Refinements – A new feature of Ruby

Shugo Maeda

2012-10-20T15:00:00 - 2012-10-20T15:45:00

Who am I?

Shugo Maeda

- Ruby committer
- Director of the Network Applied Communication Laboratory
- Secretary General of the Ruby Association

What's Ruby Association?

Organization dedicated to Ruby's

- Development, and
  - Grant program
  - Maintenance of the stable version
- Promotion
  - Certification for Ruby programmers
  - RubyWorld Conference (Nov 8 – Nov 9)
  - Seminars


I love motorcycles


I love fishing


I love my family

Today's topic

Refinements

- A new feature of Ruby 2.0

Background – existing class extensions

Subclassing

Mix-in

Singleton methods

Monkey patching

Subclassing

```
class Person
  attr_accessor :name
end


class Employee < Person
  attr_accessor :monthly_salary
end
```

Ruby's subclassing

Single inheritance

Superclasses aren't affected

Implementation-only inheritance

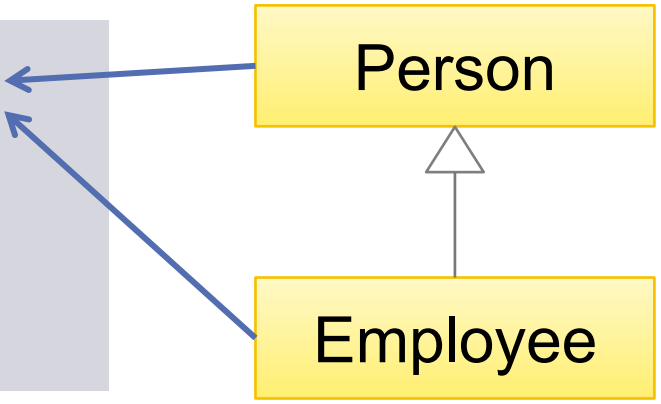
- Violations of LSP

LSP

Liskov Substitution Principle

- An instance of a subtype must behave like an instance of the supertype of the subtype

```
def print_name(person)
  puts person.name
end
```



An example of LSP violation in Ruby

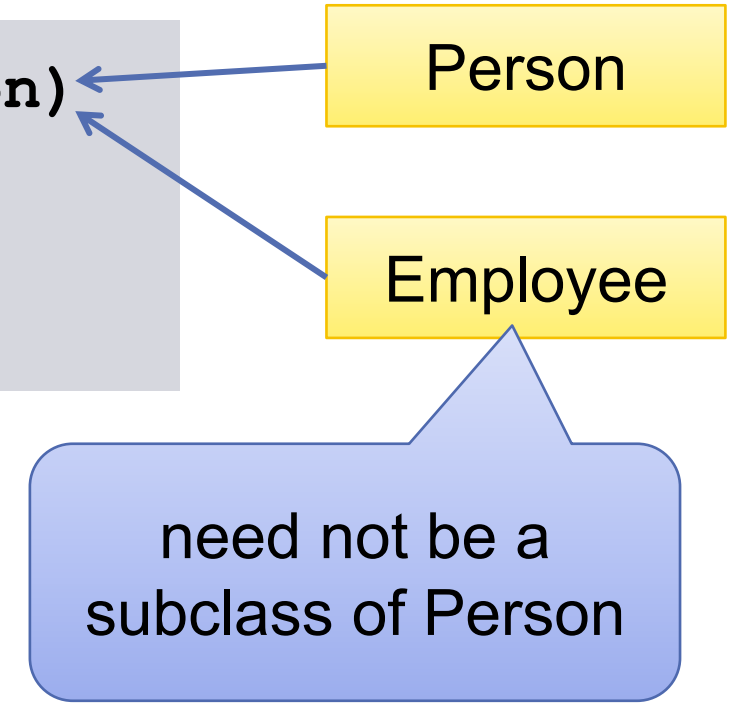
```
class Employee < Person
  undef name
end
def print_name(person)
  puts person.name
end
matz = Employee.new
matz.name = "Yukihiro Matsumoto"
print_name(matz) #=> undefined method
```

Implementation-only inheritance

Subclassing is not Subtyping

Duck typing

```
def print_name(person)
  puts person.name
end
```

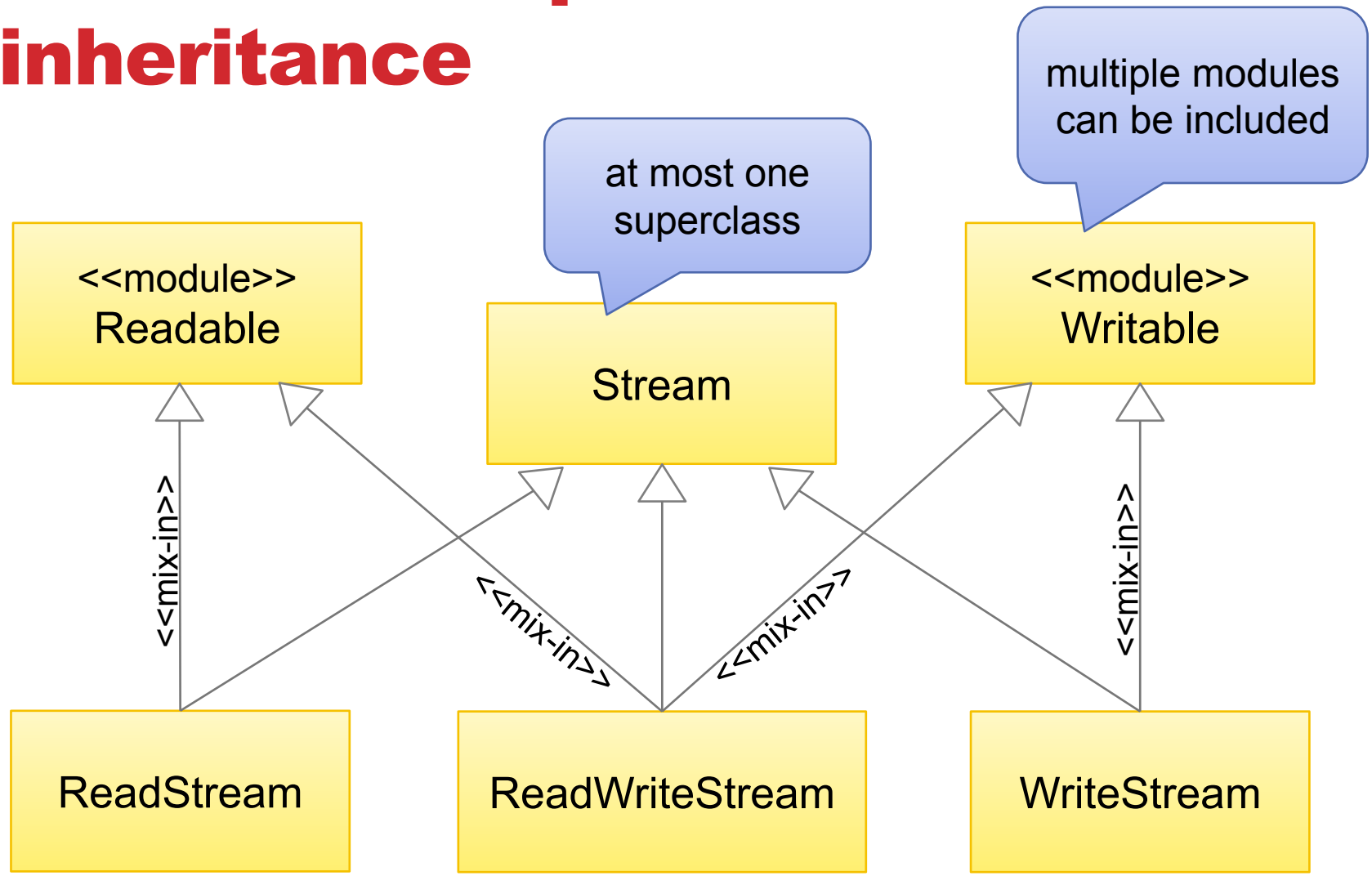


Mix-in

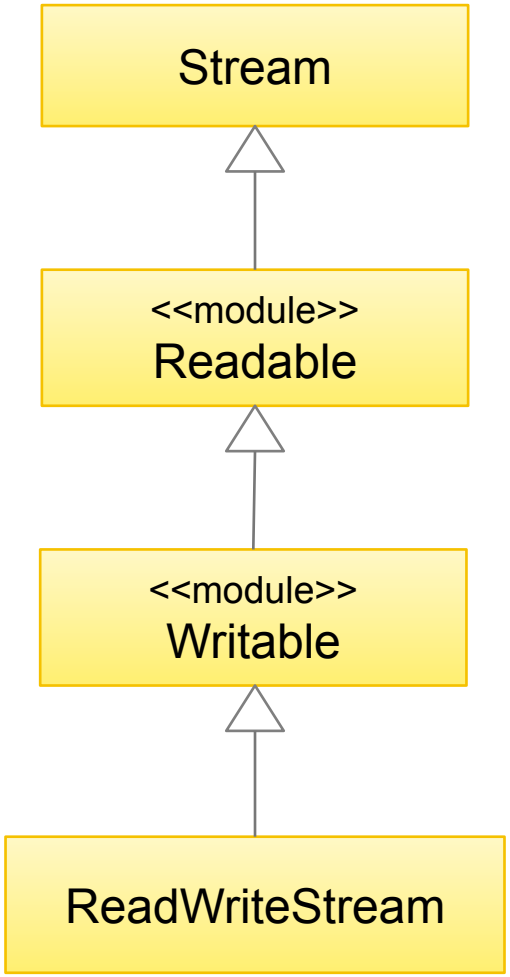
```
class; Stream; ... end
module Readable; ... end


class ReadStream < Stream
  include Readable
end
```

Limited multiple inheritance



Linearization of mix-in

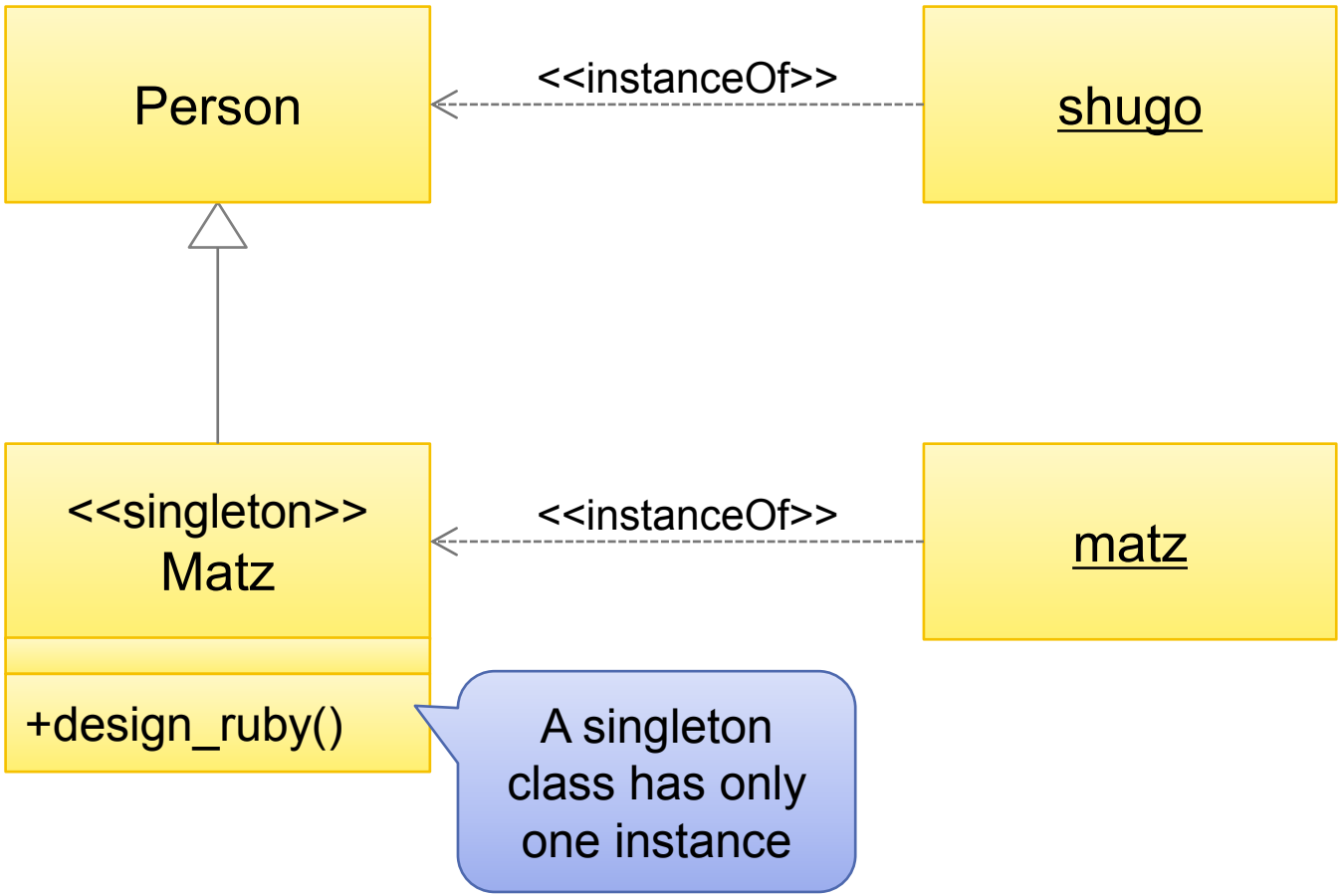



```
class Stream; end
module Readable; end
module Writable; end
class ReadWriteStream < Stream
  include Readable
  include Writable
end
```

Singleton methods

```
matz = Person.new
def matz.design_ruby
  ...
end
matz.design_ruby
shugo = Person.new
shugo.design_ruby #=> NoMethodError
```

Singleton classes behind singleton methods



Limitation of singleton methods

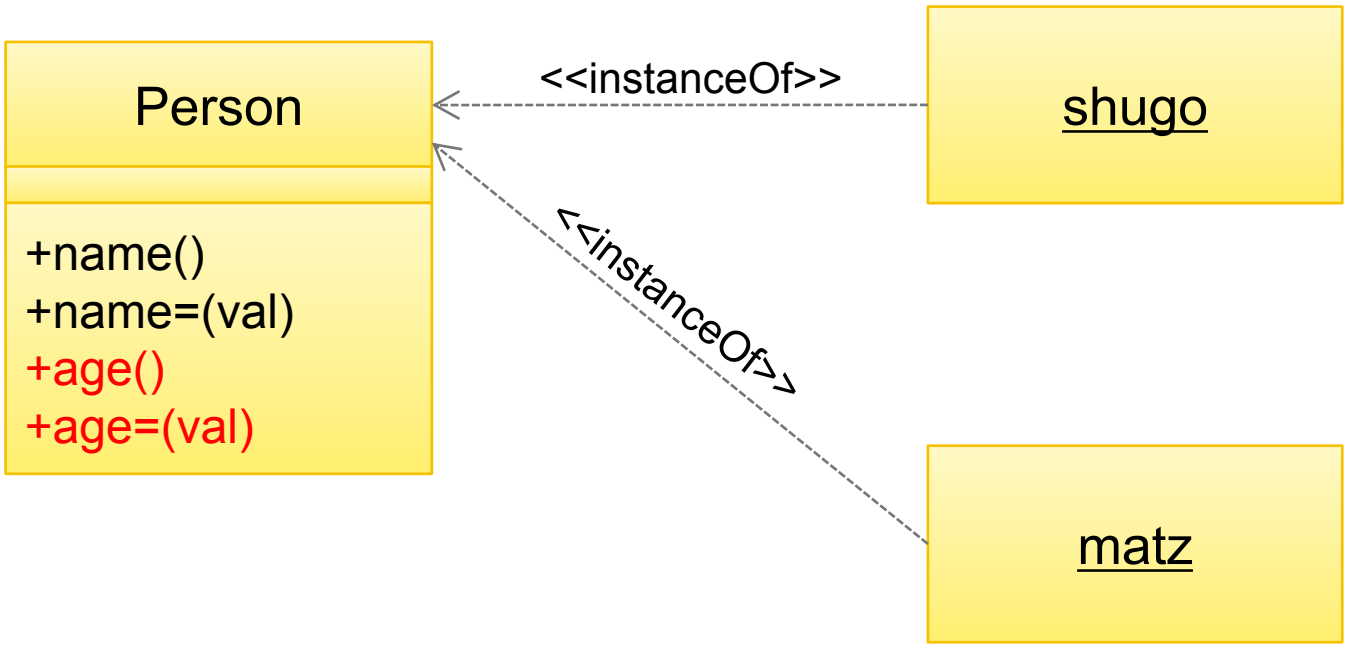
Singleton methods can't be defined for Numeric objects

```
x = 123
def x.foo #=> TypeError
  puts "foo"
end
```

Monkey patching

```
# reopen Person, and add code
class Person
  attr_accessor :age
end
shugo = Person.new
shugo.name = "Shugo Maeda"
shugo.age = 34
```

Monkey patching affects all instances



Examples of monkey patching

Ruby on Rails

- ActiveSupport
- Rails plugins

RSpec

Standard libraries

- jcode
- mathn

Monkey patching version of LSP

An instance of a monkey patched class must behave like an instance of the class without the patch

- Otherwise, it breaks existing code
- Because monkey patching affects globally

An example of LSP violation

```
# Fixnum#/ returns an integer
p 1 / 2 #=> 0


# With mathn, Fixnum#/ returns
# a rational number
require "mathn"
p 1 / 2 #=> (1/2)
```

Extensibility and modularity

Type of class extensions	Extensibility	Modularity
Subclassing	Poor	Excellent
Mix-in	Poor	Excellent
Singleton methods	Good	Good
Monkey patching	Excellent	Poor

What are Refinements?

A new way to extend classes


WARNING
Refinements are experimental, and may be removed from Ruby 2.0!

Terms and definitions

class extension

- a computational entity which extends a class or module

Refinements

- a mechanism to extend classes and modules locally

refinement

- a class extension used in Refinements

Design policy

Extensible

- Refinements can extend existing classes

Modular

- Lexically scoped

No new syntax/keyword

- Use methods and blocks to extend classes

No new semantic entity

- Use modules as namespaces of refinements

Example of a refinement

```
module MathN
  refine Fixnum do
    def /(other) quo(other) end
  end

  p 1 / 2 #=> (1/2)   affected
end
p 1 / 2   #=> 0       not affected
```

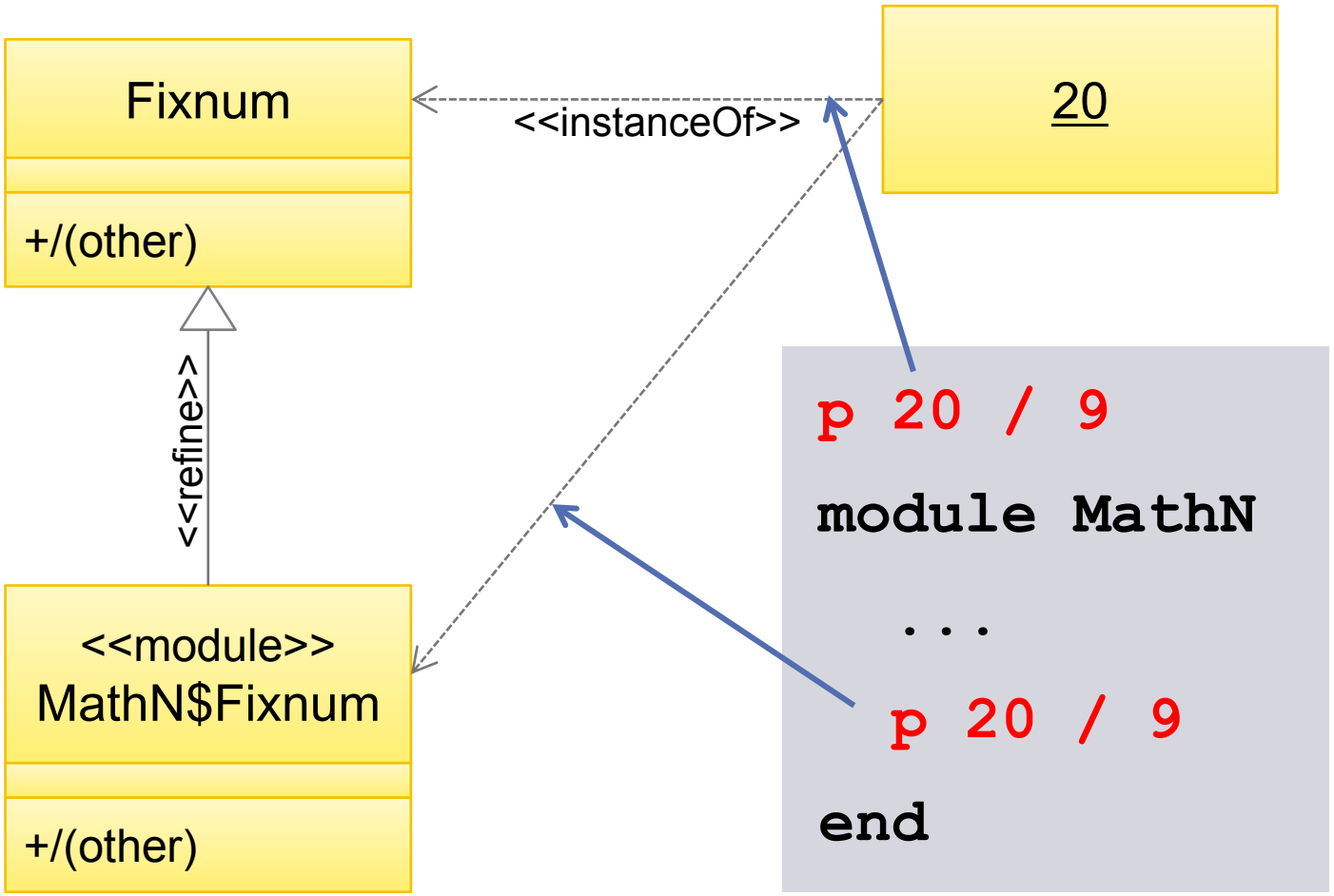
A refinement is a module

Module#refine creates an anonymous module called refinement

```
module MathN
  refine Fixnum do
    # self is a created refinement here
  end
```

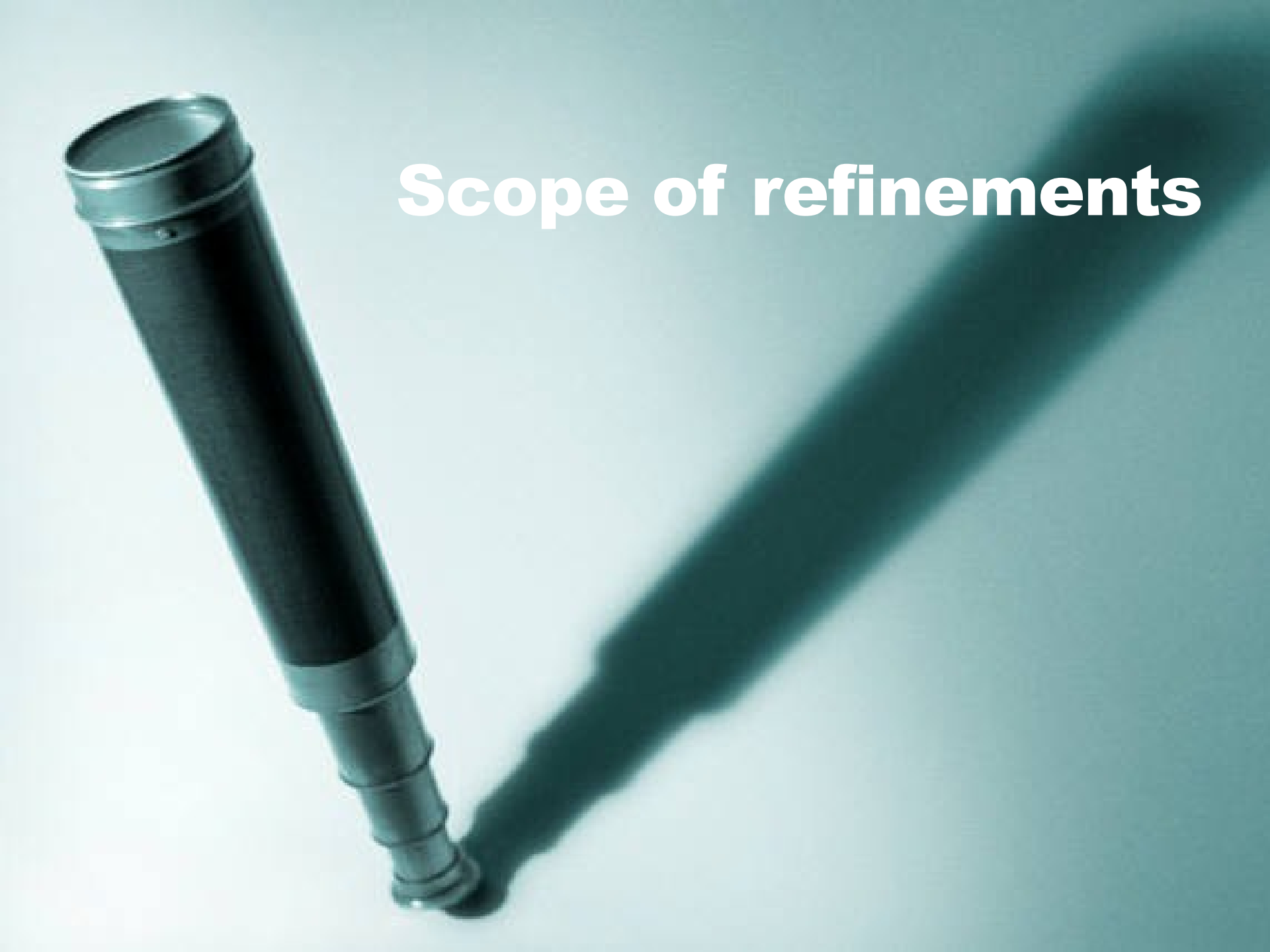
MathN$Fixnum denotes the refinement in this presentation (not valid Ruby code)

Relationship between a class and its refinement



Multiple refinements in a single module

```
module ToJSON
  refine Fixnum do
    def to_json; ... end
  end
  refine String do
    def to_json; ... end
  end
end
```


Scope of refinements

Using refinements in other modules

```
module MyModule
  using ToJSON
  p "foo".to_json #=> "\"foo\""
end
p "foo".to_json   #=> NoMethodError
```

Using refinements in toplevel

foo.rb

```
using ToJSON
p "foo".to_json #=> "\"foo\""
load("bar.rb")
```

bar.rb

```
p "bar".to_json #=> NoMethodError
```

Lexical scope

```
def foo
  p 1 / 2
end
module Bar
  using MathN
  p 1 / 2 #=> (1/2)
  foo     #=> 0
end
```

Why lexically scoped?

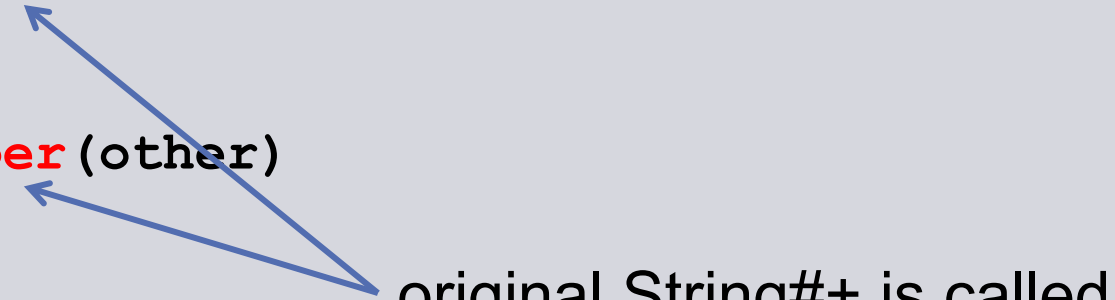
Not to break existing code

- Third party libraries might expect the original behavior of Fixnum#/

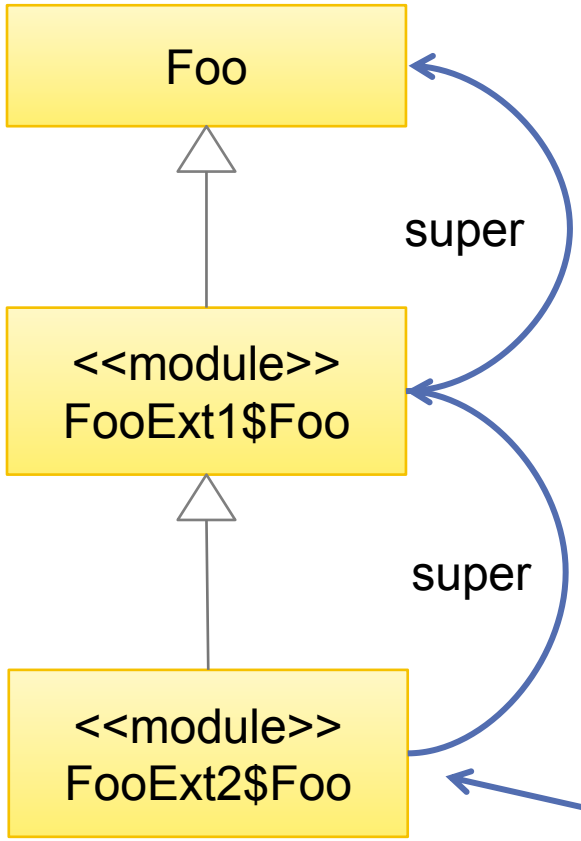

Stacking classes
and refinements

super in refined methods

```
refine String do
  def +(other)
    if encoding == Encoding::ASCII_8BIT
      super(other.force_encoding("ASCII-8BIT"))
    else
      super(other)
    end
  end
end
```

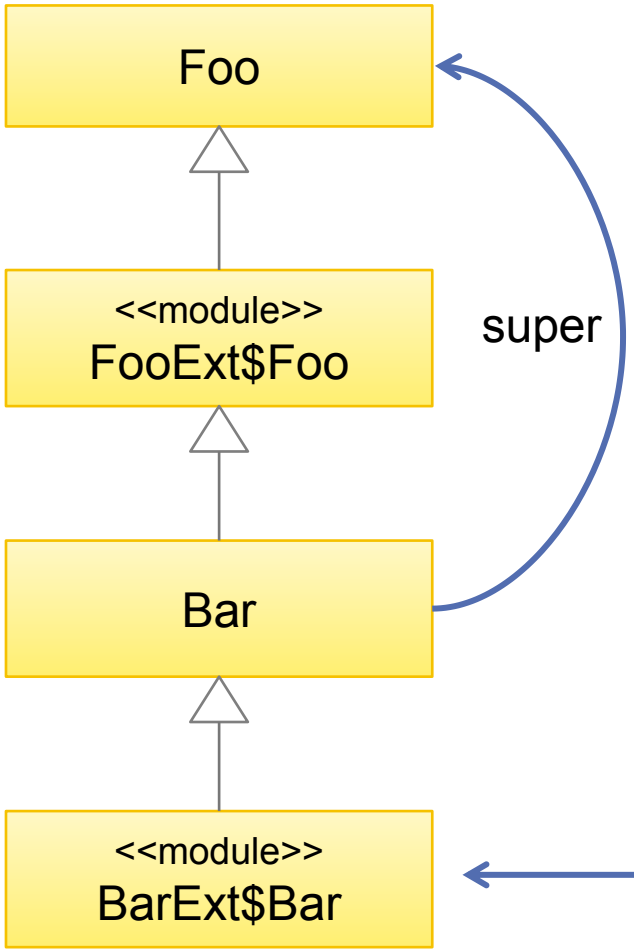


Linearization of refinements




```
class Foo; end
module FooExt1
  refine Foo do ... end
end
module FooExt2
  refine Foo do ... end
end
module FooExtUser
  using FooExt1
  using FooExt2
  Foo.new.do_something
end
```

Subclassing and refinements




```
class Foo; end
class Bar < Foo; end
module FooExt
  refine Foo do ... end
end
module BarExt
  refine Bar do ... end
end
module FooBarExtUser
  using FooExt
  using BarExt
  Bar.new.do_something
end
```

Inheritance of refinements

```
module ActiveRecord
  class Base
    using ActiveSupport::Refinements
  end
end
class Article < ActiveRecord::Base
  # ActiveSupport::Refinements is
  # available here
end
```

Mix-in and refinements

```
module MyHelper
  using ActiveSupport::Refinements
end
class Article
  include MyHelper
  # ActiveSupport::Refinements is
  # NOT available here
end
```


Metaprogramming
with refinements

module_eval

```
MathN.module_eval {p 1 / 2} #=> (1/2)
module Foo
  using MathN
end
Foo.module_eval {p 1 / 2} #=> (1/2)
```

Get refinements from modules

```
module ToJSON
  refine Fixnum do ... end
  refine String do ... end
end


p ToJSON.refinements
#=> {Fixnum=>#<Module:0x22528f4c>,
String=>#<Module:0x22528ed4>}
```

Using refinements globally

```
def using_globally(mod)
  mod.refinements.each do |c, r|
    c.send(:prepend, r)
  end
end
using_globally ToJSON
p 123.to_json #=> "123"
```


Use cases of
Refinements

Incompatible extensions

Refinements can break compatibility safely

- Refinements are lexically scoped

rationalize

Refinement version of mathn

```
require "rationalize"
module Foo
  using Rationalize
  p 1 / 2 #=> (1/2)
end
p 1 / 2 #=> 0
```

Backward compatibility

String#lines will be changed to return an Array instead of an Enumerator (#6670)

```
module StringBackwardCompat
  refine String do
    def lines(*args, &block)
      each_line(*args, &block)
    end
```

Case expression elimination

Case expressions should be replaced with dynamic dispatching

However, it's not good to add methods to built-in classes

Refinements can be used in such cases

Example of a case expression

From xmlrpc/create.rb:

```
def conv2value(param)
  val = case param
  when Fixnum, Bignum
    ...
  when TrueClass, FalseClass
    @writer.tag("boolean",
                param ? "1" : "0")
  ...
```

Examle of case expression elimination

```
module XMLRPC
  [TrueClass, FalseClass].each do |c|
    refine c do
      def to_xml(writer)
        writer.tag("boolean",
                   self ? "1" : "0")
      end
    end
  end
```

Internal DSL

Internal DSLs often extend built-in classes

Refinements can be used to extend built-in classes only in DSL contexts

rspec-refinements

Provides methods like should by refinements

```
class Foo; def tes; should; end; end

describe Foo do
  it { should be_a Foo }
  specify {
    expect { Foo.new.tes }.to \
      raise_error(NameError)
  }
end
```

activerecord-refinements

Provides operators in where conditions by refinements

```
User.where { :name == "matz" }.first
User.where { :age >= 20 }.all
:age >= 20 #=> ArgumentError
```


Are refinements slow?

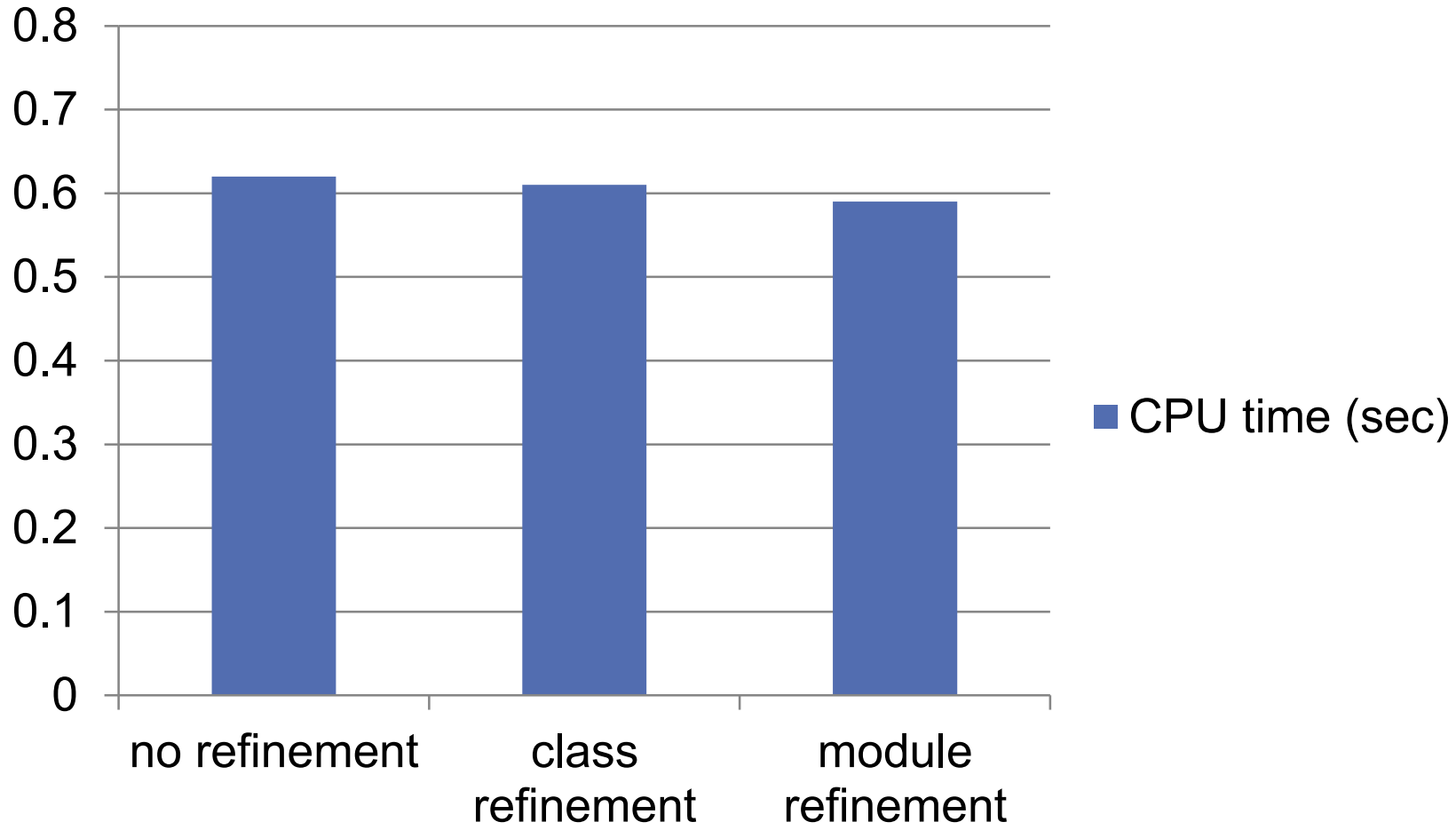
Benchmark code

```
module M; def foo; end; end
class C; include M; def bar; end; end
module MExt
  refine M do def foo; end end
end
module CExt
  refine C do def bar; end end
end
```

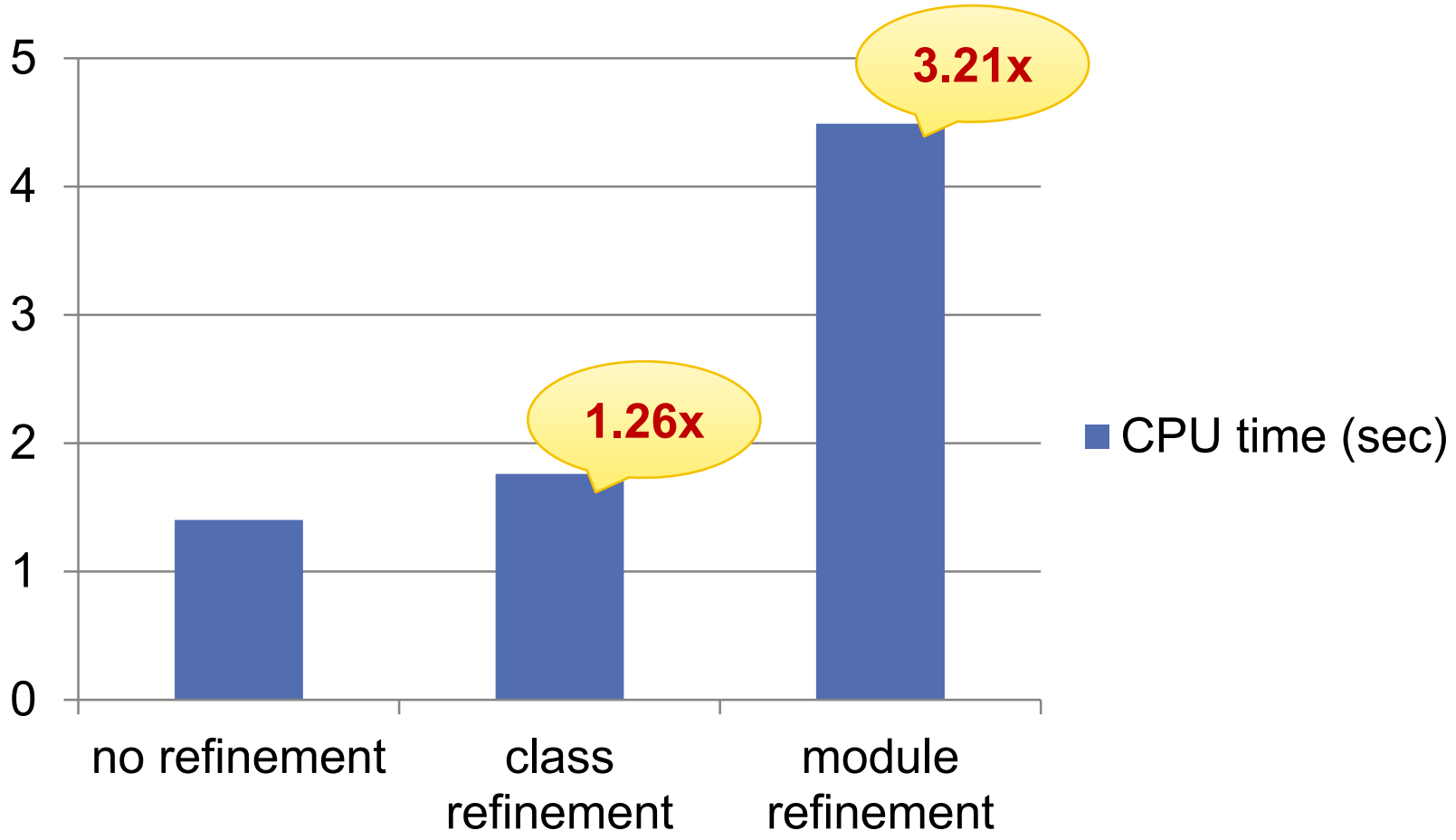
Benchmark code (cont'd)

```
N = 10000000
def run(bm, name, mod, obj, mid)
  call_100_times = 100.times.map { "obj.#{mid}" }.join("; ")
  bm.report(name) {
    mod.module_eval("#{N / 100}.times { #{call_100_times} }")
  }
end
Benchmark.bmbm do |bm|
  c = C.new
  run(bm, "no refinement", Object, c, "bar")
  run(bm, "class refinement", CExt, c, "bar")
  run(bm, "module refinement", MExt, c, "foo")
end
```

Benchmark result



Benchmark result (without method cache)



Conclusion

Refinements are extensible and modular class extensions

Previous works

Selector namespace

- SmallScript, ECMAScript 4

Classbox

- Smalltalk, Java

import-module, class-in-state

- Ruby

Thank you!