Modelling Data Hierarchies with Inheritance

Alex Korban
AUTHOR, DEVELOPER

@alexkorban korban.net

Overview

Code reuse and organization with inheritance

Overriding parent class methods and using the *super* keyword

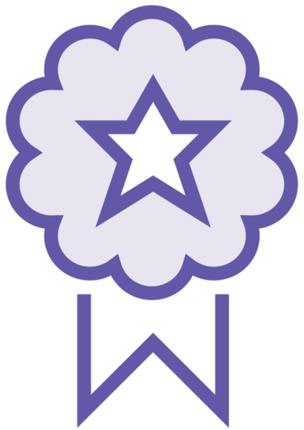
Restricting the visibility of methods in a class

Understanding the role of inheritance

Specialized Classes

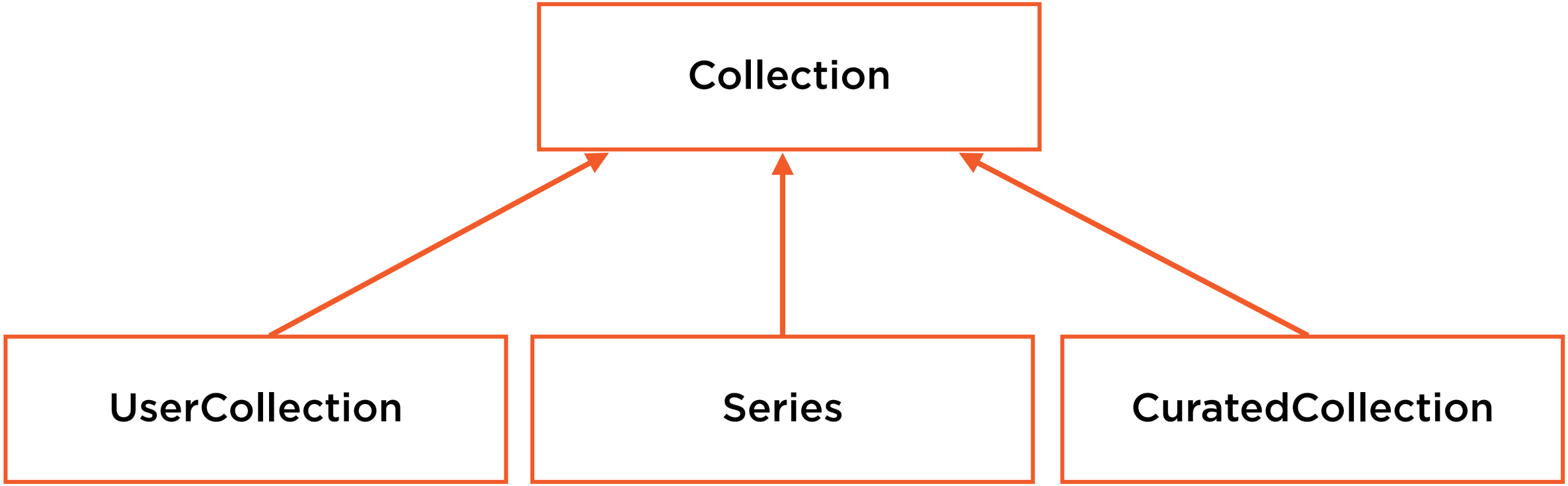
User Collection

Series



Curated Collection

Specialized Classes



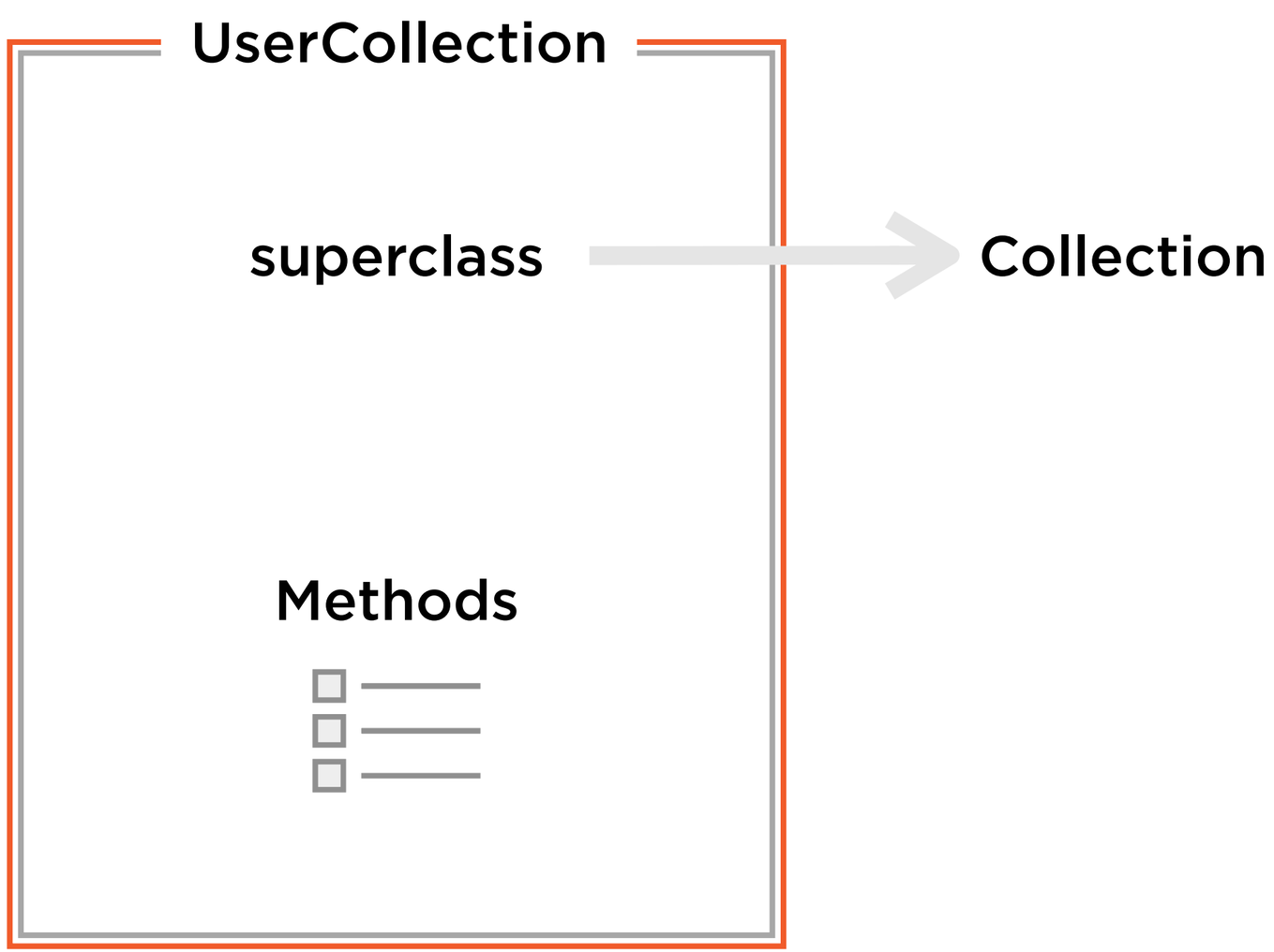
Child **Parent**

Subclass **Superclass**

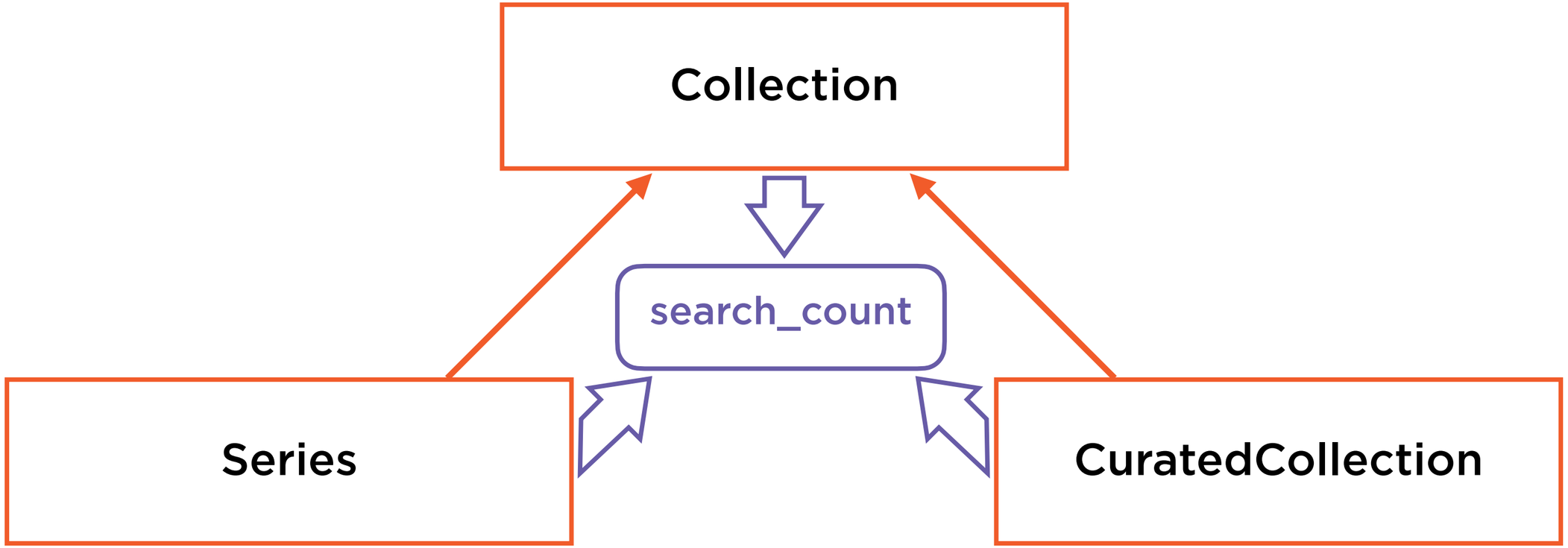
```
class UserCollection < Collection
end
```

Inheritance

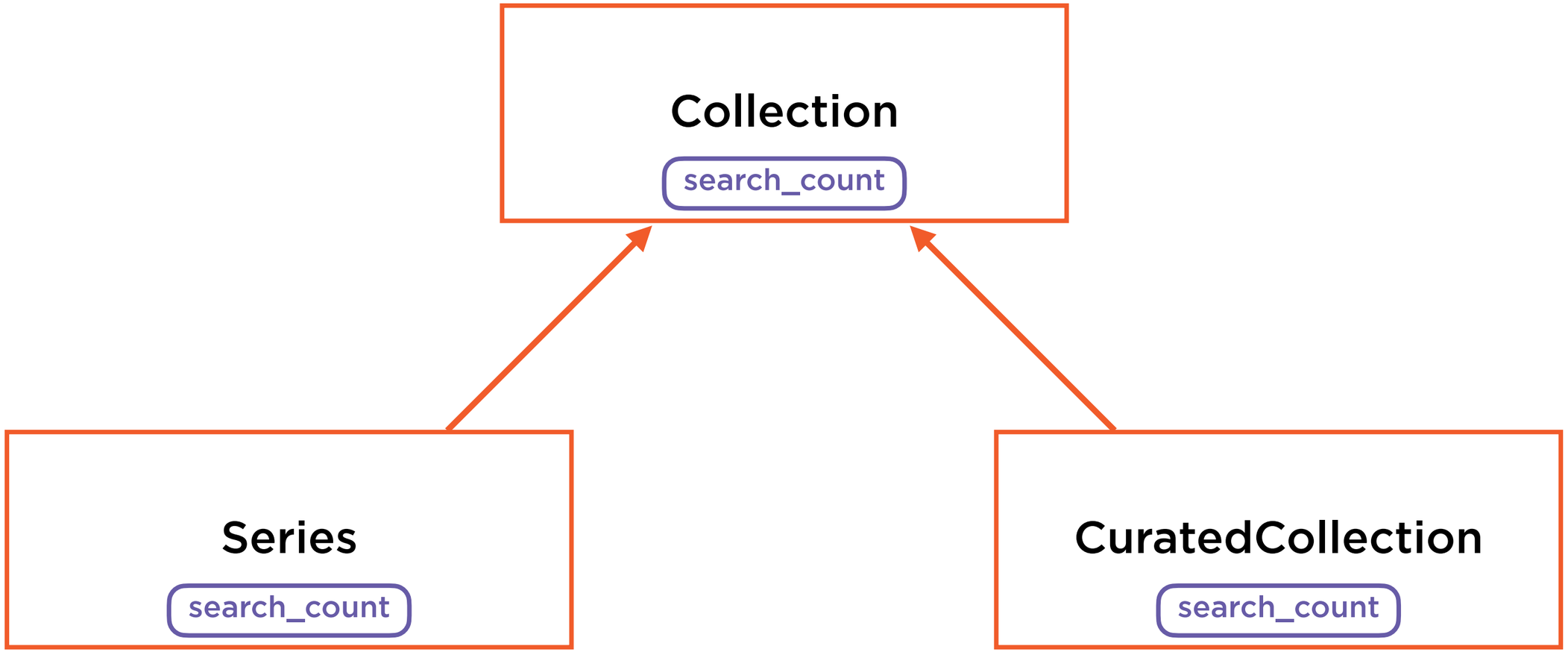
Superclass



Class Variables Are Shared with Subclasses

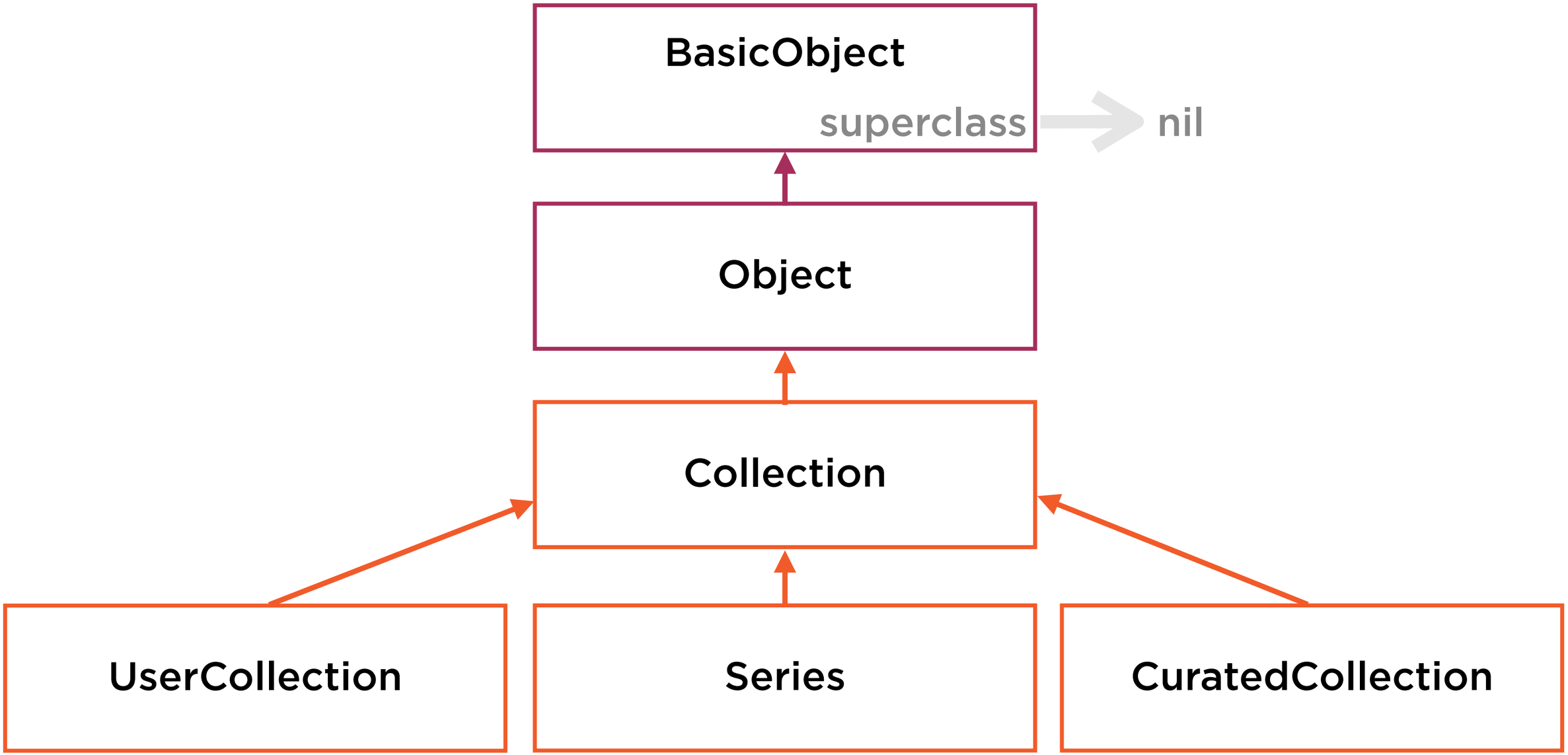


Class Instance Variables Are Defined Per Class

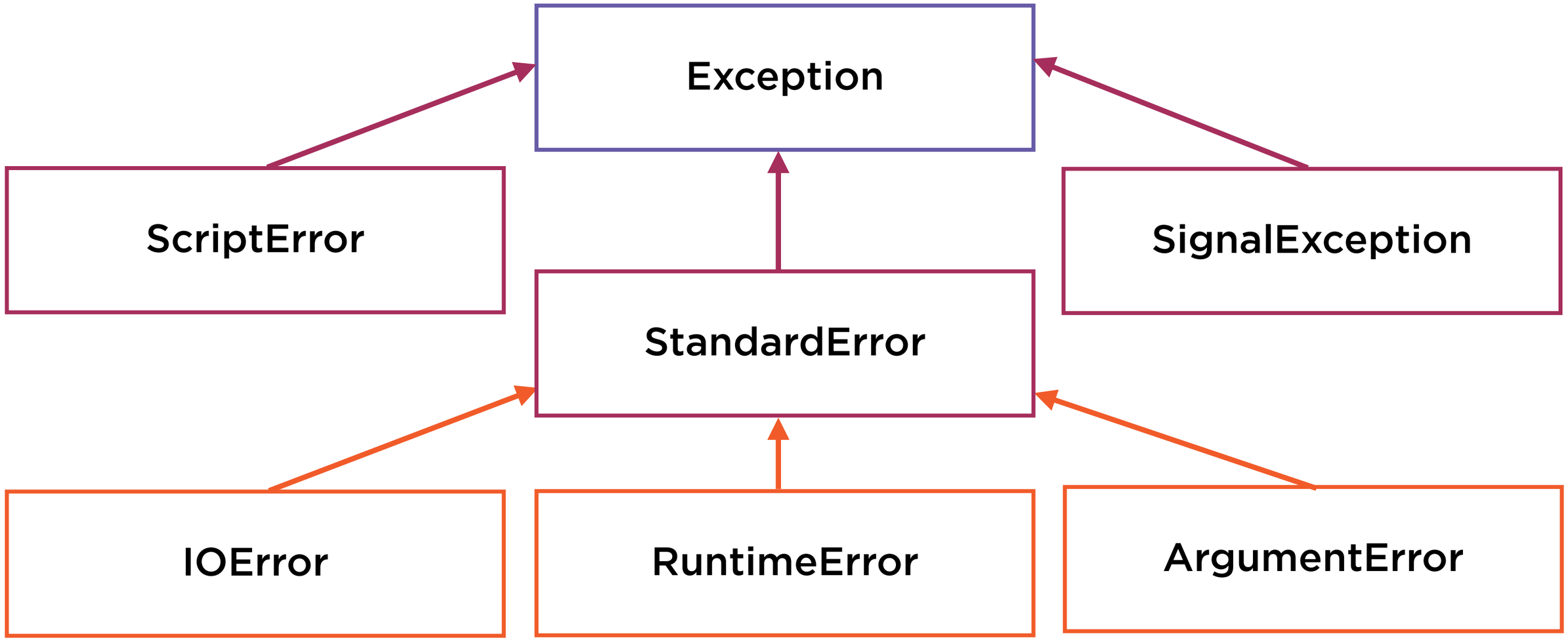


Classes inherit from the Object class by default

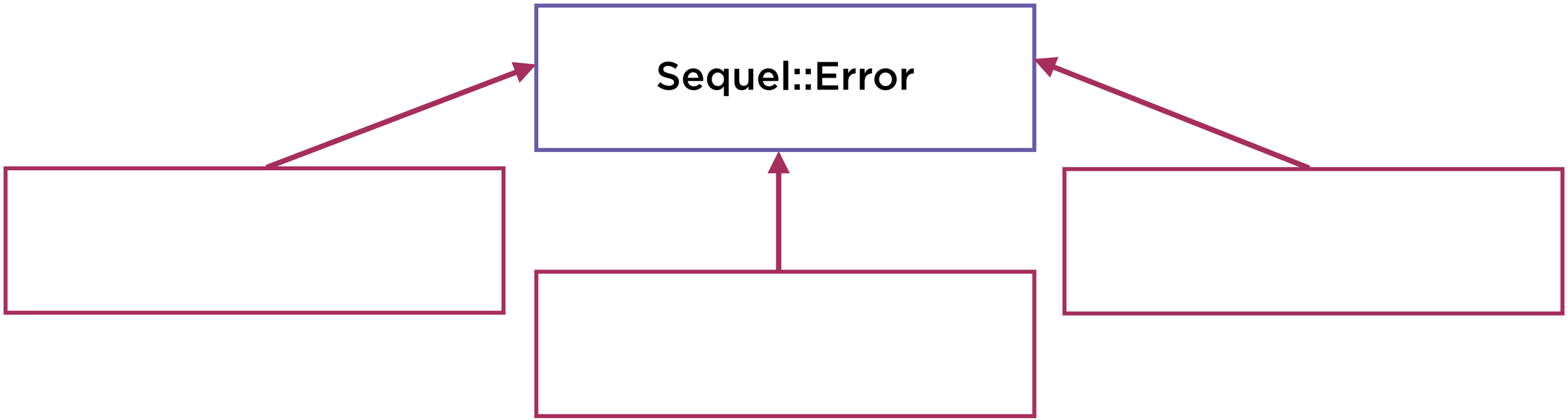
Object and BasicObject



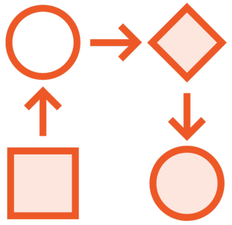
Exception Class Hierarchy



Exception Class Hierarchy



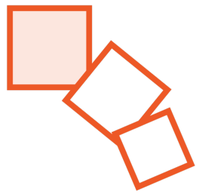
Overriding Parent Class Methods



Add or redefine functionality of the superclass in a subclass

Method call:

method_missing



Similar process for class methods and constants

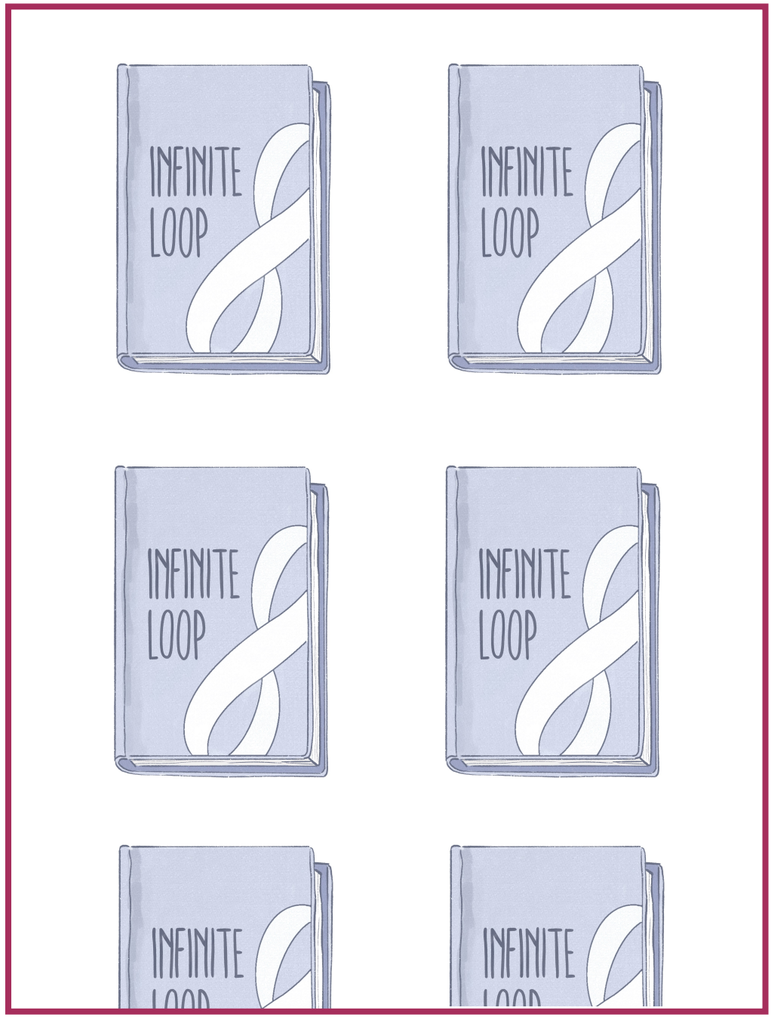
```
class Series < Collection
  def display(format)
    display(format)
  end
end
```

super Keyword

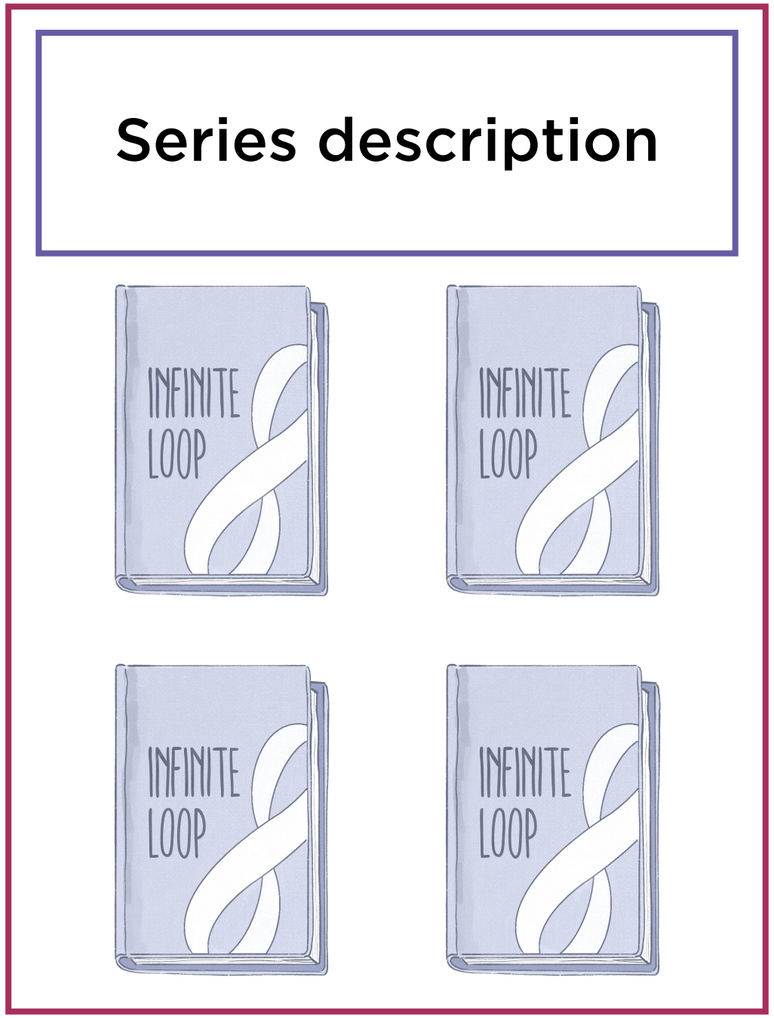
Call a method of the superclass

super

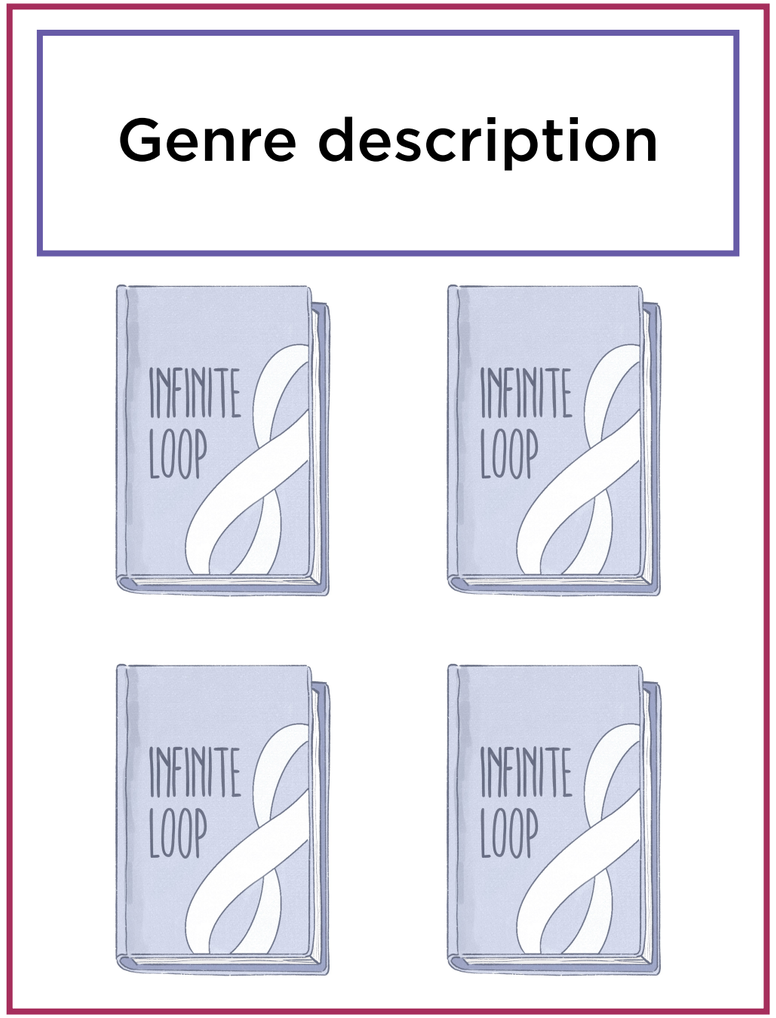
UserCollection



Series



CuratedCollection



super

```ruby
class Collection
  def initialize(name)
    @name = name
  end

  def display(format)
    puts "== #{@name} ==="
    puts "Showing books in a #{format} view"
  end
end
```

```ruby
class Series < Collection
  def display(format)
    puts "Series description"
    super
  end
end
```

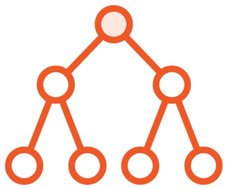
super

```
class CuratedCollection < Collection
  def initialize(name, genre)
    super(name)
    @genre = genre
  end

  def display(format, show_description:)
    @genre.display() if show_description
    super(format)
  end
end
```

initialize

Superclass *initialize* is called if *initialize* isn't overridden



Call *super* in *initialize* to ensure initialization up the chain

super

```
class Series < Collection
  def display(format)
    puts "Series description"
    super()
  end
end
```

Method Visibility

Methods are public by default

Member visibility can be controlled

Visibility rules in Ruby are different

Member Visibility

Maintain encapsulation and data hiding

Maintain internal invariants

Provide a clear API

Change implementation without affecting the public API

User Types

User

Author

Publisher

Authorization

`db_role`	**restricted visibility**
`db_plan`	**restricted visibility**
`is_authorized_for?(page)`	**public**

Method Visibility

private

protected

```ruby
class User
  def db_role
    # get the role
  end

  private :db_role
end
```

```ruby
user.db_role
```

Private Methods

Private Methods

```
class User
  private

  def db_role
    # get the role
  end
end
```

```ruby
class User
  def initialize(id)
    @id = id
  end

  private

  def db_role
    # get the role
  end

  public

  def is_authorized_for?(page)
    # report authorisation for `page`
  end
end
```

◀ public method

◀ private method

◀ public method

```ruby
class User
  def is_authorized_for?(page)
    # report authorisation for `page`
  end

  def db_role
    # get the role
  end

  def db_plan
    # get the plan
    # for the user's account
  end

  private :db_role, :db_plan
end
```

- methods are marked private after definition

Visibility specifiers are methods rather than keywords

```ruby
class Author < User
  def is_authorized_for?(page)
    if page.start_with?("author/")
      db_role == "author"
    else
      super
    end
  end
end
```

```ruby
user.db_role
```

Private Methods

Private methods can be called by subclasses

Private Class Methods

private doesn't work on class methods but you will not get an error

Use *private_class_method* instead

Pass it one name or a list of names

The counterpart is *public_class_method*

```ruby
class Collection
  private_class_method :new
end
```

Private Class Methods

```ruby
class User
  ROLES = ["user", "author", "publisher"]
  private_constant :ROLES
end
```

Private Constants

A protected method can be called on another object by an instance of the same class or one of its subclasses.

```
class Collection
  def ==(other)
    name == other.name
  end
end
```

Protected Methods

Protected Methods

```
class Collection
  def ==(other)
    id == other.id
  end

  protected
  attr_reader :id
end
```

Method Visibility

Methods are public by default

private methods can be called from subclasses

private_class_method for class methods

private_constant for constants

protected allows other objects of the same class to use the method

Limitations of Inheritance

Limited role compared to other languages

Main purpose is code reuse, not enforcing interfaces

Polymorphism can be achieved without inheritance via duck typing

Limitations of Inheritance

```
exporter = if export_format == :csv
  CSVExporter.new(current_user)
else
  JSONExporter.new(current_user)
end

exporter.export(filename)
```

The class hierarchy can be simplified when you don't need classes whose only purpose is to specify an interface.

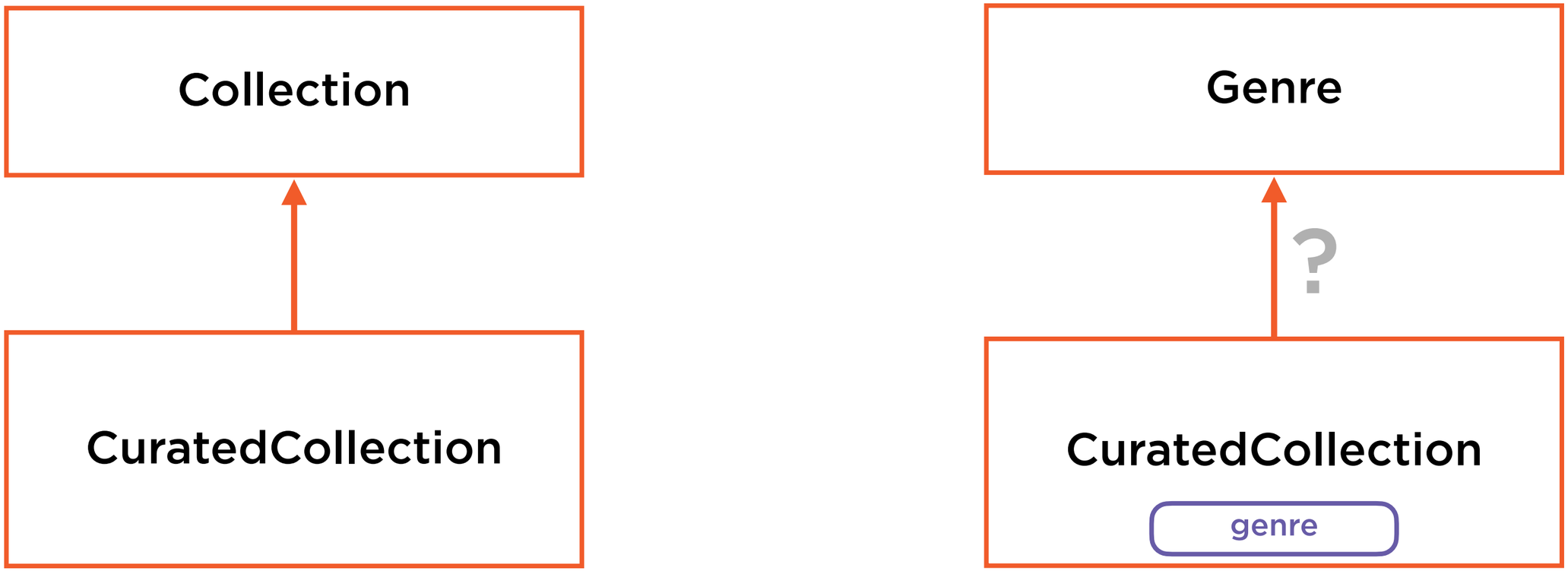
Code Reuse

Base class implements complex state manipulation

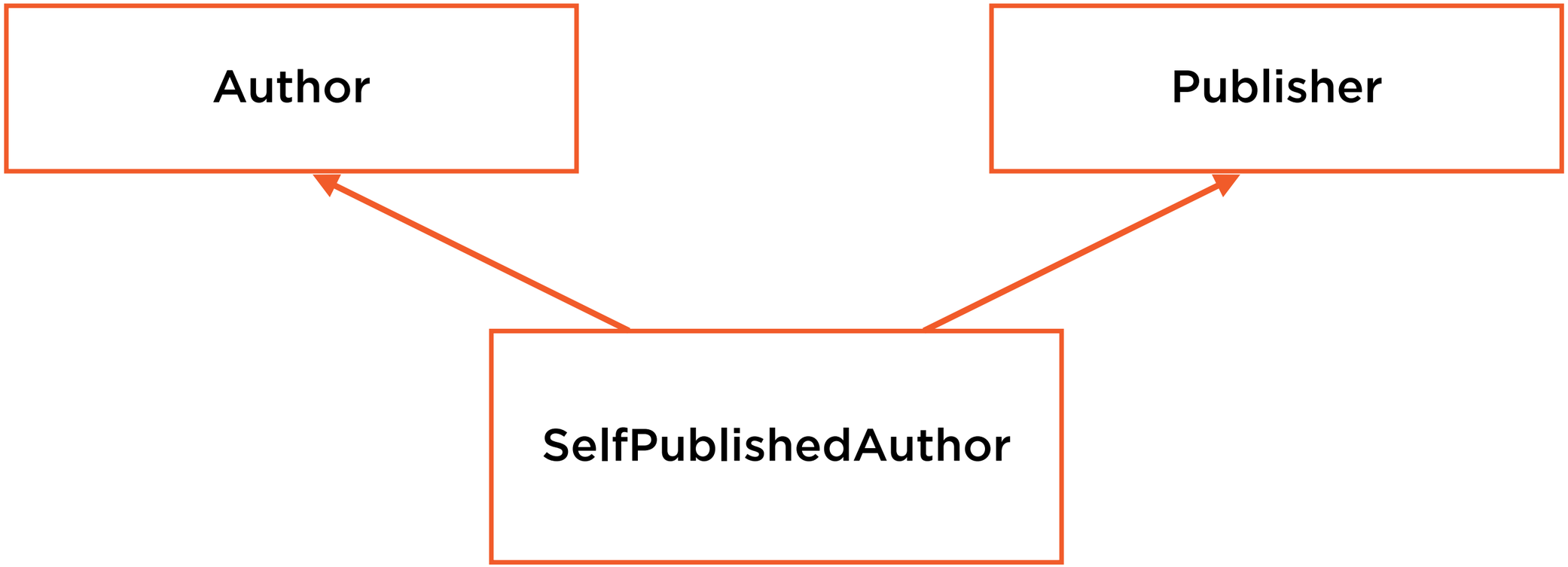
Subclasses modify only a few aspects of behavior

Inheritance should represent an *is-a* relationship

Modelling Relationships with Inheritance



Modelling Relationships with Inheritance



Summary

Create subclasses to specialize class behavior

Invoke superclass methods with *super*

Control the visibility of methods

Role of inheritance and its limitations