class_eval: Adding An Instance Method


```
class TestClass
end

test = TestClass.new
test.testMethod
# NoMethodError

TestClass.class_eval do
    def testMethod
      puts "I exist"
    end
end
# instance method testMethod has now been added

test.testMethod
# prints: I exist
```

Define a class

Do this as if inside the class definition

instance_eval: Adding A Singleton Method


```
class TestClass
end

test = TestClass.new
test.testMethod
#NoMethodError

test.instance_eval do
    def testMethod
      puts "I exist"
    end
end
# singleton method testMethod has been added to test

test.testMethod
# prints: I exist
test2 = TestClass.new
test2.testMethod
#NoMethodError
```

instance_eval: Adding a Class Method


```
class TestClass
end

TestClass.instance_eval do
    def testMethod
      puts "I exist"
    end
end
# class method testMethod has now been added

test = TestClass.new
test.testMethod
#NoMethodError
TestClass.testMethod
# prints: I exist
```

instance_eval: Adding a Class Method


```
class TestClass
end

TestClass.instance_eval do
    def testMethod
      puts "I exist"
    end
end
# class method testMethod has now been added

test = TestClass.new
test.testMethod
#NoMethodError
TestClass.testMethod
# prints: I exist
```

Define a class

instance_eval: Adding a Class Method


```
class TestClass
end

TestClass.instance_eval do
    def testMethod
      puts "I exist"
    end
end
# class method testMethod has now been added

test = TestClass.new
test.testMethod
#NoMethodError
TestClass.testMethod
# prints: I exist
```

Define a class

Do this as if inside the singleton class

instance_eval: Adding a Class Method


```
class TestClass
end

TestClass.instance_eval do
    def testMethod
      puts "I exist"
    end
end
# class method testMethod has now been added

test = TestClass.new
test.testMethod
#NoMethodError
TestClass.testMethod
# prints: I exist
```

Define a class

Do this as if inside the singleton class

Confusing terminology:

- `class_eval` can be used to define instance methods
- `instance-eval` can be used to define class methods

Reflective Features in Ruby

1. The Class `Class`
2. Meta-Object Protocol
3. Class-Level Macros
4. Eval
5. Hook Methods

Reflective Features in Ruby

1. The Class `Class`
2. Meta-Object Protocol
3. Class-Level Macros
4. Eval
5. Hook Methods

Hook Method Example 1: Missing Methods


```
class Student
  def method_missing(name)
    puts "I do not understand " + name.to_s
  end
end

aStudent = Student.new

aStudent.reflection
# prints : “I do not understand reflection”
=> nil
```

Hook Method Example 1: Missing Methods


```
class Student
  def method_missing(name)
    puts "I do not understand " + name.to_s
  end
end

aStudent = Student.new

aStudent.reflection
# prints : “I do not understand reflection”
=> nil
```

If Ruby couldn't find a method after having looked it up in the object's class hierarchy, Ruby looks for a method called `method_missing` on the original receiver, starting back at the class of `self` and then looking up the superclass chain again.

Hook Method Example 1: Missing Methods

Like Smalltalk's `doesNotUnderstand`

```
class Student
  def method_missing(name)
    puts "I do not understand " + name.to_s
  end
end

aStudent = Student.new

aStudent.reflection
# prints : "I do not understand reflection"
=> nil
```

If Ruby couldn't find a method after having looked it up in the object's class hierarchy, Ruby looks for a method called `method_missing` on the original receiver, starting back at the class of `self` and then looking up the superclass chain again.

Hook Method Example 1: Missing Methods (ctd.)


```
class MethodCatcher
  def method_missing(name, *args, &block)
    puts "Name of missing method is #{name.to_s}."
    puts "Method arguments are #{args.to_s}."
    puts “Method body is #{block.inspect}."
  end
end

catch = MethodCatcher.new
catch.someMethod(1,2){puts "something"}
# prints :
   Name of missing method is someMethod.
   Method arguments are 12.
   Method body is #<Proc:0x0033b2ec@(irb):30>.
```

Hook Methods

`method_missing(name, *args, &block)`

is a hook method called by Ruby when some method is not found during method lookup

In general, hook methods (also called callbacks)

- are methods you write
- but that get called by the Ruby interpreter at run-time when some particular event occurs

Other Hook Methods

Method-related hooks

- adding an instance method: `method_added`
- adding a singleton method: `singleton_method_added`
- and many more: `method_removed`, `singleton_method_removed`, `method_undefined`, `singleton_method_undefined`

Class and module-related hooks

- subclassing: `inherited`
- mixing in a module: `extend_object`
- and many more: `append_features, included, extended, initialize_copy, const_missing`

Object marshaling hooks

- `marshal_dump, marshal_load`

Coercion hooks

- `coerce, induced_from, to_xxx`

Hook Method Example 2: Adding Methods


```
class MyClass
  def self.method_added(name)
    puts "Adding Method #{name}"
  end
  def new_method
    # blabla
  end
end

Adding Method new_method
=> nil
```

Hook Method Example 2: Adding Methods


```
class MyClass
  def self.method_added(name)
    puts "Adding Method #{name}"
  end
  def new_method
    # blabla
  end
end

Adding Method new_method
=> nil
```

Hook **method**

Hook Method Example 2: Adding Methods


```
class MyClass
  def self.method_added(name)
    puts "Adding Method #{name}"
  end
  def new_method
    # blabla
  end
end

Adding Method new_method
=> nil
```

Hook **method**

Instance **method**

Hook Method Example 2: Adding Methods


```
class MyClass
  def self.method_added(name)
    puts "Adding Method #{name}"
  end
  def new_method
    # blabla
  end
end

Adding Method new_method
=> nil
```

Hook **method**

Instance **method**

Hook method is executed when Ruby adds the instance **method**

6.5 Meta-Programming With Methods

Meta-Programming With Methods

1. Dynamic class extension
2. Calling methods dynamically
3. Removing methods dynamically
4. Method aliasing

Dynamically Add Methods To A Class


```
class Point
  def initialize(x,y)
    @x,@y = x,y
  end
end
```

a simple class

```
origin = Point.new(0,0)
point = Point.new(0,0)
origin == point      # => false
```

```
class Point
  attr_reader :x, :y
  def ==(p)
    @x==p.x
  end
end
```

extending the class (classes are "open")

```
origin == point      # => true
newpoint = Point.new(1,0)
origin == newpoint   # => false
```

old instances see the new methos too

Even Built-In Classes Are Open


```
[1,2,3].myfind  { | entry | entry == 2 }
# => NoMethodError: undefined method ‘myfind’ ...
```

```
class Array
  def myfind
    for i in 0...size
      value = self[i]
      return value if yield(value)
    end
    return nil
  end
end
```

```
[1,2,3].myfind  { | entry | true }          # => 1
[1,2,3].myfind  { | entry | entry == 2 }    # => 2
[1,2,3].myfind  { | entry | entry > 2 }     # => 3
```

Even Built-In Classes Are Open


```
[1,2,3].myfind  { | entry | entry == 2 }
# => NoMethodError: undefined method ‘myfind’ ...
```

```
class Array
  def myfind
    for i in 0...size
      value = self[i]
      return value if yield(value)
    end
    return nil
  end
end
```

Sidenote: actually the method `find` is already implemented on collection classes

```
[1,2,3].myfind  { | entry | true }          # => 1
[1,2,3].myfind  { | entry | entry == 2 }   # => 2
[1,2,3].myfind  { | entry | entry > 2 }    # => 3
```

Calling Methods Dynamically


```
class TestClass
  def testMethod
    puts "I exist"
  end
  def testMethod2(arg)
    puts "I also exist with argument #{arg}"
  end
  def method_missing(methodid)
      puts "#{methodid} does not exist"
  end
end

test = TestClass.new
selector = :testMethod
test.send(:testMethod)
value = 2
test.send("#{selector}2".to_s,value)
```

Removing Methods From a Class or Module


```
class TestClass
  def testMethod
    puts "I exist"
  end
  def method_missing(methodid)
      puts "#{methodid} does not exist"
  end
end

test = TestClass.new
test.testMethod  # prints: I exist
class TestClass
  remove_method(:testMethod)
end
test.testMethod
# prints: testMethod does not exist
```

Method Aliasing


```
class Object
  def timestamp
    return @timestamp
  end
  def timestamp=(aTime)
   @timestamp = aTime
  end
end

class Class
  alias_method :old_new, :new
  def new(*args)
    result = old_new(*args)
    result.timestamp = Time.now
    return result
  end
end
```

```
class Test
end
obj1 = Test.new
sleep 2
obj2 = Test.new

puts obj1.timestamp
# 2016-03-20 15:56:44 +0100
puts obj2.timestamp
# 2016-03-20 15:56:46 +0100
```

Conclusion

This was only an introduction to some of the powerful metaprogramming and reflective features offered by Ruby...

... but there exist many more

Just try them out for yourself !