The Pragmatic Programmers

Programming Ruby 3.3

The Pragmatic Programmers' Guide

Noel Rappin
with Dave Thomas
Edited by Katharine Dvorak

The Facets of Ruby Series

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.prag-prog.com.

One of the principles of good software design is the elimination of unnecessary duplication. We work hard to make sure that each concept in our application is expressed only once in our code. Why? Because the world changes. And when you adapt your application to each change, you want to know that you've changed exactly the code you need to change. If each real-world concept is implemented at a single point in the code, this becomes vastly easier.

We've already seen how classes help reduce duplication. All the methods in a class are automatically accessible to instances of that class. But we want to do other, more general types of sharing. Maybe we're dealing with an application that ships goods. Many forms of shipping are available, but all forms share some basic functionality, perhaps weight calculation. We don't want to duplicate the code that implements this functionality across the implementation of each shipping type.

Or maybe we have a more generic capability that we want to inject into a number of different classes. For example, an online store may need the ability to calculate sales tax for carts, orders, quotes, and so on. Again, we don't want to duplicate the sales tax code in each of these places.

In this chapter, we'll look at two different but related mechanisms for this kind of sharing in Ruby. The first, *class-level inheritance*, is common in object-oriented languages. We'll then look at *mixins*, a technique that's often preferable to inheritance. We'll wind up with a discussion of when to use each.

Inheritance and Messages

In a previous chapter, we saw that when the `puts` method needs to convert an object to a string, it calls that object's `to_s` method. But we've also written our own classes that don't explicitly implement `to_s`. Despite this, instances of these classes respond successfully when we call `to_s` on them. How this works has to do with inheritance and how Ruby uses it to determine what method to run when you send a message to an object.

Inheritance allows you to create a class that's a specialization of another class. This specialized class is called a *subclass* of the original, and the original is a *superclass* of the subclass. People also refer to this relationship as *child* and *parent* classes.

The basic mechanism of subclassing is that the child inherits all of the capabilities of its parent class. All the parent's instance methods are available to instances of the child.

Let's look at a minimal example and then later build on it. Here's a definition of a parent class and a child class that inherits from it:

```
class Parent
  def say_hello
    puts "Hello from #{self}"
  end
end

p = Parent.new
p.say_hello

class Child < Parent
end

c = Child.new
c.say_hello
```

produces:

```
Hello from #<Parent:0x0000000100937780>
Hello from #<Child:0x0000000100936f60>
```

The parent class defines a single instance method, say_hello. We call that method by creating a new instance of the class, storing a reference to that instance in the variable p, and then using dot syntax, p.say_hello.

We then create a subclass using class Child < Parent. The < notation means we're creating a subclass of the thing on the right. The fact that we use the less-than sign is meant to signal that the child class is supposed to be a specialization of the parent.

Note that the child class defines no methods, but when we create an instance of it, we can call say_hello. That's because the child inherits all the methods of its parent. Also note that when we output the value of self—the current object—it shows that we're in an instance of class Child, even though the method we're running is defined in the parent.

The superclass method returns the parent of a particular class:

```
class Parent
end

class Child < Parent
end

Child.superclass # => Parent
```

But what's the superclass of Parent?

```
class Parent
end
Parent.superclass # => Object
```

If you don't define an explicit superclass when defining a class, Ruby automatically uses the built-in class Object as the class's parent. Let's go further:

```
Object.superclass # => BasicObject
```

Class BasicObject is a very, very minimal object that's used in certain kinds of metaprogramming, acting as a blank canvas. What's its parent?

```
BasicObject.superclass # => nil
```

So, we've finally reached the end. BasicObject is the root class of our hierarchy of classes. Given any class in any Ruby application, you can ask for its superclass, then the superclass of that class, and so on, and you'll eventually get back to BasicObject.

We've seen that if you call a method in an instance of class Child and that method isn't in Child's class definition, Ruby will look in the parent class. It goes deeper than that because if the method isn't defined in the parent class, Ruby continues looking in the parent's parent, the parent's parent's parent, and so on, through the ancestors until it runs out of classes. Method lookup in Ruby is actually a little bit more complex, we'll talk more about it in Method Lookup, on page ?.

And this explains our original question about to_s. We can work out why to_s is available in just about every Ruby object. to_s, it turns out, is defined in class Object. Because Object is an

ancestor of every Ruby class except BasicObject, instances of every Ruby class have a to_s method defined:

```
class Person
  def initialize(name)
    @name = name
  end
end

p = Person.new("Michael")
puts p
```

produces:

```
#<Person:0x0000000100e781b8>
```

We saw in the previous chapter that we can override the to_s method:

```
class Person
  def initialize(name)
    @name = name
  end

  def to_s
    "Person named #{@name}"
  end
end

p = Person.new("Michael")
puts p
```

produces:

```
Person named Michael
```

Armed with our knowledge of subclassing, we now know there's nothing special about this code. The puts method calls to_s on its arguments. In this case, the argument is a Person object. Because class Person defines a to_s method, that method is called. If it doesn't define a to_s method, then Ruby looks for (and finds) to_s in Person's parent class, Object.

It's common to use subclassing to add application-specific behavior to a standard library or framework class. If you've used Ruby on Rails,[1] you'll have subclassed ActionController::Base when writing your own controller classes. Your controllers get all the behavior of the base controller and add their own specific handlers to individual user actions.

Let's look at an example where inheritance can spare us a significant amount of duplication. Imagine you're working on a task-tracker application. A task might be in one of several states—it might be done, it might be started but incomplete, or it might be defined but not started. There may be other statuses, but just those three are probably enough to make the point.

If you're writing code that interacts with the tasks in this system, you'll likely have to take a task's status into account in your code. In other words, you'll likely be forever writing code like this:

```
def chatty_string(task)
  case task.status
```

1. http://www.rubyonrails.org

```
  when "done" then "I'm done"
  when "started" then "I'm not done"
  when "unstarted" then "I haven't even started"
  end
end
```

You'll be continually switching on the status of a task. This is a form of duplication. If the list of statuses changes, every one of these if statements or case statements would need to be updated. So it seems worth trying to reduce the number of times we use that switching logic.

We can use inheritance to create a hierarchy of status classes, and then only do our switching logic once:

tut_modules/status.rb

```
class Status
  def self.for(status_string)
    case status_string
    when "done" then DoneStatus.new
    when "started" then StartedStatus.new
    when "defined" then DefinedStatus.new
    end
  end

  def done? = false

  def chatty_string = raise NotImplementedError
end

class DoneStatus < Status
  def to_s = "done"

  def done? = true

  def chatty_string = "I'm done"
end
```

```
class StartedStatus < Status
  def to_s = "started"

  def chatty_string = "I'm not done"
end

class DefinedStatus < Status
  def to_s = "defined"

  def chatty_string = "I'm not even started"
end
```

Now, if we want to get at that particular chatty string, rather than having to do the case expression explicitly, we can write something like this:

```
Status.for(task.status).chatty_string
```

The case logic is now behind the scenes, in our Status.for method. Once we call it, we know what kind of status we have, and each kind of status knows its own behavior, so we can now call chatty_string directly on the status. More to the point, once we call Status.for, we don't need to have that case logic again; we've removed the potential duplication.

The done? method is defined in the parent class as being false, which is fine for the StartedStatus and DefinedStatus classes, but incorrect for the DoneStatus class, which therefore overrides done? to the correct value—true—for that class. There is no default for the chatty_string method though, so the parent class throws an exception if it's somehow called. This is a signal that all the subclasses must define this method.

This is a common idiom when using subclassing. A parent class assumes that it'll be subclassed and calls a method that it expects its children to implement. The parent takes on the brunt of the processing but also invokes what are effectively hook methods in subclasses to add application-level functionality. As we'll see at the end of this chapter, just because this idiom is common doesn't always make it a good design.

Instead, let's look at *mixins*, a different way of sharing functionality in Ruby code. But, before we look at mixins, we'll need to get familiar with Ruby *modules*.

In Ruby, a module can do everything that a class can do, except create instances. It turns out, that even without creating instances, it still can be useful to group related methods and data together. Let's explore how.