6

METHOD LOOKUP AND CONSTANT LOOKUP

As we saw in Chapter 5, classes play an important role in Ruby, holding method definitions and constant values, among other things. We also learned how Ruby implements inheritance using the `super` pointer in each `RClass` structure.

In fact, as your program grows, you might imagine it organized by class and superclass, creating a kind of giant tree structure. At the base is the `Object` class (or, actually, the internal `BasicObject` class). This class is Ruby's default superclass, and all of your classes appear somewhere higher up in the tree, branching out in different directions. In this chapter we'll study how Ruby uses this superclass tree to look up methods. When you write code that calls a method, Ruby looks through this tree in a very precise manner. We'll step through a concrete example to see the method lookup process in action.

Later in this chapter we'll learn another way to visualize your Ruby code. Every time you create a new class or module, Ruby adds a new scope to a different tree, a tree based on the syntactical structure of your program. The trunk of this tree is the top-level scope, or the beginning of your Ruby code file where you start typing. As you define more and more highly nested modules and classes, this tree would grow higher and higher as well. We'll learn how this syntax, or namespace, tree allows Ruby to find constant definitions, just as the superclass tree allows Ruby to find methods.

But before we get to method and constant lookup, let's get started with a look at Ruby modules. What are modules? How are they different from classes? What happens when you include a module into a class?

ROADMAP

How Ruby Implements Modules

As you may know, modules are very similar to classes in Ruby. You can create a module just as you create a class—by typing the `module` keyword followed by a series of method definitions. But while modules are similar to classes, they are handled differently by Ruby in three important ways:

- Ruby doesn't allow you to create objects directly from modules. In practice this means that you can't call the `new` method on a module because `new` is a method of `Class`, not of `Module`.
- Ruby doesn't allow you to specify a superclass for a module.
- In addition, you can include a module into a class using the `include` keyword.

But what are modules exactly? How does Ruby represent them internally? Does it use an `RModule` structure? And what does it mean to "include" a module into a class?

Modules Are Classes

As it turns out, internally Ruby implements modules as classes. When you create a module, Ruby creates another `RClass/rb_classext_struct` structure pair, just as it would for a new class. For example, suppose we define a new module like this.

```
module Professor
end
```

Internally, Ruby would create a class, not a module! Figure 6-1 shows how Ruby represents a module internally.

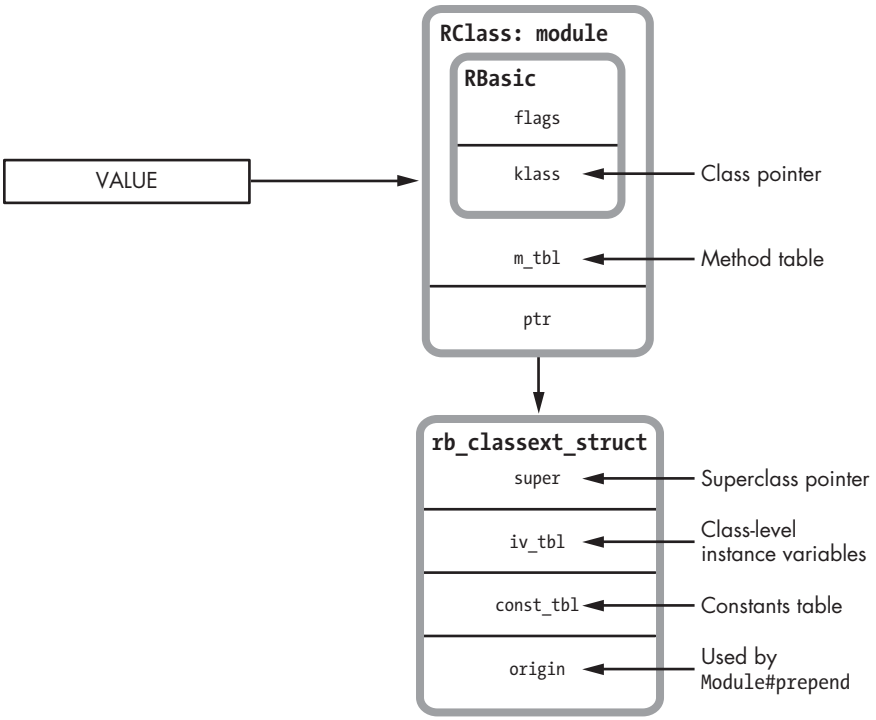

Figure 6-1: The portion of Ruby's class structures that's used for modules

In this figure I show Ruby's `RClass` structure again. However, I've removed some of the values from the diagram because modules don't use all of them. Most importantly, I removed `iv_index_tbl` because you can't create object instances of a module—in other words, you can't call the `new` method on a module. This means there are no object-level attributes to keep track of. I also removed the `refined_class` and `allocator` values because modules don't use them either. I've left the `super` pointer because modules do have superclasses internally even though you aren't allowed to specify them yourself.

A technical definition of a Ruby module (ignoring the `origin` value for now) might look like this:

> A Ruby module is a Ruby object that also contains method definitions, a superclass pointer, and a constants table.

Including a Module into a Class

The real magic behind modules happens when you include a module into a class, as shown in Listing 6-1.

```
module Professor
end
```

```
class Mathematician < Person
  include Professor
end
```

Listing 6-1: Including a module into a class

When we run Listing 6-1, Ruby creates a copy of the `RClass` structure for the `Professor` module and uses it as the new superclass for `Mathematician`. Ruby's C source code refers to this copy of the module as an *included class*. The superclass of the new copy of `Professor` is set to the original superclass of `Mathematician`, which preserves the superclass, or ancestor chain. Figure 6-2 summarizes this somewhat confusing state of affairs.

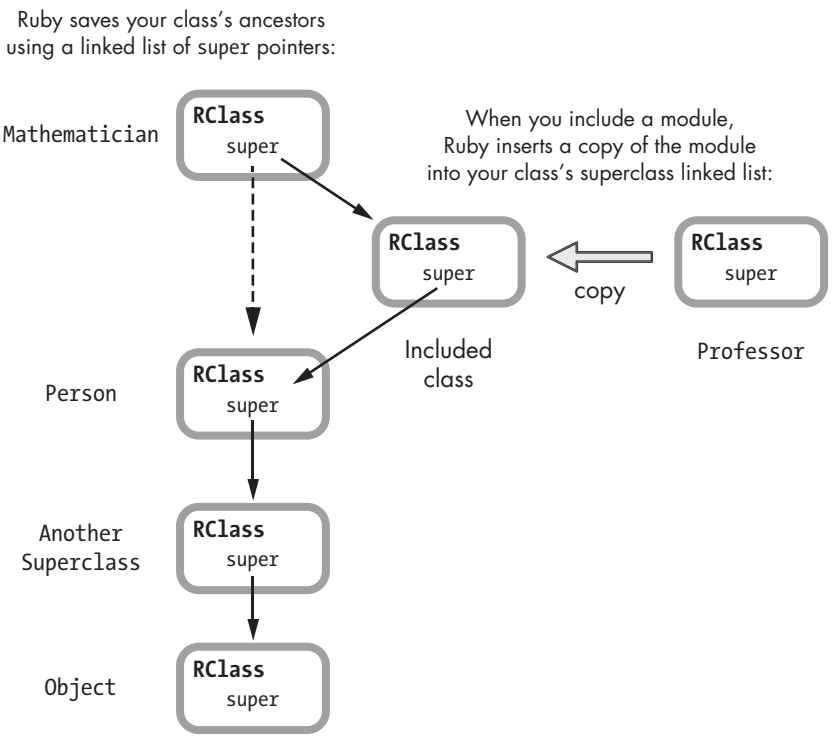

Figure 6-2: Including a module into a class

You can see the `Mathematician` class at the top-left corner of Figure 6-2. Below it and along the left side, you see its superclass chain: `Mathematician`'s superclass is `Person`, whose superclass is `Another Superclass`, and so on. The `super` pointer in each `RClass` structure (actually, each `rb_classext_struct` structure) points down to the next superclass.

Now to the `Professor` module on the right side of Figure 6-2. When we include this module into the `Mathematician` class, Ruby changes the `super` pointer of `Mathematician` to point to a copy of `Professor` and the `super` pointer of this copy of `Professor` to point to `Person`, the original superclass of `Mathematician`.

NOTE *Ruby implements* `extend` *in exactly the same way, except the included class becomes the superclass of the target class's class, or metaclass. Thus,* `extend` *allows you to add class methods to a class.*

Ruby's Method Lookup Algorithm

Whenever you call a method, whenever you "send a message to a receiver" to use object-oriented programming jargon, Ruby needs to determine which class implements that method. Sometimes this is obvious: The receiver's class might implement the target method. However, this isn't often the case. It might be that some other module or class in your system implements the method. Ruby uses a very precise algorithm to search through the modules and classes in your program in a particular order to find the target method. An understanding of this process is essential for every Ruby developer, so let's take a close look at it.

The flowchart in Figure 6-3 gives you a graphical picture of Ruby's method lookup algorithm.

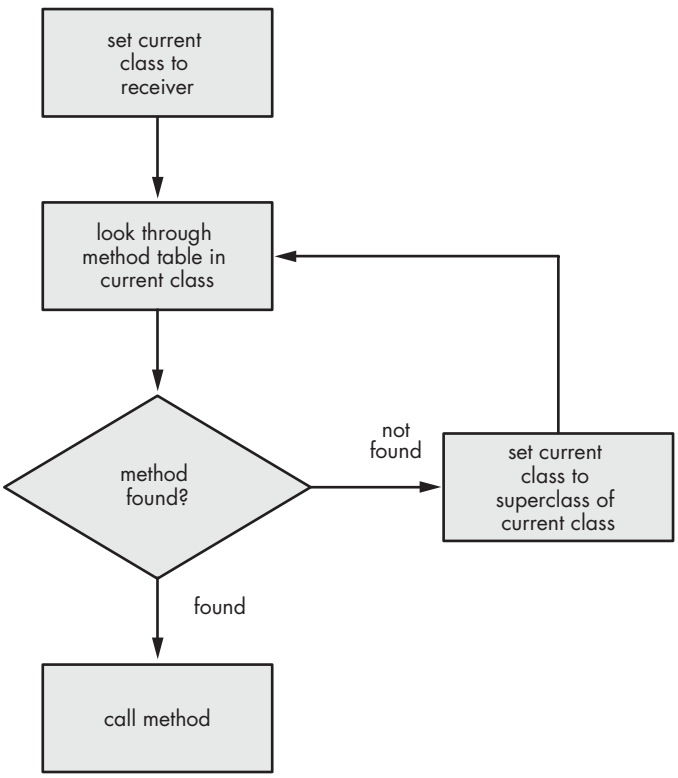

Figure 6-3: Ruby's method lookup algorithm

This algorithm is remarkably simple, isn't it? As you can see, Ruby simply follows the `super` pointers until it finds the class or module that contains

the target method. You might imagine that Ruby would have to distinguish between modules and classes using some special logic—that it would have to handle the case where there are multiple included modules, for example. But no, it's just a simple loop on the `super` pointer linked list.

A Method Lookup Example

In a moment we'll walk through this algorithm to be sure we understand it thoroughly. But first, let's set up an example we can use that has a class, a superclass, and a module. This will allow us to see how classes and modules work together inside of Ruby.

Listing 6-2 shows the `Mathematician` class with the accessor methods `first_name` and `last_name`.

```
class Mathematician
  attr_accessor :first_name
  attr_accessor :last_name
end
```

Listing 6-2: A simple Ruby class, repeated from Listing 5-1

Now let's introduce a superclass. In Listing 6-3, at ❶ we set `Person` as the superclass of `Mathematician`.

```
class Person
end

❶ class Mathematician < Person
  attr_accessor :first_name
  attr_accessor :last_name
end
```

Listing 6-3: `Person` is the superclass of `Mathematician`.

We'll move the name attributes to the `Person` superclass because not only mathematicians have names. We end up with the code shown in Listing 6-4.

```
class Person
  attr_accessor :first_name
  attr_accessor :last_name
end

class Mathematician < Person
end
```

Listing 6-4: Now the name attributes are in the `Person` superclass.

Finally, we'll include the `Professor` module into the `Mathematician` class at ❶. Listing 6-5 shows the completed example.

```
class Person
  attr_accessor :first_name
  attr_accessor :last_name
end

module Professor
  def lectures; end
end

class Mathematician < Person
❶   include Professor
end
```

Listing 6-5: Now we have a class that includes a module and has a superclass.

The Method Lookup Algorithm in Action

Now that we have our example set up, we're ready to see how Ruby finds a method we call. Every time you call any method in one of your programs, Ruby follows the same process we're about to see here.

To kick things off, let's call a method. Using this code, we create a new mathematician object and set its first name:

```
ramanujan = Mathematician.new
ramanujan.first_name = "Srinivasa"
```

To execute this code, Ruby needs to find the `first_name=` method. Where is this method? How does Ruby find it exactly?

First, Ruby gets the class from the `ramanujan` object via the `klass` pointer, as shown in Figure 6-4.

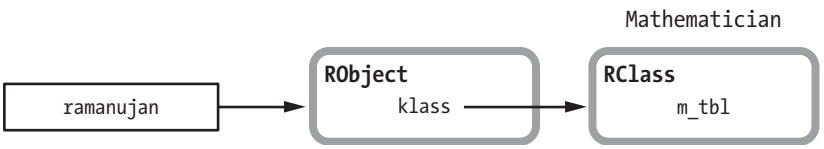

Figure 6-4: Ruby first looks for the `first_name=` method in the object's class.

Next, Ruby checks to see whether `Mathematician` implements `first_name=` directly by looking through its method table, as shown in Figure 6-5.

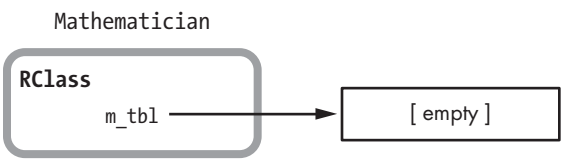

Figure 6-5: Ruby first looks for `first_name=` in the class's method table.

Because we've moved all of the methods down into the Person superclass, the first_name= method is no longer there. Ruby continues through the algorithm and gets the superclass of Mathematician using the super pointer, as shown in Figure 6-6.

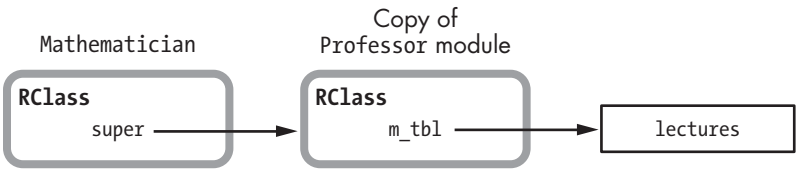

Figure 6-6: The superclass of Mathematician is the copy of the Professor module.

Remember, this is not the Person class; it's the *included* class, which is a copy of the Professor module. Because it's a copy, Ruby looks through the method table for Professor. Recall from Listing 6-5 that Professor contains only the single method lectures. Ruby won't find the first_name= method.

NOTE *Notice that because Ruby inserts modules above the original superclass in the superclass chain, methods in an included module override methods present in a superclass. In this case, if Professor also had a first_name= method, Ruby would call it and not the method in Person.*

Because Ruby doesn't find first_name= in Professor, it continues to iterate over the super pointers, but this time it uses the super pointer in Professor, as shown in Figure 6-7.

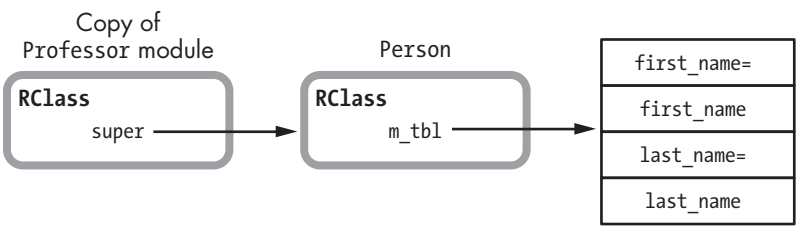

Figure 6-7: The Person class is the superclass of the included copy of the Professor module.

Note that the superclass of the Professor module—or more precisely, the superclass of the included copy of the Professor module—is the Person class. This was the original superclass of Mathematician. Finally, Ruby sees the first_name= method in the method table for Person. Because it has identified which class implements first_name=, Ruby can now call the method using the method dispatch process we learned about in Chapter 4.

Multiple Inheritance in Ruby

What is most interesting here is that internally, Ruby implements module inclusion using class inheritance. Essentially, there is no difference between including a module and specifying a superclass. Both procedures make new

methods available to the target class, and both use the class's `super` pointer internally. Including multiple modules into a Ruby class is equivalent to specifying multiple superclasses.

Still, Ruby keeps things simple by enforcing a single list of ancestors. While including multiple modules does create multiple superclasses internally, Ruby maintains them in a single list. The result? As a Ruby developer, you get the benefits of multiple inheritance (adding new behavior to a class from as many different modules as you would like) while keeping the simplicity of the single inheritance model.

Ruby benefits from this simplicity as well! By enforcing this single list of superclass ancestors, its method lookup algorithm can be very simple. Whenever you call a method on an object, Ruby simply has to iterate through the superclass linked list until it finds the class or module that contains the target method.

The Global Method Cache

Depending on the number of superclasses in the chain, method lookup can be time consuming. To alleviate this, Ruby caches the result of a lookup for later use. It records which class or module implemented the method that your code called in two caches: a global method cache and an inline method cache.

Let's learn about the global method cache first. Ruby uses the *global method cache* to save a mapping between the receiver and implementer classes, as shown in Table 6-1.

Table 6-1: An Example of What the Global Method Cache Might Contain

`klass`	`defined_class`
`Fixnum#times`	`Integer#times`
`Object#puts`	`BasicObject#puts`
etc...	etc...

The left column in Table 6-1, `klass`, shows the receiver class; this is the class of the object you call a method on. The right column, `defined_class`, records the result of the method lookup. This is the implementer class, or the class that implements the method Ruby was looking for.

Let's take the first row of Table 6-1 as an example; it reads `Fixnum#times` and `Integer#times`. In the global method cache, this information means that Ruby's method lookup algorithm started to look for the `times` method in the `Fixnum` class but actually found it in the `Integer` class. In a similar way, the second row of Table 6-1 means that Ruby started to look for the `puts` method in the `Object` class but actually found it in the `BasicObject` class.

The global method cache allows Ruby to skip the method lookup process the next time your code calls a method listed in the first column of the global cache. After your code has called `Fixnum#times` once, Ruby knows that it can execute the `Integer#times` method, regardless of from where in your program you call `times`.

The Inline Method Cache

Ruby uses another type of cache, called an *inline method cache*, to speed up method lookup even more. The inline cache saves information alongside the compiled YARV instructions that Ruby executes (see Figure 6-8).

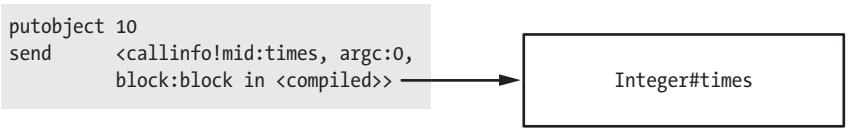

Figure 6-8: The YARV instructions on the left should call the implementation of `Integer#times` on the right.

On the left side of this figure, we see the compiled YARV instructions that correspond to the code `10.times do... end`. First, `putobject 10` pushes the `Fixnum` object `10` onto YARV's internal stack. This is the receiver of the `times` method call. Next, `send` calls the `times` method, as indicated by the text between the angle brackets.

The rectangle on the right side of the figure represents the `Integer#times` method, which Ruby found using its method lookup algorithm (after looking up the `times` method among the `Fixnum` class and its superclasses). Ruby's inline cache enables it to save the mapping between the `times` method call and the `Integer#times` implementation right in the YARV instructions. Figure 6-9 shows how the inline cache might look.

Figure 6-9: The inline cache saves the result of method lookup next to the send instruction that needs to call the method.

If Ruby executes this line of code again, it will immediately execute `Integer#times` without having to call the method lookup algorithm.

Clearing Ruby's Method Caches

Because Ruby is a dynamic language, you can define new methods when you like. In order for you to be able to do so, Ruby must clear the global and inline method caches, because the results of method lookups might change. For example, if we add a new definition of the `times` method to the `Fixnum` or `Integer` classes, Ruby would need to call the new `times` method, not the `Integer#times` method that it was previously using.

In effect, whenever you create or remove (*undefine*) a method, include a module into a class, or perform a similar action, Ruby clears the global and inline method caches, forcing a new call to the method lookup code. Ruby also clears the cache when you use refinements or employ other types

of metaprogramming. In fact, clearing the cache happens quite frequently in Ruby. The global and inline method caches might remain valid for only a short time.

Including Two Modules into One Class

While Ruby's method lookup algorithm may be simple, the code that it uses to include modules is not. As we saw above, when you include a module into a class, Ruby inserts a copy of the module into the class's ancestor chain. This means that if you include two modules, one after the other, the second module appears first in the ancestor chain and is found first by Ruby's method lookup logic.

For example, suppose we include two modules into `Mathematician`, as shown in Listing 6-6.

```
class Mathematician < Person
  include Professor
  include Employee
end
```

Listing 6-6: Including two modules into one class

Now `Mathematician` objects have methods from the `Professor` module, the `Employee` module, and the `Person` class. But which methods does Ruby find first and which methods override which?

Figures 6-10 and 6-11 show the order of precedence. Because we include the `Professor` module first, Ruby inserts the included class corresponding to the `Professor` module as a superclass first.

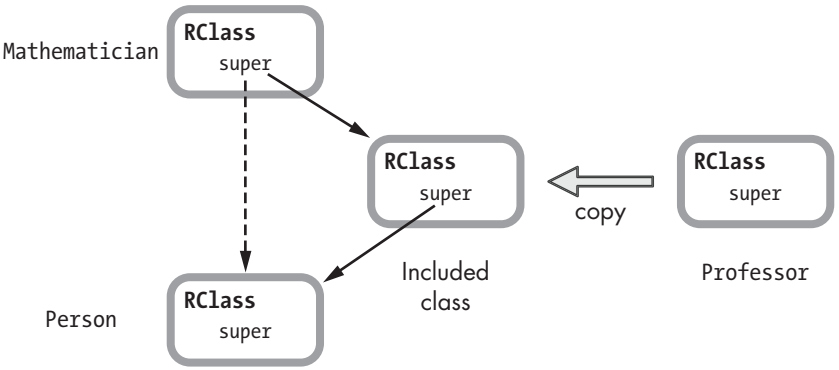

Figure 6-10: In Listing 6-6 we include the `Professor` module first.

Now, when we include the `Employee` module, the included class for the `Employee` module is inserted above the included class for the `Professor` module, as shown in Figure 6-11.

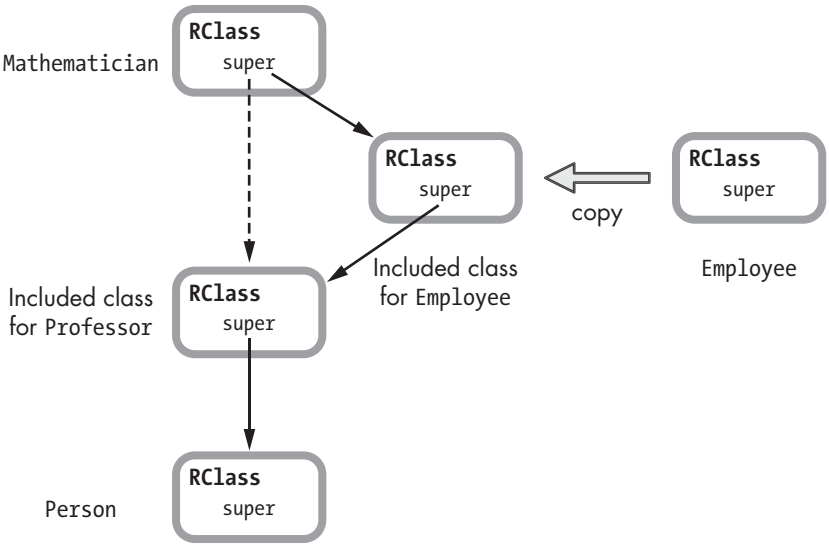

Figure 6-11: In Listing 6-6 we include the `Employee` module second, after including `Professor`.

Because `Employee` appears above `Professor` in the superclass chain, as shown along the left side of Figure 6-11, methods from `Employee` override methods from `Professor`, which in turn override methods from `Person`, the actual superclass.

Including One Module into Another

Modules don't allow you to specify superclasses. For example, we can't write the following:

```
module Professor < Employee
end
```

But we can include one module into another, as shown in Listing 6-7.

```
module Professor
  include Employee
end
```

Listing 6-7: One module including another module

What if we include `Professor`, a module with other modules included into it, into `Mathematician`? Which methods will Ruby find first? As shown in Figure 6-12, when we include `Employee` into `Professor`, Ruby creates a copy of `Employee` and sets it as the superclass of `Professor`.

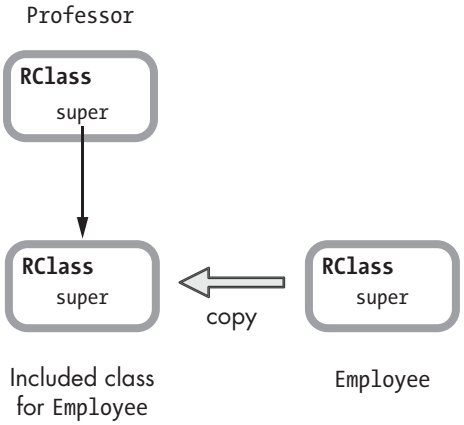

Figure 6-12: When you include one module into another, Ruby sets it as the superclass of the target module.

Modules can't have a superclass in your code, but they can inside Ruby because Ruby represents modules with classes internally!

Finally, when we include `Professor` into `Mathematician`, Ruby iterates over the two modules and inserts them both as superclasses of `Mathematician`, as shown in Figure 6-13.

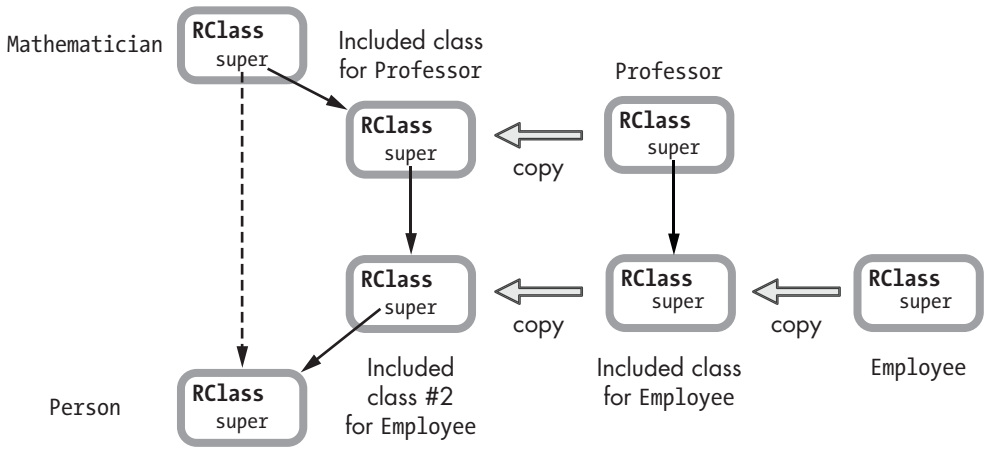

Figure 6-13: Including two modules into a class at the same time

Now Ruby will find the methods in `Professor` first and `Employee` second.

A Module#prepend Example

In Figure 6-2 we saw how Ruby includes a module into a class. Specifically, we saw how Ruby inserts a copy of the module's `RClass` structure into the superclass chain for the target class, between the class and its superclass.

Beginning with version 2.0, Ruby now allows you to "prepend" a module into a class. We'll use the `Mathematician` class to explain, as shown in Listing 6-8.

```
class Mathematician
❶  attr_accessor :name
end

poincaré = Mathematician.new
poincaré.name = "Henri Poincaré"
❷ p poincaré.name
 => "Henri Poincaré"
```

Listing 6-8: A simple Ruby class with a `name` attribute

First, we define the `Mathematician` class with just the single attribute `name` at ❶. Then, we create an instance of `Mathematician`, set its name, and display it at ❷.

Now suppose we make all of our mathematicians professors by including the `Professor` module into the `Mathematician` class again, as shown at ❶ in Listing 6-9.

```
module Professor
end

class Mathematician
  attr_accessor :name
❶  include Professor
end
```

Listing 6-9: Including the `Professor` module into the `Mathematician` class

Figure 6-14 shows the superclass chain for `Mathematician` and `Professor`.

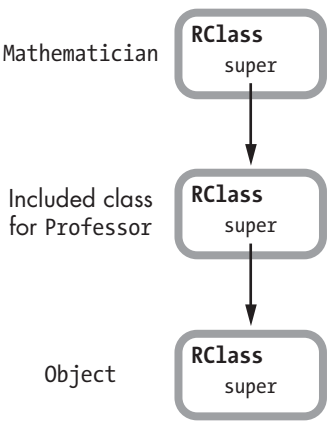

Figure 6-14: `Professor` is a superclass of `Mathematician`.

If we decide to display the title `Prof.` in front of each mathematician's name, we can just add that behavior to the `Mathematician` class, as shown in Listing 6-10.

```
module Professor
end

class Mathematician
  attr_writer :name
  include Professor
❶ def name
    "Prof. #{@name}"
  end
end
```

Listing 6-10: An ugly way to display the `Prof.` title before each mathematician's name

But this is a very ugly solution: The `Mathematician` class has to do the work of displaying the professor title at ❶. What if other classes include `Professor`? Shouldn't they display the `Prof.` title also? If `Mathematician` contains the code for showing `Prof.`, then any other classes that include `Professor` would be missing this code.

It makes more sense to include the code for displaying the title in the `Professor` module, as shown in Listing 6-11. This way every class that includes `Professor` will be able to display the title `Prof.` along with its class name.

```
module Professor
❶ def name
    "Prof. #{super}"
  end
end

class Mathematician
  attr_accessor :name
❷ include Professor
end

poincaré = Mathematician.new
poincaré.name = "Henri Poincaré"
❸ p poincaré.name
 => "Henri Poincaré"
```

Listing 6-11: How can we get Ruby to call the module's `name` method?

At ❶ we define a `name` method inside `Professor` that will display the `Prof.` title before the actual name (assuming that `name` is defined in a superclass). At ❷ we include `Professor` into `Mathematician`. Finally, at ❸ we call the `name` method, but we get the name `Henri Poincaré` without the `Prof.` title. What went wrong?

The problem, as shown in Figure 6-14, is that `Professor` is a superclass of `Mathematician`, not the other way around. This means when I call `poincaré.name` at ❸ in Listing 6-11, Ruby finds the `name` method from `Mathematician`, not from `Professor`. Figure 6-15 shows visually what Ruby's method lookup algorithm finds when I call `poincaré.name`.

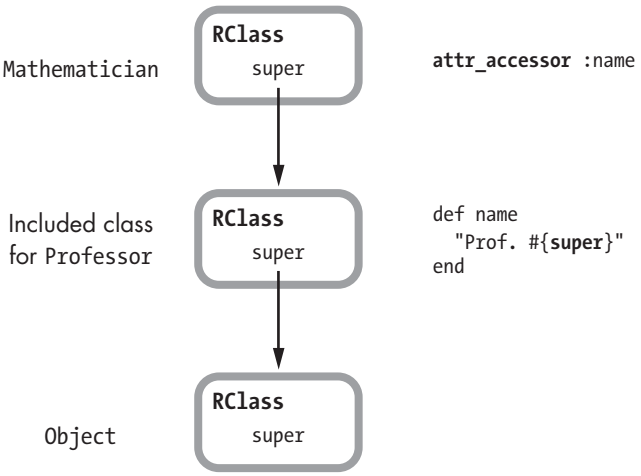

Figure 6-15: Ruby calls the `attr_accessor` method before finding the `name` method from `Professor`.

When we call `name` at ❸ in Listing 6-11, Ruby finds the first `name` method that it sees in the superclass chain starting from the top and moving down. As you can see in Figure 6-15, the first `name` method is the simple `attr_accessor` method in `Mathematician`.

However, if we prepend `Professor` instead of including it, we get the behavior we were hoping for, as shown in Listing 6-12.

```
module Professor
  def name
    "Prof. #{super}"
  end
end

class Mathematician
  attr_accessor :name
❶ prepend Professor
end

poincaré = Mathematician.new
poincaré.name = "Henri Poincaré"
❷ p poincaré.name
 => "Prof. Henri Poincaré"
```

Listing 6-12: Using `prepend`, Ruby finds the module's `name` method first.

The only difference here is the use of `prepend` at ❶.

How Ruby Implements Module#prepend

When you prepend a module to a class, Ruby places it before the class in the superclass chain, as shown in Figure 6-16.

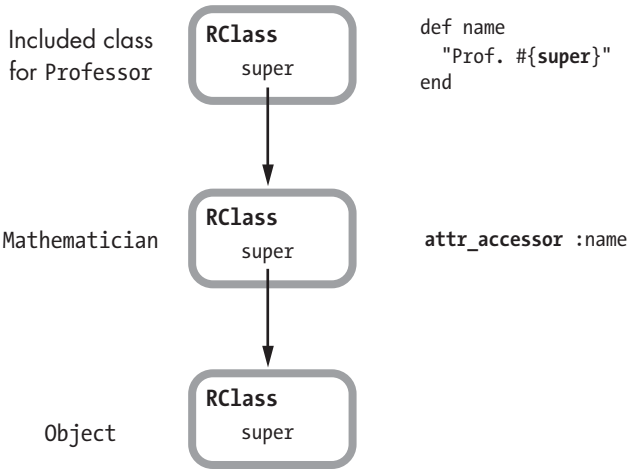

Figure 6-16: Using prepend, Ruby places the module before the target class in the superclass chain.

But there is something odd here. When we call `name` on a mathematician object, how does Ruby find the module's method? That is, at ❷ in Listing 6-12, we're calling `name` on the `Mathematician` class, not on the `Professor` module. Ruby should find the simple `attr_accessor` method, not the version from the module, but that's not the case. Does Ruby look backward up the superclass chain to find the module? If so, how does it do this when the `super` pointers point down?

The secret is that internally Ruby uses a trick to make it seem as if `Mathematician` is the superclass of `Professor` when it's not, as shown in Figure 6-17. Prepending a module is like including a module. `Mathematician` is at the top of the superclass chain, and moving down the chain, we see that Ruby still sets the included class for `Professor` to be the superclass of `Mathematician`.

But below `Professor` in Figure 6-17 we see something new, the *origin class* for `Mathematician`. This is a new copy of `Mathematician` that Ruby creates to make `prepend` work.

When you prepend a module, Ruby creates a copy of the target class (called the *origin class* internally) and sets it as the superclass of the prepended module. Ruby uses the `origin` pointer that we saw in the `rb_classext_struct` structure in Figures 6-1 and 6-2 to track this new origin copy of the class. In addition, Ruby moves all of the methods from the original class to the origin class, which means that those methods may now be overridden by methods with the same name in the prepended module. In Figure 6-17 you can see that Ruby moved the `attr_accessor` method down from `Mathematician` to the origin class.

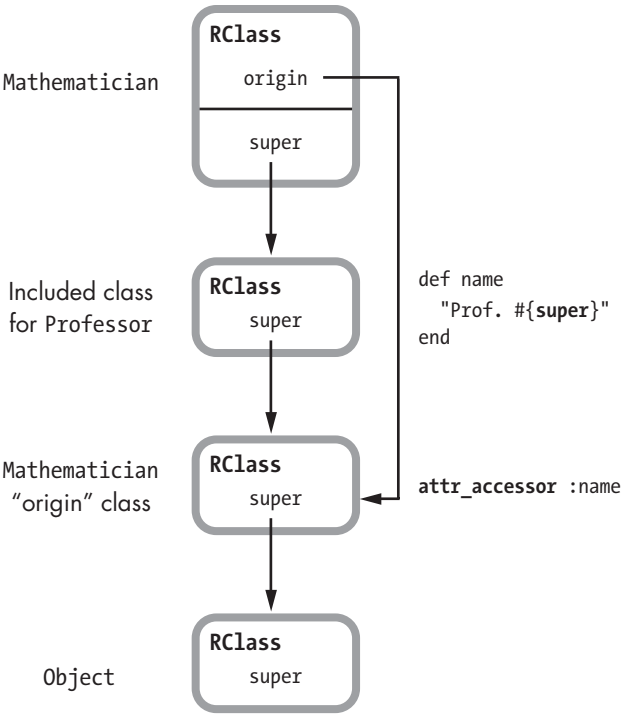

Figure 6-17: Ruby creates a copy of the target class and sets it as the superclass of the prepended module.

Experiment 6-1: Modifying a Module After Including It

Following a suggestion by Xavier Noria, this experiment will explore what happens when you modify a module once it's been included into a class. We'll use the same `Mathematician` class and the `Professor` module but with different methods, as shown in Listing 6-13.

```
module Professor
  def lectures; end
end

class Mathematician
❶   attr_accessor :first_name
    attr_accessor :last_name
❷   include Professor
end
```

Listing 6-13: Another example of including a module into a class

This time the Mathematician class contains the accessor methods at ❶ for @first_name and @last_name, and we've included the Professor module again at ❷. If we inspect the methods of a mathematician object, as shown in Listing 6-14, we should see the attribute methods, such as first_name= and the lectures method from Professor.

```
fermat = Mathematician.new
fermat.first_name = 'Pierre'
fermat.last_name = 'de Fermat'

p fermat.methods.sort
 => [ ... :first_name, :first_name=, ... :last_name, :last_name=, :lectures ... ]
```

Listing 6-14: Inspecting the methods of a mathematician object

No surprise; we see all the methods.

Classes See Methods Added to a Module Later

Now let's add some new methods to the Professor module after including it into the Mathematician class. Does Ruby know that the new methods should be added to Mathematician as well? Let's find out by running Listing 6-15 right after Listing 6-14 finishes.

```
module Professor
  def primary_classroom; end
end

p fermat.methods.sort
❶ => [ ... :first_name, :first_name=, ... :last_name, :last_name=, :lectures,
... :primary_classroom, ... ]
```

Listing 6-15: Adding a new method to Professor after including it into Mathematician

As you can see, at ❶ we get all the methods, including the new primary_classroom method that was added to Professor after it was included into Mathematician. No surprise here either. Ruby is one step ahead of us.

Classes Don't See Submodules Included Later

Now for one more test. What if we reopen the Professor module and include yet another module into it using Listing 6-16?

```
module Employee
  def hire_date; end
end

module Professor
  include Employee
end
```

Listing 6-16: Including a new module into Professor after it was included into Mathematician

This is getting confusing, so let's review what we did in Listings 6-13 and 6-16:

- In Listing 6-13 we included the `Professor` module into the `Mathematician` class.
- Then, in Listing 6-16 we included the `Employee` module into the `Professor` module. Therefore, the methods of the `Employee` module should now be available on a mathematician object.

Let's see whether Ruby works as expected:

```
p fermat.methods.sort
 => [ ... :first_name, :first_name=, ... :last_name, :last_name=, :lectures ... ]
```

It didn't work! The `hire_date` method is *not* available in the `fermat` object. Including a module into a module already included into a class *does not* affect that class.

As we've learned how Ruby implements modules, this fact shouldn't be too hard to understand. Including `Employee` into `Professor` changes the `Professor` module, not the copy of `Professor` that Ruby created when we included it into `Mathematician`, as shown in Figure 6-18.

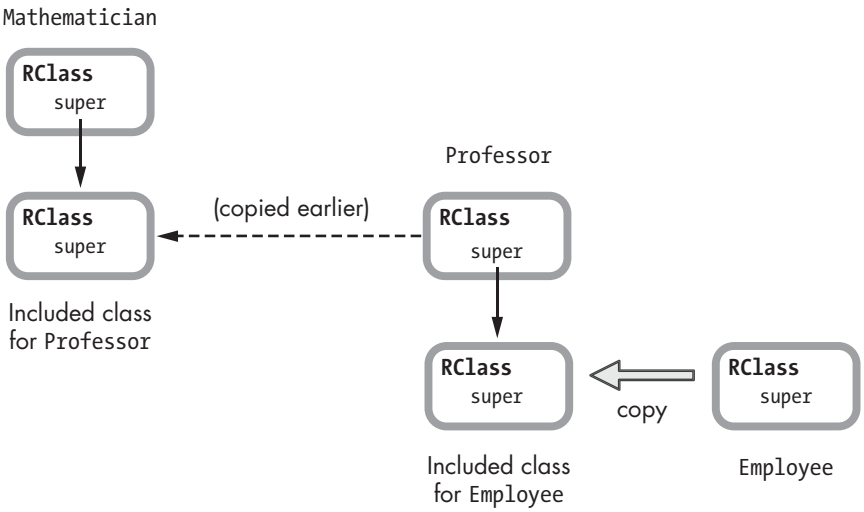

Figure 6-18: The `Employee` module is included into the original `Professor` module, not the included copy used by `Mathematician`.

Included Classes Share the Method Table with the Original Module

But what about the `primary_classroom` method we added in Listing 6-15? How was Ruby able to include the `primary_classroom` method into `Mathematician` even though we added it to `Professor` after we included `Professor` into `Mathematician`? Figure 6-18 shows that Ruby created a copy of the `Professor` module before we added the new method to it. But how does the `fermat` object get the new method?

As it turns out, when you include a module, Ruby copies the `RClass` structure, not the underlying method table, as shown in Figure 6-19.

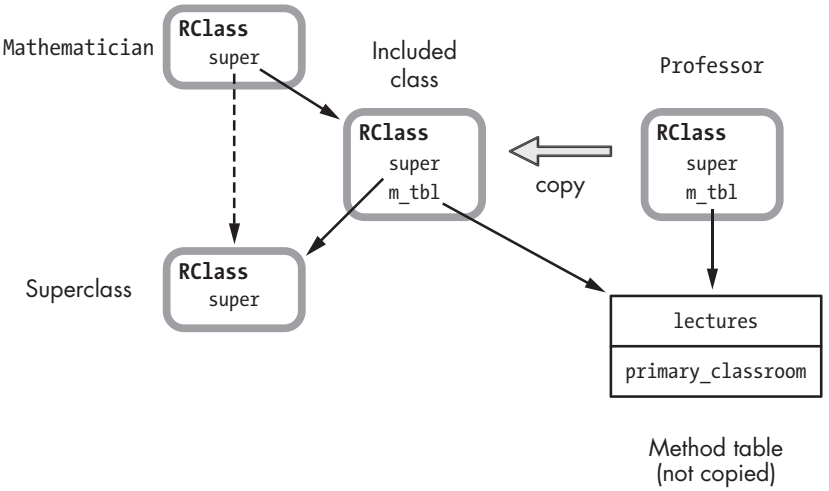

Figure 6-19: Ruby doesn't copy the method table when you include a module.

Ruby doesn't copy the method table for `Professor`. Instead, it simply sets `m_tbl` in the new copy of `Professor`, the "included class," to point to the same method table. This means that modifying the method table by reopening the module and adding new methods will change both the module and any classes into which it was already included.

A CLOSE LOOK AT HOW RUBY COPIES MODULES

By looking at Ruby's C source code directly, you'll gain a precise understanding of how Ruby copies modules when you include them and why Ruby behaves as you'll see in this experiment. You'll find the C function that Ruby uses to make a copy of a module in the *class.c* file. Listing 6-17 shows a portion of the function `rb_include_class_new`.

```
VALUE
❶ rb_include_class_new(VALUE module, VALUE super)
{
❷     VALUE klass = class_alloc(T_ICLASS, rb_cClass);
      --snip--
❸     RCLASS_IV_TBL(klass) = RCLASS_IV_TBL(module);
      RCLASS_CONST_TBL(klass) = RCLASS_CONST_TBL(module);
❹     RCLASS_M_TBL(klass) = RCLASS_M_TBL(RCLASS_ORIGIN(module));
❺     RCLASS_SUPER(klass) = super;
      --snip--
      return (VALUE)klass;
}
```

Listing 6-17: A portion of the `rb_include_class_new` C function, from class.c

At ❶ Ruby passes in `module` (the target module to copy) and `super` (the superclass to use for the new copy of `module`). By specifying a particular superclass, Ruby inserts the new copy into the superclass chain at a particular place. If you search *class.c* for `rb_include_class_new`, you'll find that Ruby calls it from another C function, `include_modules_at`, which handles the complex internal logic that Ruby uses to include modules.

At ❷ Ruby calls `class_alloc` to create a new `RClass` structure and saves a reference to it in `klass`. Notice the first parameter to `class_alloc` is the value `T_ICLASS`, which identifies the new class as an included class. Ruby uses `T_ICLASS` throughout its C source code when dealing with included classes.

At ❸ Ruby copies a series of pointers from the original module's `RClass` structure over to the new copy using three C macros that operate on `RClass`.

- `RCLASS_IV_TBL` gets or sets a pointer to the instance variable table.
- `RCLASS_CONST_TBL` gets or sets a pointer to the constant variable table.
- `RCLASS_M_TBL` gets or sets a pointer to the method table.

For example, `RCLASS_IV_TBL(klass) = RCLASS_IV_TBL(module)` sets the instance variable table pointer in `klass` (the new copy) to the instance variable pointer from `module` (the target module to copy). Now `klass` and `module` use the same instance variables. In the same way, `klass` shares constant and method tables with `module`. Because they share the same method table, adding a new method to `module` also adds it to `klass`. This explains the behavior we saw in Experiment 6-1: Adding a method to a module also adds it to each class that includes that module.

Also note at ❹ Ruby uses `RCLASS_ORIGIN(module)`, not `module`. Normally `RCLASS_ORIGIN(module)` is the same as `module`; however, if you have earlier used `prepend` in `module`, then `RCLASS_ORIGIN(module)` instead returns the origin class for `module`. Recall that when you call `Module#prepend`, Ruby makes a copy (the origin class) of the target module and inserts the copy into the superclass chain. By using `RCLASS_ORIGIN(module)`, Ruby gets the original module's method table, even if you prepended it with a different module.

Finally, at ❺ Ruby sets the superclass pointer of `klass` to the specified superclass and returns it.

Constant Lookup

We've learned about Ruby's method lookup algorithm and how it searches through the superclass chain to find the right method to call. Now we'll turn our attention to a related process: Ruby's constant lookup algorithm, or the process Ruby uses to find a constant value that you refer to in your code.

Clearly method lookup is central to the language, but why study constant lookup? As Ruby developers, we don't use constants very often in our code—certainly not as often as we use classes, modules, variables, and blocks.

One reason is that constants, like modules and classes, are central to the way Ruby works internally and to the way we use Ruby. Whenever you define a module or class, you also define a constant. And whenever you refer to or use a module or class, Ruby has to look up the corresponding constant.

The second reason has to do with the way Ruby finds a constant that you refer to in your code. As you may know, Ruby finds constants defined in a superclass, but it also finds constants in the surrounding namespace or syntactical scope of your program. Studying how Ruby handles syntactical scope leads us to some important discoveries about how Ruby works internally.

Let's begin by reviewing how constants work in Ruby.

Finding a Constant in a Superclass

One way that Ruby searches for the definition of a constant you refer to is by using the superclass chain just as it would look for a method definition. Listing 6-18 shows an example of one class finding a constant in its superclass.

```
class MyClass
❶  SOME_CONSTANT = "Some value..."
end

❷ class Subclass < MyClass
    p SOME_CONSTANT
end
```

Listing 6-18: Ruby finds constants you define in a superclass.

In Listing 6-18 we define `MyClass` with a single constant, `SOME_CONSTANT` at ❶. Then we create `Subclass` and set `MyClass` as a superclass at ❷. When we print the value of `SOME_CONSTANT`, Ruby uses the same algorithm it uses to find a method, as shown in Figure 6-20.

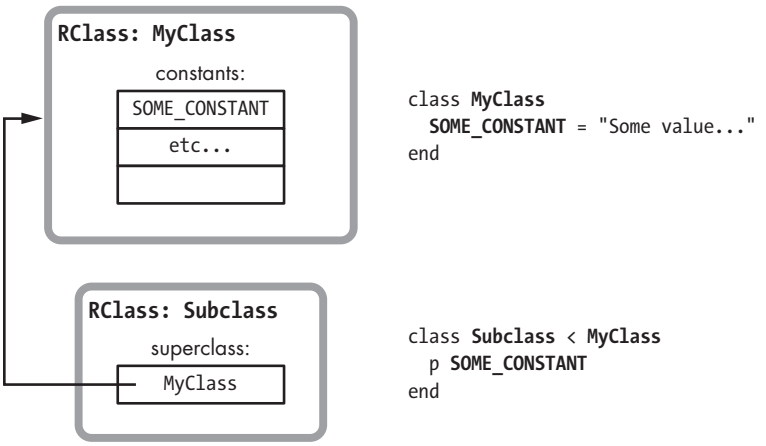

Figure 6-20: Ruby searches for constants using the superclass chain, just as it does with methods.

Here, on the right, you see the code from Listing 6-18 and, on the left, the RClass structures that correspond to each of the two classes we created. At the top left of the figure, you see MyClass, which contains the value of SOME_CONSTANT in its constants table. Below that is Subclass. When we refer to SOME_CONSTANT from inside Subclass, Ruby uses the super pointer to find MyClass and the value of SOME_CONSTANT.

How Does Ruby Find a Constant in the Parent Namespace?

Listing 6-19 shows another way to define a constant.

```
❶ module Namespace
❷   SOME_CONSTANT = "Some value..."
❸   class Subclass
❹     p SOME_CONSTANT
    end
end
```

Listing 6-19: Using a constant defined in the surrounding namespace

Using idiomatic Ruby style, we create a module called Namespace at ❶. Then, inside this module, we declare the same SOME_CONSTANT value at ❷. Next, we declare Subclass inside Namespace at ❸, and we're able to refer to and print the value of SOME_CONSTANT, just as in Listing 6-18.

But how does Ruby find SOME_CONSTANT in Listing 6-19 when we display it at ❹? Figure 6-21 shows the problem.

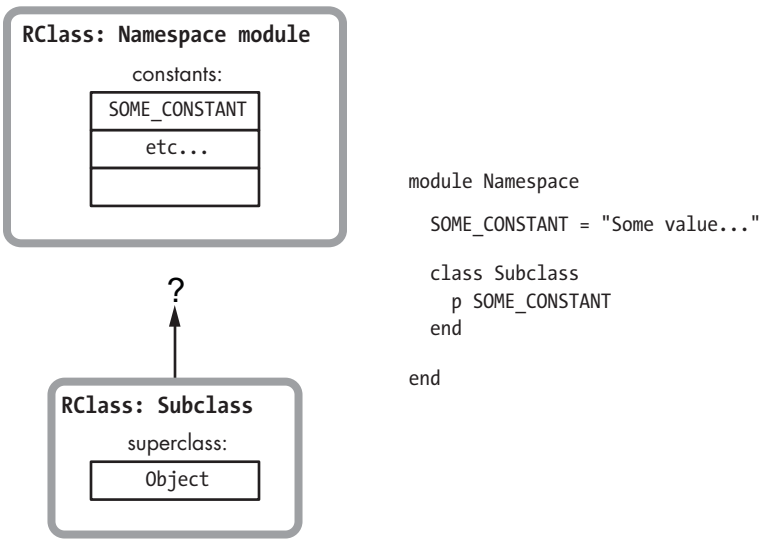

Figure 6-21: How does Ruby find constants in the surrounding namespace?

On the left side of this figure are two RClass structures, one for the Namespace module and one for Subclass. Notice that Namespace is not a superclass of Subclass; the super pointer in Subclass refers to the Object class, Ruby's default superclass. Then how does Ruby find SOME_CONSTANT when we refer to it inside of Subclass? Somehow Ruby allows you to search up the "namespace chain" to find constants. This behavior is called using lexical scope to find a constant.

Lexical Scope in Ruby

Lexical scope refers to a section of code within the syntactical structure of your program, rather than within the superclass hierarchy or some other scheme. For example, suppose we use the class keyword to define MyClass, as shown in Listing 6-20.

```
class MyClass
  SOME_CONSTANT = "Some value..."
end
```

Listing 6-20: Defining a class with the class keyword

This code tells Ruby to create a new copy of the RClass structure, but it also defines a new scope or syntactical section of your program. This is the area between the class and end keywords, as shown with shading in Figure 6-22.

```
class MyClass

  SOME_CONSTANT = "Some value..."

end
```

Figure 6-22: The class keyword creates a class and a new lexical scope.

Think of your Ruby program as a series of scopes, one for each module or class that you create and another for the default, top-level lexical scope. To keep track of where this new scope lies inside your program's lexical structure, Ruby attaches a couple of pointers to the YARV instruction snippet corresponding to the code it compiles inside this new scope, as shown in Figure 6-23.

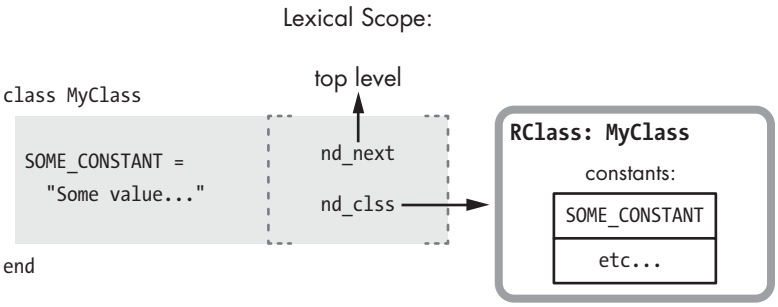

Figure 6-23: For each snippet of compiled code, Ruby uses pointers to track the parent lexical scope and the current class or module.

This figure shows the lexical scope information attached to the right side of the Ruby code. There are two important values here:

- First, the `nd_next` pointer is set to the parent or surrounding lexical scope—the default or top-level scope in this case.
- Next, the `nd_clss` pointer indicates which Ruby class or module corresponds to this scope. In this example, because we just defined `MyClass` using the `class` keyword, Ruby sets the `nd_clss` pointer to the `RClass` structure corresponding to `MyClass`.

Creating a Constant for a New Class or Module

Whenever you create a class or module, Ruby automatically creates a corresponding constant and saves it in the class or module for the parent lexical scope.

Let's return to the "namespace" example from Listing 6-19. Figure 6-24 shows what Ruby does internally when you create `MyClass` inside `Namespace`.

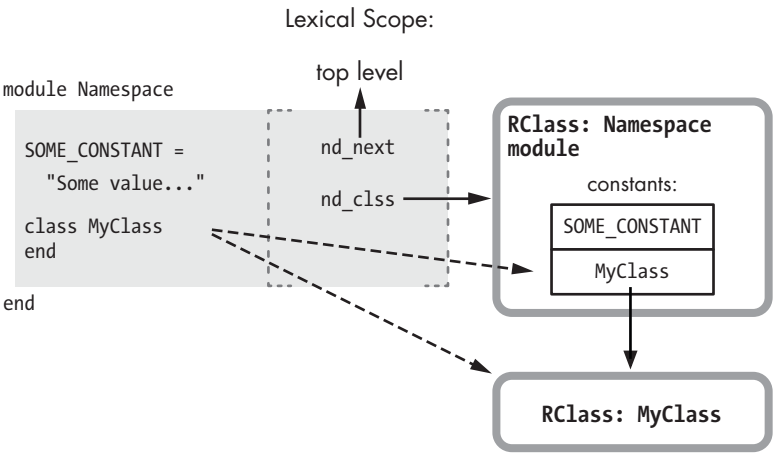

Figure 6-24: When you declare a new class, Ruby creates a new `RClass` structure and defines a new constant set to the new class's name.

The dashed arrows in this figure show what actions Ruby takes when you create a new class or module:

- First, Ruby creates a new `RClass` structure for the new module or class, as shown at the bottom.
- Then, Ruby creates a new constant using the new module or class name and saves it inside the class corresponding to the parent lexical scope. Ruby sets the value of the new constant to be a reference or pointer to the new `RClass` structure. In Figure 6-24 you can see that the `MyClass` constant appears in the constants table for the `Namespace` module.

The new class also gets its own new lexical scope, as shown in Figure 6-25.

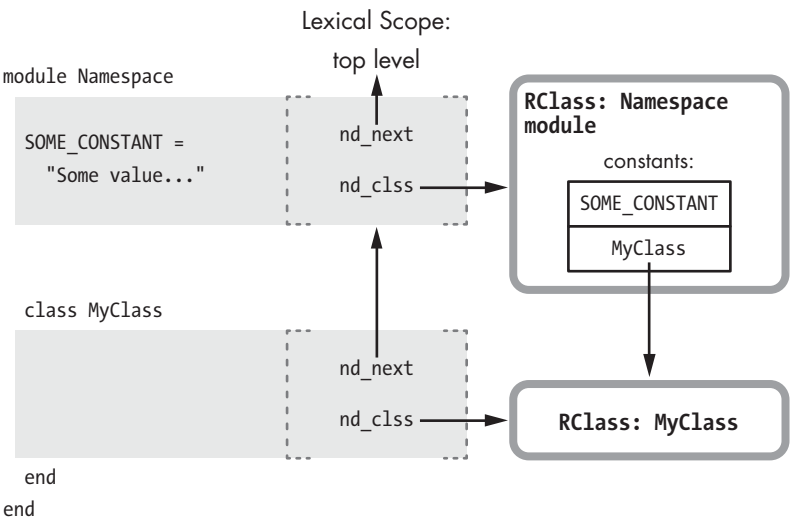

Figure 6-25: A new class also gets its own lexical scope, shown here as the second shaded rectangle.

This figure shows a new shaded rectangle for the new scope. Its `nd_clss` pointer is set to the new `RClass` structure for `MyClass`, and its `nd_next` pointer is set to the parent scope that corresponds to the `Namespace` module.

Finding a Constant in the Parent Namespace Using Lexical Scope

In Listing 6-21 let's return to the example from Listing 6-19, which prints the value of `SOME_CONSTANT`.

```
module Namespace
  SOME_CONSTANT = "Some value..."
  class Subclass
❶   p SOME_CONSTANT
  end
end
```

Listing 6-21: Finding a constant in the parent lexical scope (repeated from Listing 6-19)

In Figure 6-20 we saw how Ruby iterates over `super` pointers to find a constant from a superclass. But in Figure 6-21 we saw that Ruby couldn't use `super` pointers to find `SOME_CONSTANT` in this example because `Namespace` is not a superclass of `MyClass`. Instead, as Figure 6-26 shows, Ruby can use the `nd_next` pointers to iterate up through your program's lexical scopes in search of constant values.

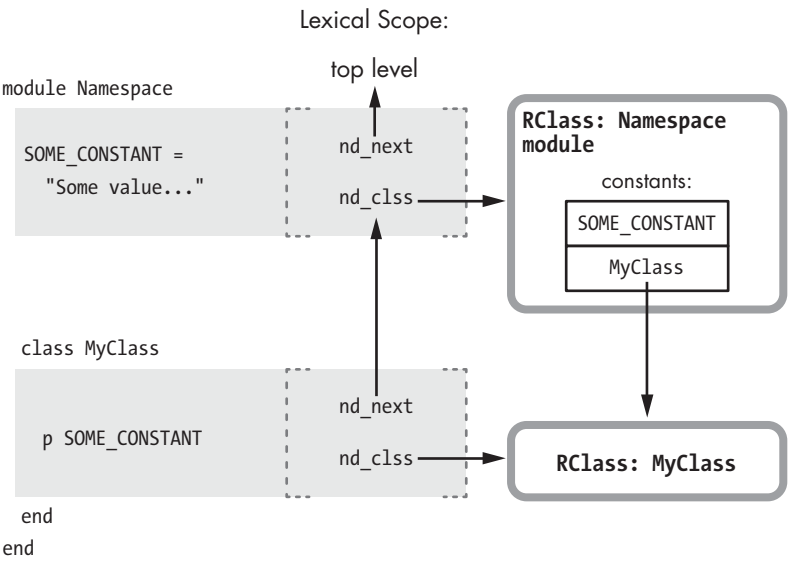

Figure 6-26: Ruby can find `SOME_CONSTANT` in the parent lexical scope using the `nd_next` and `nd_clss` pointers.

By following the arrows in this figure, you can see how the `p SOME_CONSTANT` command at ❶ in Listing 6-21 works:

- First, Ruby looks for the value of `SOME_CONSTANT` in the current scope's class, `MyClass`. In Figure 6-26 the current scope contains the `p SOME_CONSTANT` code. You can see how Ruby finds the current scope's class on the right using the `nd_clss` pointer. Here, `MyClass` has nothing in its constants table.
- Next, Ruby finds the parent lexical scope using the `nd_next` pointer, moving up Figure 6-26.
- Ruby repeats the process, searching the current scope's class using the `nd_clss` pointer. This time the current scope's class is the `Namespace` module, at the top right of Figure 6-26. Now Ruby finds `SOME_CONSTANT` in `Namespace`'s constants table.

Ruby's Constant Lookup Algorithm

The flowchart in Figure 6-27 summarizes how Ruby iterates over the lexical scope chain while looking for constants.

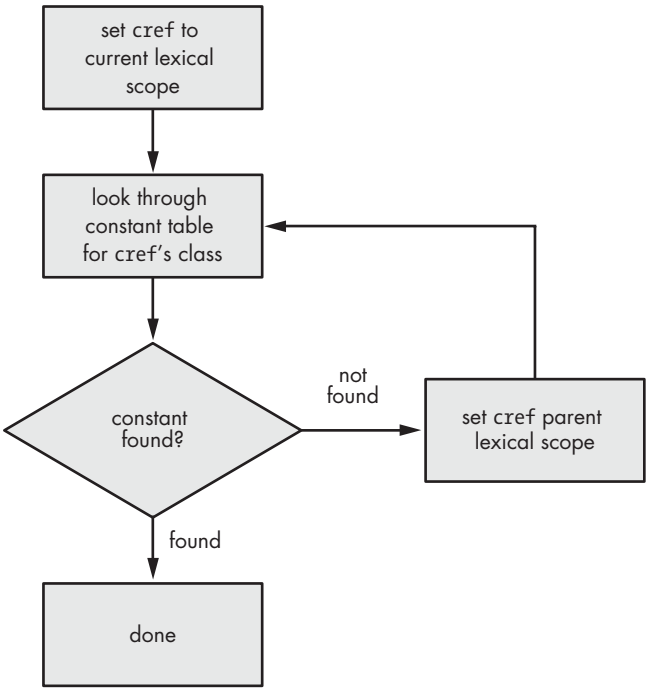

Figure 6-27: Part of Ruby's constant lookup algorithm

Notice that this figure is very similar to Figure 6-3. Ruby iterates over the linked list formed by the `nd_next` pointers in each lexical scope while looking for a constant, just as it iterates over the `super` pointers while looking for a method. Ruby uses superclasses to find methods and parent lexical scopes to find constants.

However, this is just part of Ruby's constant lookup algorithm. As we saw earlier in Figure 6-20, Ruby also looks through superclasses for constants.

Experiment 6-2: Which Constant Will Ruby Find First?

We've just learned that Ruby iterates over a linked list of lexical scopes in order to look up constant values. However, we saw earlier in Figure 6-20 that Ruby also uses the superclass chain to look up constants. Let's use Listing 6-22 to see how this works in more detail.

```
class Superclass
❶  FIND_ME = "Found in Superclass"
end
```

```
module ParentLexicalScope
❷  FIND_ME = "Found in ParentLexicalScope"

  module ChildLexicalScope

    class Subclass < Superclass
      p FIND_ME
    end

  end
end
```

Listing 6-22: Does Ruby search the lexical scope chain first? Or does it search the superclass chain first? (find-constant.rb)

Notice here that I've defined the constant `FIND_ME` twice—at ❶ and at ❷. Which constant will Ruby find first? Will Ruby first iterate over the lexical scope chain and find the constant at ❷? Or will it iterate over the superclass chain and find the constant value at ❶?

Let's find out! When we run Listing 6-22, we get the following:

```
$ ruby find-constant.rb
"Found in ParentLexicalScope"
```

You can see that Ruby looks through the lexical scope chain first.

Now let's comment out the second definition at ❷ in Listing 6-22 and try the experiment again:

```
module ParentLexicalScope
❷  #FIND_ME = "Found in ParentLexicalScope"
```

When we run the modified Listing 6-22, we get the following:

```
$ ruby find-constant.rb
"Found in Superclass"
```

Because now there is only one definition of `FIND_ME`, Ruby finds it by iterating over the superclass chain.

Ruby's Actual Constant Lookup Algorithm

Unfortunately, things aren't quite so simple; there are some other quirks in Ruby's behavior with regard to constants. Figure 6-28 is a simplified flowchart showing Ruby's entire constant lookup algorithm.

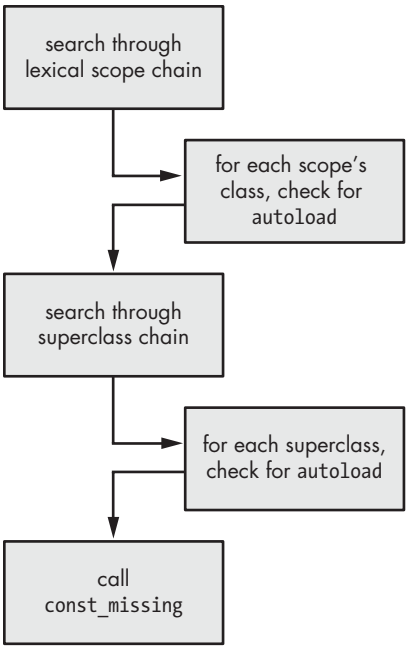

Figure 6-28: A high-level summary of Ruby's constant lookup algorithm

At the top, you can see that Ruby begins by iterating up the lexical scope chain, as we saw in Listing 6-22. Ruby always finds constants, including classes or modules, that are defined in a parent lexical scope. However, as Ruby iterates up the scope chain, it looks to see whether you used the `autoload` keyword, which instructs it to open and read in a new code file if a given constant is undefined. (The Rails framework uses `autoload` to allow you to load models, controllers, and other Rails objects without having to use `require` explicitly.)

If Ruby loops through the entire lexical scope chain without finding the given constant or a corresponding `autoload` keyword, it then iterates up the superclass chain, as we saw in Listing 6-18. This allows you to load constants defined in a superclass. Ruby once again honors any `autoload` keyword that might exist in any of those superclasses, loading an additional file if necessary.

Finally, if all else fails and the constant still isn't found, Ruby calls the `const_missing` method on your module if you provided one.

Summary

In this chapter we've learned two very different ways to look at your Ruby program. On the one hand, you can organize your code by class and superclass, and on the other, you can organize it by lexical scope. We saw how internally Ruby uses different sets of C pointers to keep track of these two trees as it executes your program. The `super` pointers found in the `RClass` structures form the superclass tree, while the `nd_next` pointers from the lexical scope structures form the namespace or lexical scope tree.

We studied two important algorithms that use these trees: how Ruby looks up methods and constants. Ruby uses the class tree to find the methods that your code (and Ruby's own internal code) calls. Similarly, Ruby uses both the lexical scope tree and the superclass hierarchy to find constants that your code refers to. Understanding the method and constant lookup algorithms is essential. They allow you to design your program and organize your code using these two trees in a way that is appropriate for the problem you are trying to solve.

At first glance, these two organizational schemes seem completely orthogonal, but in fact they are closely related by the way Ruby's classes behave. When you create a class or module, you add both to the superclass and lexical scope hierarchy, and when you refer to a class or superclass, you instruct Ruby to look up a particular constant using the lexical scope tree.