Using Modules to Organize and Reuse Functionality

Alex Korban
AUTHOR, DEVELOPER
@alexkorban korban.net

Overview

Modules are collections of methods and constants

Namespacing and nesting modules and classes

Include modules in your classes

Use modules for instance and class methods

Control the order of method lookup

Hook methods

A module is a collection of methods and constants. A module cannot be instantiated.

Module Functions

Namespacing

Organize code

Refinements

Add or modify class functionality

Mixins

Reuse code and build up class functionality

Namespacing

```
module Log

  APP_PREFIX = "LOG"

  def self.error(msg)
    puts "[#{APP_PREFIX}] ERROR: #{msg}"
  end

  def self.info(msg)
    puts "[#{APP_PREFIX}] INFO: #{msg}"
  end

end
```

Module Expressions

Can contain arbitrary code, just like class expressions

You can define module-level variables

```ruby
class Logger
  def log(prefix, msg)
    puts "#{prefix}: #{msg}"
  end
end
```

Logger Class

Module-level Variables

```
module Log
  APP_PREFIX = "LOG"

  @logger = Logger.new

  def self.error(msg)
    @logger.log("[#{APP_PREFIX}] ERROR", msg)
  end

  def self.info(msg)
    @logger.log("[#{APP_PREFIX}] ERROR", msg)
  end
end
```

Constants and module methods can be made private with *private_constant* and *private_class_method*

Instance Variables

Modules can include attribute accessor definitions

Methods can contain code to get/set instance variables

Instance variables are created in the object which the methods are called on

Instance Variables

```
module Tagged
  def tag(tag)
    @tags ||= []
    @tags << tag
  end

  def untag(tag)
    @tags.delete(tag) if !@tags.nil?
  end

  attr_reader :tags
end
```

All instance variables share the same namespace within a class, so you have to be mindful of name clashes

Nested Modules and Classes

```
module Libra
  module Log
    module LogHelpers
      def truncate
        # ...
      end
    end
  end
end

Libra::Log::LogHelpers::truncate
```

```
Helpers = Libra::Log::LogHelpers
Helpers::truncate
```

Module Names Are Constants

Nested Modules and Classes

```
module Libra
  class Collection
  end
end

c = Libra::Collection.new
```

Nested Modules and Classes

```
module Libra
  class Collection
    module Utils
      class CollectionHelper
        def self.cleanup
        end
      end
    end
  end
end

Libra::Collection::Utils::CollectionHelper.cleanup
```

Mixins

- Mix in modules to add functionality to classes
- A mixin's methods and constants become part of the class
- A class can include multiple mixins
- Simplify class hierarchies and write loosely coupled code

Mixins

```
module Tagged
  def tag(tag)
    @tags ||= []
    @tags << tag
  end

  def untag(tag)
    @tags.delete(tag) if !@tags.nil?
  end

  attr_reader :tags
end
```

```
class Collection
  include Tagged
end
```

Mix in a Module into a Class

Enumerable

Searching

Sorting

Traversal

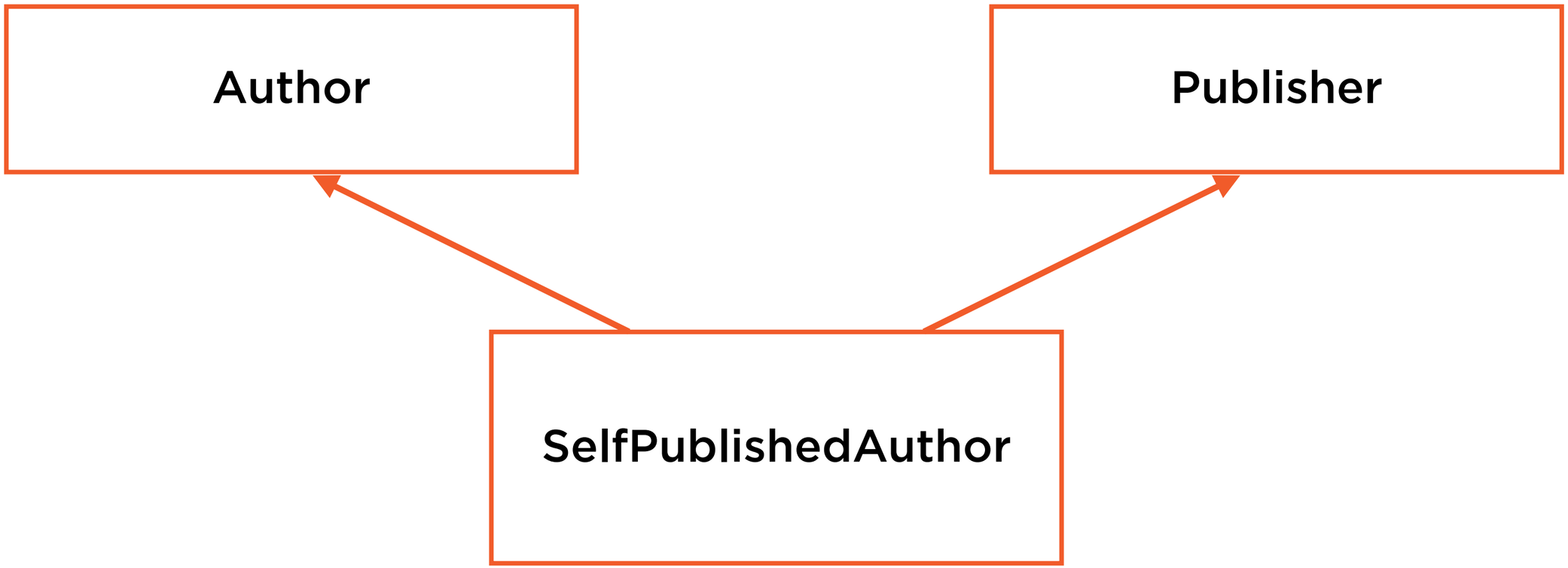
Filtering

Multiple Mixins

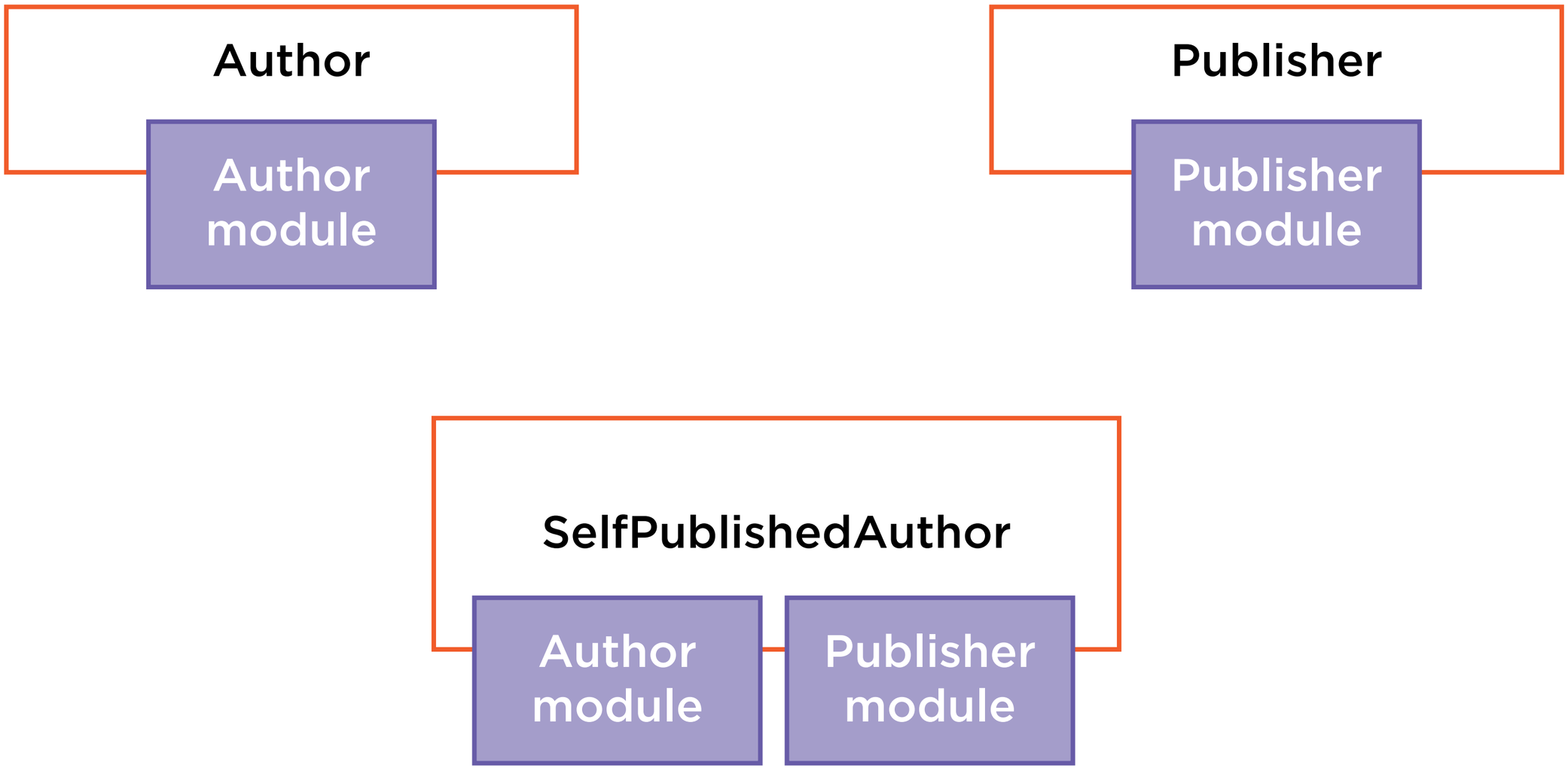
```ruby
class Collection
  include Tagged
  include Enumerable

  def each(&block)
    @books.each {|book| block.call(book) }
  end
end
```

```ruby
class Book
  def <=>(other)
    name <=> other.name
  end
end
```

Mixins



Mixins


Author
Author module
Publisher
Publisher module
SelfPublishedAuthor
Author module
Publisher module

```
def self.find_by_tags(tagged_collection, tags)
  tagged_collection.filter {|c| tags & c.tags == tags }
end
```

Search Collections by Item Tags

Search Collections by Item Tags

```
module TaggedFind
  def find_by_tags(tagged_collection, tags)
    tagged_collection.filter {|c| tags & c.tags == tags }
  end
end

class Book
  extend TaggedFind
end

Book.find_by_tags(collection.books, ["ruby", "testing"])
```

Both *extend* and *include* act on instance methods in a module

Combine Class and Instance Methods

```
module Tagged
  def tag(tag)
    @tags ||= []
    @tags << tag
  end

  def untag(tag)
    @tags.delete(tag) if !@tags.nil?
  end

  module ClassMethods
    def find_by_tags(tagged_collection, tags)
      tagged_collection.filter {|c| tags & c.tags == tags }
    end
  end
end
```

```ruby
class Book
  include Tagged
  extend Tagged::ClassMethods
end

Book.find_by_tags(collection.books, ["ruby", "testing"])
```

Combine Class and Instance Methods

Singleton Methods in a Module

```
module AccountMgmt
  def cancel_account!
    puts "Account cancelled for #{name}"
  end

  def update_billing(details)
    @billing_details = details
  end
end

current_user = User.new
current_user.extend AccountMgmt
```

Hook Methods

```
module Tagged
  def tag(tag); @tags ||= []; @tags << tag; end
  def untag(tag); @tags.delete(tag) if !@tags.nil?; end

  module ClassMethods
    def find_by_tags(tagged_collection, tags)
      tagged_collection.filter {|c| tags & c.tags == tags }
    end
  end

  def self.included(base)
    base.extend(ClassMethods)
  end
end
```

```ruby
class Book
  include Tagged
end

Book.find_by_tags([b1, b2], ["testing", "ruby"])  # This works now
```

Hook Methods

Other Hook Methods

extended	method_added	method_undefined
prepended	method_removed	inherited

Measuring Method Execution Time

```
class Collection
  def find_by_author(author)
    puts "in find_by_author"
  end

  # pass a block for custom sorting
  def custom_sort
    puts "in custom_sort"
    yield
  end

  log_time :find_by_author
  log_time :custom_sort
end
```

Measuring Method Execution Time

```
def self.log_time(method)
  alias_method "_original_#{method}".to_sym, method

  define_method(method) {| *args, &block |
    start_time = Time.now

    puts "Calling #{method} with args #{args.inspect} #{'and a block' if block}"
    result = __send__ "_original_#{method}".to_sym, *args, &block

    end_time = Time.now - start_time
    puts "Call to #{method} with args #{args.inspect} took #{end_time}s"
    result
  }
end
```

prepend allows you to include a module in such a way that its methods are looked up *before* the methods of the class itself

Measuring Method Execution Time

```
module LogTime
  module ClassMethods
    def log_time(method)
      # ...
    end
  end

  def self.included(base_class)
    base_class.extend(ClassMethods)
    log_time_module = const_set("#{base_class}LogTime", Module.new)
    base_class.prepend(log_time_module)
  end
end
```

Measuring Method Execution Time

```
module ClassMethods
  def log_time(method)
    LogTime.const_get("#{self}LogTime").define_method(method) {|*args, &block|
     start_time = Time.now

     puts "Calling #{method} with args #{args.inspect} #{'and a block' if block}"
     result = super(*args, &block)

     end_time = Time.now - start_time
     puts "Call to #{method} with args #{args.inspect} took #{end_time}s"
     result
    }
  end
end
```

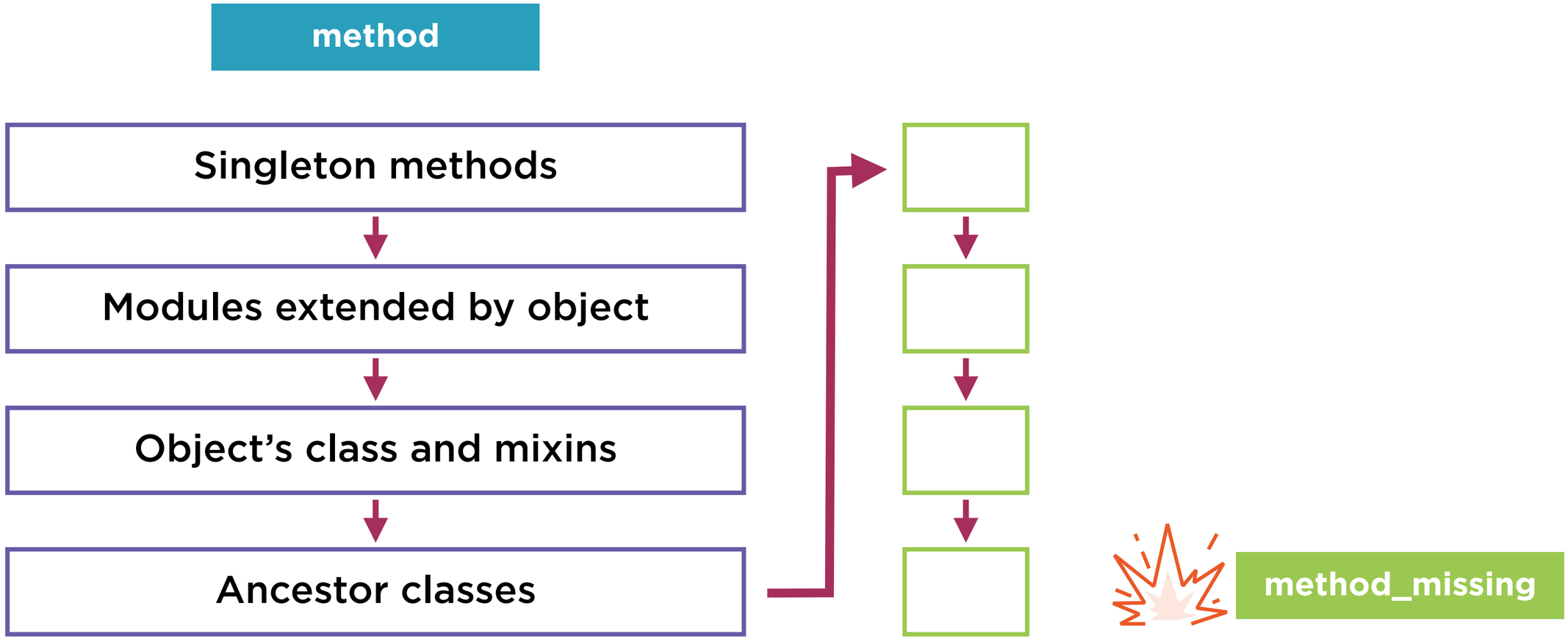
Measuring Execution Time

log_time **calls can appear before method definitions**

alias_method **requires methods to be defined first**

It's only a proof of concept to show *prepend*

Method Lookup



Summary

A module is a collection of methods and constants

Modules are used for namespacing and refinements

Modules are also the basis for mixins, an alternative to multiple inheritance and interfaces

Creating modules with methods and constants

Organizing modules and classes into hierarchies

Mixing modules into classes

Altering the order of method lookup

Using hook methods